Automated Software Engineering (2024) 31:30
https://doi.org/10.1007/s10515-024-00431-2

Bash comment generation via data augmentation and semantic-aware CodeBERT

Yiheng Shen[1] · Xiaolin Ju[1] · Xiang Chen[1] · Guang Yang[2]

Received: 25 October 2023 / Accepted: 3 March 2024

Abstract
Understanding Bash code is challenging for developers due to its syntax flexibility and unique features. Bash lacks sufficient training data compared to comment generation tasks in popular programming languages. Furthermore, collecting more real Bash code and corresponding comments is time-consuming and labor-intensive. In this study, we propose a two-module method named Bash2Com for Bash code comments generation. The first module, NP-GD, is a gradient-based automatic data augmentation component that enhances normalization stability when generating adversarial examples. The second module, MASA, leverages CodeBERT to learn the rich semantics of Bash code. Specifically, MASA considers the representations learned at each layer of CodeBERT as a set of semantic information that captures recursive relationships within the code. To generate comments for different Bash snippets, MASA employs LSTM and attention mechanisms to dynamically concentrate on relevant representational information. Then, we utilize the Transformer decoder and beam search algorithm to generate code comments. To evaluate the effectiveness of Bash2Com, we consider a corpus of 10,592 Bash code and corresponding comments. Compared with the state-of-the-art baselines, our experimental results show that Bash2Com can outperform all baselines by at least 10.19%, 11.81%, 2.61%, and 6.13% in terms of the performance measures BLEU-3/4, METEOR, and ROUGR-L. Moreover, the rationality of NP-GD and MASA in Bash2Com are verified by ablation studies. Finally, we conduct a human evaluation to illustrate the effectiveness of Bash2Com from practitioners' perspectives.

Keywords Bash code · Code comment generation · Adversarial training · Data augmentation

Extended author information available on the last page of the article

Published online: 26 March 2024

1 Introduction

Bash, the default programming language of Linux shell, is a crucial tool for developing and maintaining the Linux operating system (Newham 2005). However, its syntax flexibility and lack of explicit structure can pose challenges for developers seeking to understand Bash scripts in the context of developing and maintaining software systems (Lin et al. 2018). According to our statistics, more than 151,640 posts on Stack Overflow are searching for the keyword “Bash" until February 22, 2023. For instance, Fig. 1 shows a post[1] which indicates that the user cannot understand a Bash snippet. The expression -z string is true if the length of the string is zero. This is difficult for people who are just engaged in software development and are not experienced developers in this field. Automatically generating comments for Bash code is necessary due to the difficulty in understanding the semantics of Bash code, especially for developers unfamiliar with Bash code.

Program understanding is crucial for software development, maintenance, and reuse. In practice, developers usually spend an average of 59% of their time on understanding programs (Xia et al. 2017). Naturally, high-quality comments can improve program understanding efficiency significantly (He 2019). Existing automatic code comment generation work focuses on popular programming languages

Fig. 1 A Stack Overflow post on understanding of a Bash snippet

[1] https://stackoverflow.com/questions/18096670/what-does-z-mean-in-bash

(such as Java and Python, and so on) (Liu et al. 2018; Hu et al. 2018; Yang et al. 2021, 2022). However, few works of comment generation focus on domain-specific languages (such as Bash Yu et al. 2022), which should be solved urgently.

Compared to other programming languages, generating code comments for Bash is challenging due to insufficient training data. The current corpus available for studying Bash comment generation is limited to 10,592 data pairs, as reported in a recent study by Yu et al. (2022). To overcome this limitation, more realistic Bash code and corresponding comments must be collected, which can be time-consuming and labor-intensive. To address this issue, we propose Normalized Projected Gradient Descent(NP-GD), a gradient-based adversarial training that uses generated adversarial examples to augment the training data and improve model performance. Previous studies have not investigated whether adversarial training in the Bash comment generation task can enhance model performance (Lin et al. 2018, 2017; Yu et al. 2022).

Moreover, previous studies have failed to fully leverage Bash code's semantic representation information. Specifically, BASHEXPLAINER (Yu et al. 2022) used the vector output of the last layer of the 12-layer CodeBERT (Feng et al. 2020) as the coding vector of this sentence. However, Kondratyuk and Straka (2019) found that relying solely on the output vector of the final layer may result in the loss of valuable information that other layers, such as the lower and middle layers of BERT can capture. According to Jawahar et al. (2019), the level of semantic information decreases as the encoder layer becomes shallower in CodeBERT. In contrast, the level of semantic information increases as the encoder layer becomes deeper. Su and Cheng (2020) further investigated this phenomenon and found that each layer of BERT offers a rich source of linguistic information through its output representation. Lower layers capture surface features, middle layers capture syntactic features, and higher layers capture semantic features. To address the issue of insufficient utilization of semantic information in Bash code, we propose a solution called Memory Attention Semantic-Aware (MASA). This method fully captures the learned information and enhances CodeBERT's semantic awareness.

In this study, we propose a novel method Bash2Com for generating Bash code comments. Specifically, Bash2Com includes two mentioned components: the data augmentation component NP-GD and the semantic-aware component MASA. In particular, the adversarial examples generated by NP-GD are added to the training data, which allows the model to learn more knowledge from the adversarial examples to improve generalization. MASA learns a rich semantic representation fed into the Transformer decoder. Afterward, MASA utilizes the beam search algorithm (Wiseman and Rush 2016) to generate comments for Bash code.

To evaluate the effectiveness of Bash2Com, we conducted experiments on the corpus shared by Yu et al. (2022). Our evaluation includes a recent Bash comment generation baseline proposed by Yu et al. (2022), as well as state-of-the-art baselines from the source code summarization research domain including information retrieval methods (Haiduc et al. 2010a, b; Zhang et al. 2020; Liu et al. 2018), deep learning methods (Ahmad et al. 2020; Iyer et al. 2016; Feng et al. 2020; Guo et al. 2022; Phan et al. 2021; Ahmad et al. 2021; Wang et al. 2021) and hybrid methods (Hu et al. 2020; Zhang et al. 2020; Yu et al. 2022). The results of both the evaluation

and the case study demonstrate that Bash2Com outperforms all the baseline methods. In the next step, we perform ablation research to verify the rationality of the component settings in our proposed method. Specifically, we consider four methods (i.e., FGSM Goodfellow et al. 2015, FGM Miyato et al. 2017, PGD Madry et al. 2018, and NP-GD) in the data augmentation component and consider three different RNNs (i.e., GRU Cho et al. 2014, Bi-LSTM Graves and Schmidhuber 2005, and LSTM Hochreiter and Schmidhuber 1996) in the semantic-aware component. The results of our ablation experiments indicate that the best performance is achieved by selecting NP-GD and LSTM. Finally, we conducted a human evaluation to demonstrate that Bash2Com outperforms baseline methods from practitioners' perspectives regarding informativeness, naturalness, and similarity.

In summary, the main contributions of our work can be summarized as follows:

- We propose a novel Bash comment generation method Bash2Com, which includes two components NP-GD and MASA. Specifically, NP-GD enhances the data representation, and MASA improves the semantic information representation learned by each layer in CodeBERT.
- We conduct experiments on a high-quality corpus shared by Yu et al. (2022), and the empirical results show our method's effectiveness and the rationality of the component settings.
- We share empirical corpus and scripts on our project homepage[2] to promote the replication of our research and encourage more follow-up research on this research topic.

The rest of this paper is organized as follows: Sect. 2 offers some background on data augmentation technology and CodeBERT. Section 3 describes the framework and details of our proposed method. Section 4 shows our empirical settings. Section 5 presents our result analysis for research questions. Section 6 analyzes potential threats to our empirical results. Section 7 summarizes related studies to our work and emphasizes the novelty of our study. Finally, Sect. 8 summarizes our work and shows potential future directions.

2 Research background

2.1 Data augmentation in NLP

In natural language processing (NLP), typical data augmentation methods can be categorized as rule-based (Wei and Zou 2019; Xie et al. 2020) and gradient-based (Goodfellow et al. 2015; Miyato et al. 2017). The method of data augmentation based on rules is easy to use, but it has certain limitations. For instance, it requires manual rule setting, which results in less coverage and poor diversity. While these methods are effective for natural languages, there are significant syntactic differences between Bash code and

[2] https://github.com/syhstudy/Bash2Com

natural language. Therefore, a simple rule-based approach may not apply to Bash code. The method of data augmentation based on gradients is known as adversarial training. This involves training a model with initial examples and generating adversarial examples that can resist attacks and improve the model's robustness. Gradient-based data augmentation methods have been successfully applied to many tasks (such as object detection Li et al. 2017, segmentation Xie et al. 2017, image classification Shrivastava et al. 2017, Text classification Zhu et al. 2020). The fundamental concept of adversarial training can be expressed through the following Max-Min formula (Madry et al. 2018).

$$\min_{\theta} \mathbb{E}_{(x,y)} \sim D\left[\max_{\|\delta\| \leq \epsilon} (L(f_\theta(x+\delta), y))\right] \tag{1}$$

where x represents the input to the model. δ represents the perturbation superimposed on the input. $f_\theta()$ is the neural network function. y is the label of the example. max(L) is the optimization objective to find the perturbation that maximizes the loss function. D is the training set. E is the maximum likelihood estimation.

2.2 CodeBERT

CodeBERT is a pre-trained model based on the transformer architecture (Feng et al. 2020), which operates in dual mode using both natural language (NL) and programming language (PL) encoders. Previous studies have demonstrated that CodeBERT exhibits high performance in software engineering tasks, such as code summarization (Gu et al. 2022; Chen et al. 2022; Yang et al. 2023), API recommendation (Irsan et al. 2023), code search (Shi et al. 2023), and software testing (Fatima et al. 2022). CodeBERT has been pre-trained on a large-scale corpus (Husain et al. 2019) with two self-supervised tasks: Masked Language Model (MLM) (Kenton and Toutanova 2019) and Replaced Token Detection (RTD) (Clark et al. 2019). The MLM task utilizes bi-modal data, feeding the code with corresponding comments and randomly masking positions to replace the token with a special token [MASK]. The objective of the MLM task is to predict the original token, with the loss function defined as follows.

$$L_{MLM}(\theta) = \sum_{i \in \boldsymbol{m}^{\boldsymbol{w}} \cup \boldsymbol{m}^{\boldsymbol{c}}} -\log p^{D_1}(x_i \mid \boldsymbol{w}^{\text{masked}}, \boldsymbol{c}^{\text{masked}}) \tag{2}$$

where p^{D_1} is the predicted token by model. m^w and m^c are the random set of positions for NL and PL to mask as the token [MASK], which means w^{masked} and c^{masked}. x is the input, expressed as $x = w + c$.

The RTD task utilizes uni-modal code and comment to train the data generator to restore the randomly masked token. The loss function of the RTD task is defined as follows.

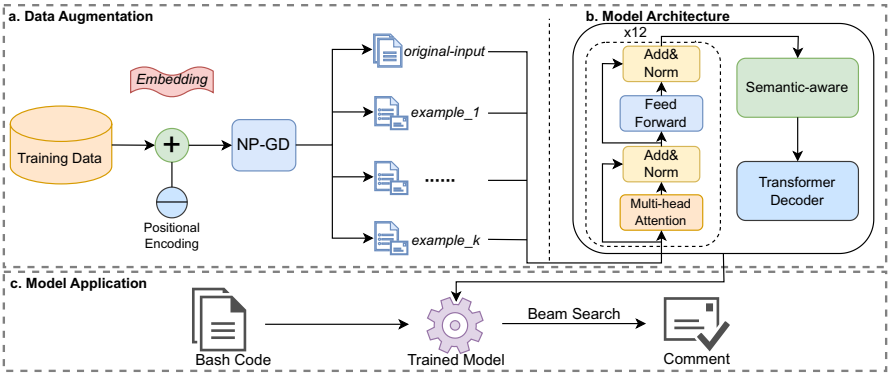

Fig. 2 Overall framework of our method Bash2Com

$$L_{RTD}(\theta) = \sum_{i=1}^{|w|+|c|} \Big[\delta(i)\log p^{D_2}(\boldsymbol{x}^{\text{corrupt}}, i) + (1-\delta(i))(1-\log p^{D_2}(\boldsymbol{x}^{\text{corrupt}}, i))\Big] \tag{3}$$

$$\delta(i) = \begin{cases} 0\ , \text{ if } x_i^{\text{corrupt}} = x_i. \\ 1\ , \text{ otherwise.} \end{cases} \tag{4}$$

where θ is the discriminator parameter, $\delta(i)$ is an indicator function, and p^{D_2} is the discriminator that predicts the probability of the i-th word being original. The RTD randomly replaces a token in the input sequence. A discriminator, trained as a binary classifier, determines whether the replaced token is original or not. This process allows the RTD to learn the importance of each token in the sequence.

3 Our approach

Our proposed method, called Bash2Com, consists of two main parts: *Data Augmentation* and *Model Architecture*, as illustrated in Fig. 2. The data augmentation part is designed to enhance the representation of Bash code for a small corpus, while the model architecture part aims to better represent semantic information by combining the knowledge learned from each layer in CodeBERT.

3.1 Data augmentation

We introduce a novel gradient-based method called NP-GD to automatically generate dependable training samples for data augmentation. By leveraging the strengths of L_1 and L_2 normalizations (Schmidt et al. 2007), NP-GD enhances the stability of the normalization process within the Projected Gradient Descent (PGD) method (Madry et al. 2018).

Word Embedding In the standard encoder-decoder architecture, the word embedding layer captures the relationships between tokens by mapping the text to a vector representation. For the given Bash code X, the input to the model is tokenized by the BPE algorithm (Provilkov et al. 2020) to obtain the sequence $x = (x_1, \cdots, x_N)$, where N is the length of this sequence. To unify the length of the input sequence, we utilize padding or truncation operations. Thus, the output of the word embedding layer can be guaranteed to be uniform $\boldsymbol{x} = (\boldsymbol{x_1}, \cdots, \boldsymbol{x_n})$. At the same time, we add absolute position encoding (APE) (Vaswani et al. 2017) in the embedding layer to better extract the code representation information. Thus the final output is $\boldsymbol{x} = \boldsymbol{x} + APE(\boldsymbol{x})$, $\boldsymbol{x} \in \mathbb{R}^{batch \times n \times d_{model}}$.

Most adversarial training methods in natural language (Goodfellow et al. 2015; Miyato et al. 2017; Madry et al. 2018; Zhu et al. 2020; Jiang et al. 2020) add perturbations to the embedding layer. Compared to adding perturbations directly to the original samples, the subtle perturbations added to the embedding layer have minimal impact on semantics, effectively increasing the number of correct examples in the training data (Morris et al. 2020; Dong et al. 2021), thereby enhancing model performance. Therefore, we adopt the same approach as previous natural language processing methods and generate examples by perturbing the embedding layer.

NP-GD For a given input embedding sequence $\boldsymbol{x}$, NP-GD aims to add perturbation δ and generate K adversarial examples $\left\{\boldsymbol{x_{adv}}^i\right\}_{i=1}^K$. Each $\boldsymbol{x_{adv}}$ is semantically similar to $\boldsymbol{x}$ but different from the source input. The original input $\boldsymbol{x}$ together with the adversarial examples $\left\{\boldsymbol{x_{adv}}^i\right\}_{i=1}^K$ is utilized as the training data to fine-tune CodeBERT.

Our method, NP-GD, is inspired by the concept of PGD (Madry et al. 2018) and employs a multi-iteration attack to compute gradients and perturbations. Like PGD, NP-GD solves the internal maximization problem iteratively, with each iteration projecting the perturbation to a specified range. Specifically, in each iteration of the attack, NP-GD first calculates the gradient g_t of the input $\boldsymbol{x_t}$ at that time, where,

$$g_t = \frac{\partial L(f_\theta(\boldsymbol{x_t}, y))}{\partial \theta}. \tag{5}$$

Then NP-GD computes the perturbation δ_{t+1} as follows,

$$\delta_{t+1} = \alpha(g_t \cdot \|g_t\| / \mid g_t \mid) \tag{6}$$

where α denotes the step size. The normalization method used in PGD is L_∞ normalization. However, the solutions obtained by L_∞ normalization are usually not sparse and do not guarantee to reduce the complexity of the model. To alleviate this issue, our proposed NP-GD takes the L_1 normalization method into account inspired by the study of Simon-Gabriel et al. (2019), as shown in Eq. 6. Specifically, it performs L_1 normalization on the vectors and then applies L_2 normalization to the generated vectors. NP-GD has the advantage of using L_1 normalization to reduce the effect of large values on the vectors and then applying L_2 normalization to ensure that the resulting vectors have a consistent length and sum to 1. Therefore, NP-GD can improve the stability of the normalization process while retaining the advantages of L_1 and L_2 normalization.

Finally, NP-GD can obtain the batch data $\boldsymbol{x_{t+1}}$ by adding a perturbation δ_{t+1} as follows.

$$\boldsymbol{x_{t+1}} = \Pi_{x+\mathcal{S}}\left(\boldsymbol{x_t} + \delta_{t+1}\right) \tag{7}$$

where $\mathcal{S}$ denotes the constraint range of the perturbation δ and its value is limited between -1 and 1. In our study, we utilize $\mathcal{S}$ to measure the magnitude of δ by the L_1 and L_2 normalization (Madry et al. 2018). This can ensure the semantic invariance of examples as much as possible. Moreover, because noise is also a data augmentation method (Moreno-Barea et al. 2018), Bash2Com can accept examples with semantic changes and consider them reasonable errors. NP-GD continues to compute the gradient g_{t+1} on the new batch data input $\boldsymbol{x_{t+1}}$ and repeat the previous steps. After the above steps are iterated K times, NP-GD gets the accumulated gradients by K attacks and then executes gradient descent to update Bash2Com parameters.

3.2 Model architecture

The proposed model adopts a standard encoder-decoder framework. Specifically, Bash2Com first maps <Bash code, code comments> to a high-dimensional semantic space through embedding and utilizes CodeBERT to encode high-dimensional semantic encoding. Then, to extract more semantic information from the various layers of representational data output by CodeBERT, we introduce a new semantic-aware method called MASA, which aids in producing Bash comments. In more detail, MASA utilizes the LSTM to capture different information from CodeBERT's 12 layers sequentially, and subsequently employs an attention mechanism to fuse all the outputs of LSTM. Finally, we utilize the Transformer decoder to generate Bash code comments from the vector representations fused by MASA. The detailed architecture of MASA is illustrated in Fig. 3.

CodeBERT Layer In this layer, we construct an encoder model by using CodeBERT, which generates better semantic representation information for Bash code by fine-tuning CodeBERT. Specifically, for a given input embedding sequence $\boldsymbol{x}$, CodeBERT feeds it into the model to obtain a set of semantic vectors $\left\{\boldsymbol{x_{enc}}^{i}\right\}_{i=1}^{12}$ from all layers, where each $\boldsymbol{X_{enc}} \in \mathbb{R}^{batch \times n \times d_{model}}$.

Semantic-aware layer In this layer, we aim to extract sufficient semantic representations of the target input and combine each layer's semantics according to its weights. Generally speaking, researchers extract the vector of the first token from the encoder in the last layer of CodeBERT as an aggregate sequence representation in previous studies (Jawahar et al. 2019; Kondratyuk and Straka 2019; Yu et al. 2022; Liu et al. 2022). To compute the attention score, we extract the vector of the first token of each layer in CodeBERT as the semantic representation and connect them to a new semantic representation vector $\boldsymbol{X_{Sem}}$ to better use the rich representation information learned by the layers in the pre-trained model. CodeBERT utilizes a 12-layer encoder for pre-training, which is expressed as $\boldsymbol{X_{Sem}} \in \mathbb{R}^{batch \times 12 \times d_{model}}$. Jawahar et al. (2019) found that the shallower the encoder layer of the BERT model, the lower the semantic information represented, while the deeper the encoder layer

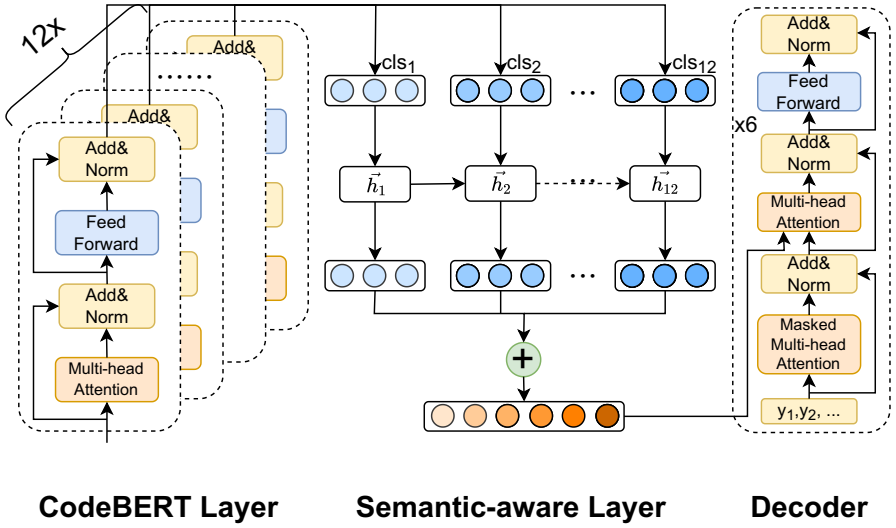

Fig. 3 The model architecture of MASA

of the BERT model, the higher the information represented. Therefore, $\boldsymbol{X_{Sem}}$ can represent the semantic relationship from shallow to deep in CodeBERT.

Given the aforementioned characteristics, attempting to learn all semantic information through LSTM is natural. As shown in Fig. 2, we utilize an LSTM, which contains 12 cells for learning for a given input $\boldsymbol{X_{Sem}} = (\boldsymbol{X_1}, \cdots, \boldsymbol{X_i}, \cdots, \boldsymbol{X_{12}})$. Each semantic information in $\boldsymbol{X_{Sem}}$ is represented by the implicit vector learned by LSTM as follows.

$$\overrightarrow{h_i} = \overrightarrow{\text{LSTM}}(X_i), \ i \in [1, 12] \tag{8}$$

Not all representative information contributes equally to a given target input. Certain source code-related tasks may emphasize low-level representation information (Kondratyuk and Straka 2019), while others may prioritize high-level representation information (Lan et al. 2019). To extract more significant representation information, we employ an attention mechanism. Specifically, we first convert $\boldsymbol{h_i}$ to $\boldsymbol{u_i}$ via the full connection layer $\boldsymbol{u_i} = tanh(Wh_i + b)$. Then, the similarity with the context vector can be calculated and converted into probability distribution by Softmax as follows.

$$\alpha_i = \frac{exp(u_i^T u_w)}{\sum_i exp(u_i^T u_w)} \tag{9}$$

where α_i can be treated as the importance of the output vector for each layer. Therefore, using α_i as a global weighted summation over $\{\boldsymbol{h}_i\}_{i=1}^{12}$ can generate the input vector $\boldsymbol{X_{out}}$ for decoder layer as the fusion of Fig. 3 shows.

$$X_{\text{out}} = \sum_{i}^{12} \alpha_i h_i \tag{10}$$

Decoder layer We utilize the Transformer decoder as the decoder module in our framework. Transformer (Vaswani et al. 2017) leverages an auto-regressive mechanism to predict the next possible word in the generation process based on previous content. Therefore, inferring the subsequent output from the encoder part's output is necessary. Each decoder layer utilizes an attention mechanism for the $\boldsymbol{X_{out}}$ output by MASA. Other operations are consistent with those of the CodeBERT encoder. Finally, the decoder output is passed to a fully connected neural network. The network predicts the probability of the next token through the Softmax layer, which can be defined as follows.

$$P(y_{t+1} \mid y_1, \cdots, y_t) = softmax(W \cdot X_{\text{out}} + b) \tag{11}$$

where y denotes the predicted token. We train our model parameters θ by the loss function L for a given input text x based on cross-entropy, defined as follows.

$$L = -\sum_{i=1}^{|y|} logP_\theta(y_i \mid y < i, x) \tag{12}$$

3.3 Model application

Our trained model can generate relevant comments by analyzing the application developer's Bash code. As the output of Bash2Com is a comment, which can be treated as a token sequence, we employ beam search (Sutskever et al. 2014) to enhance performance.

Beam search returns a list of the most probable output sequences, providing developers with a selection of the most likely comments. This method involves iteratively examining the comment tokens of every step to select the lowest cost k tokens, where k represents the beam width. After pruning any residual branches, the process identifies potential tokens for subsequent steps until reaching the end-of-sequence marker. Ultimately, our model generates and returns k candidate comments for each Bash code. These candidates are then ranked based on their average probabilities throughout the beam search process. Finally, we select the highest-scoring sequence from the final beam as the generated comment. This would be the most likely relevant comment based on the trained model and beam search.

4 Experimental setup

Our empirical study aims to answer the following three research questions.

- RQ1: How successful is our proposed method, Bash2Com, in generating Bash comments?

Table 1 Statistics for length of samples in the corpus

Code length statistics					
Average	Mode	Median	<16	<32	<48
8.528	4	7	90.8%	99.7%	99.9%
Code comment length statistics					
Average	Mode	Median	<16	<32	<48
11.874	10	11	80.3%	99.5%	99.9%

- RQ2: How much does our proposed component, NP-GD, contribute to the effectiveness of our proposed method, Bash2Com, in achieving its objectives?
- RQ3: How much does our proposed component, MASA, contribute to the effectiveness of our proposed method, Bash2Com, in achieving its objectives?

4.1 Experimental subject

Our study considers the corpus shared by Yu et al. (2022) as our experimental subject. First, Yu et al. considered NL2Bash (Lin et al. 2018), which initiated the research of mapping natural language (NL) to Bash command. The corpus consists of 9,305 NL-command pairs. Each pair includes a Bash command scraped from the web and an expert-generated natural language description, covering over 100 commonly used Bash utilities. Previous studies (Kan et al. 2020; Trizna 2021) utilized this corpus as a fundamental dataset, and their findings consistently show its consistency and stability, further strengthening its importance. Then, to comprehensively evaluate their methods, Yu et al. integrated the corpus of NLC2CMD competition [3] to enhance diversity. NLC2CMD consists of nearly 1,700 examples collected from user interactions with the Tellina system (Lin et al. 2017), as well as over 120 examples submitted by 21 participants from both industry and academia through the competition website. Later, we check and delete over 500 duplicate samples. Finally, we create an extended corpus with 10,592 samples, each containing <Bash code, code comments>. The statistical information for this corpus is presented in Table 1. The data reveals that most of the Bash code and its corresponding comments in the corpus consist of approximately 8 or 11 words. Furthermore, it was found that 99.9% Bash code and its corresponding comments in the corpus are no longer than 48 words.

In our empirical study, we used a random sampling method to divide the corpus into three sets: a training set, a validation set, and a test set. The split ratio was 80% for training, 10% for validation, and 10% for testing. This ratio is consistent with the settings used in a previous study (Yu et al. 2022) to ensure a fair comparison.

[3] https://eval.ai/web/challenges/challenge-page/674/leaderboard/1831

4.2 Performance measures

To quantitatively compare the performance between our proposed method and baselines, we consider three performance measures (i.e., BLEU Papineni et al. 2002, METEOR Banerjee and Lavie 2005, and ROUGE-L ROUGE 2004). These performance measures have been extensively utilized in previous studies on neural machine translation and automatic code comment generation (Lin et al. 2023; Wei et al. 2020; Li et al. 2021; Yang et al. 2023; Cao et al. 2021; Zhang et al. 2020; Yang et al. 2021; Liu et al. 2022; Li et al. 2022). The higher the performance measure of these metrics, the better the performance of the corresponding method.

BLEU. BLEU (Bilingual Evaluation Understudy) (Papineni et al. 2002), proposed by IBM in 2002, serves as an evaluation metric for machine translation tasks. It is an accuracy-based similarity measure that assesses the level of simultaneous occurrence of n-grams between candidate texts and reference texts. Commonly used variants of BLEU include BLEU-1, BLEU-2, BLEU-3, and BLEU-4, where “n-gram" represents a sequence of n consecutive words. The calculation formula is:

$$BLUE = BP \cdot exp\left(\sum_{n=1}^{N} w_n log p_n\right) \tag{13}$$

where BP (brevity penalty) is a penalty term for penalizing excessively long translations. p_n is the geometric mean of the ratio of the count of n-gram matches in the translation to the count of n-grams in the reference translation. Setting N = 4 and uniform weights $w_n = 1/N$.

METEOR METEOR (Metric for Evaluation of Translation with Explicit Ordering) (Banerjee and Lavie 2005) addresses some of the shortcomings inherent in the BLEU measure by leveraging knowledge sources, such as WordNet, to expand the synset. The calculation formula is:

$$METEOR = (1 - \alpha) \times \text{precision} + \alpha \times \text{recall} \times \text{FP} \tag{14}$$

where fragmentation penalty (FP) is a penalty term used to penalize fragmented segments in the candidate translation.

ROUGE-L. ROUGE-L (Recall-Oriented Understudy for Gisting Evaluation) (ROUGE 2004) is a recall-based metric that calculates the length of the longest common subsequence (referred to as "L") between the candidate abstract and the reference abstract. A higher score is achieved with a longer common subsequence length. The calculation formula is:

$$ROUGE - L = \frac{\text{Longest Common Subsequence (LCS)}}{\text{Reference Summary Length}} \tag{15}$$

where the Longest Common Subsequence (LCS) represents the length of the longest common subsequence between the candidate summary and the reference summary.

The LCS measures the similarity in content between the two summaries. Reference Summary Length refers to the length of the reference summary.

To ensure consistency in the performance measures and avoid any discrepancies due to differences in their implementation, we utilize the implementation provided by the nlg-eval library[4] for three performance measures, which can mitigate threats to internal validity.

4.3 Baselines

We compare our proposed method Bash2Com's performance with state-of-the-art baselines. Specifically, our experiment contains the only baseline BASHEXPLANER (Yu et al. 2022) in the Bash code comment generation task. In addition, to evaluate Bash2Com more comprehensively, we consider related work for similar tasks (such as source code summarization) as baselines. Finally, we classify these baselines according to their approach characteristics into three categories: information retrieval-based, deep learning-based, and hybrid methods.

The first category includes four information retrieval-based methods. This type of method is an important method in the source code summarization task, which uses the historical code base to mitigate the huge training cost on large-scale data sets. It can achieve competitive performance when a similar code exists in the historical code base.

- **LSI** (Haiduc et al. 2010a) improves keyword-based search by capturing semantic relationships through dimensionality reduction and latent semantic analysis, enhancing retrieval accuracy and handling issues like polysemy and variant forms.
- **VSM** (Haiduc et al. 2010b) represents documents and queries as vectors in a high-dimensional space. It calculates the similarity between vectors to rank documents, commonly using cosine similarity.
- **BM25** (Zhang et al. 2020) considers factors like term frequency, document length, and inverse document frequency to score and rank documents based on relevance to a given query.
- **NNGen** (Liu et al. 2018) generates commit messages based on nearest neighbors, which ranks code based on cosine similarity and BLEU value.

The second category includes seven deep learning-based methods.

- **Transformer** (Ahmad et al. 2020) is an encoder-decoder framework. It utilizes self-attention mechanisms to capture contextual relationships between words in a sequence, enabling effective language understanding and generation, and has achieved state-of-the-art performance in various language-related tasks.
- **CODE-NN** (Iyer et al. 2016) is the first deep learning model that utilizes LSTM and attention mechanism for comment generation tasks.

[4] https://github.com/Maluuba/nlg-eval

- **CodeBERT** (Feng et al. 2020) is built using a transformer-based neural architecture and trained with a hybrid objective function. It leverages the power of pre-training on large-scale data and fine-tuning on specific tasks, enabling it to understand and generate code with improved accuracy and efficiency.
- **UniXcoder** (Guo et al. 2022) is a unified cross-modal pre-trained model for programming languages, which uses a masked attention matrix to control the behavior of the model and enhances the code representation with cross-modal content such as AST and code comments.
- **CoTexT** (Phan et al. 2021) is a pre-trained transformer-based encoder-decoder model that uses self-supervision to learn representative contexts between natural language (NL) and programming language (PL).
- **PLBART** (Ahmad et al. 2021) is a sequence-to-sequence model that learns program syntax, style, and logical flow crucial to program semantics. It enhances code-related tasks, benefiting software development and programming language processing applications.
- **CodeT5** (Wang et al. 2021) is a unified pre-trained encoder-decoder model that supports multi-task learning and can better use the information of code tokens, allowing CodeT5 to understand and generate code based on the given context and task requirements.

The third category includes three hybrid methods of considering multiple inputs.

- **Hybrid-Deepcom** (Hu et al. 2020) considers the semantic and structural information of Java code and obtains the syntactic information of the code by traversing AST.
- **Rencos** (Zhang et al. 2020) first retrieves similar code and then fuses the vectors by the decoder.
- **BASHEXPLAINER** (Yu et al. 2022) is the only model in Bash comment generation. It utilizes two-stage training strategies: the fine-tuning stage and the information retrieval enhancement stage.

For the pre-trained-based baselines (i.e., UniXcoder,[5] CoTexT,[6] PLBART,[7] and CodeT5[8]), we implement them with Hugging Face according to the paper description, and our implementation results are close to the results reported in their studies. For the remaining baselines, we directly utilize the scripts shared by original studies. To ensure a fair comparison between Bash2Com and baselines, we consider the original parameter settings (Table 2).

[5] https://huggingface.co/microsoft/unixcoder-base
[6] https://huggingface.co/razent/cotext-1-cc
[7] https://huggingface.co/uclanlp/plbart-base
[8] https://huggingface.co/Salesforce/codet5-base

Table 2 Comparison results between Bash2Com and baselines in terms of BLEU- 1/2/3/4, METEOR and ROUGE-L measures

Method type	Method name	BLEU-1 (%)	BLEU-2 (%)	BLEU-3 (%)	BLEU-4 (%)	METEOR (%)	ROUGE-L (%)
Information retrieval	LSI	30.18	18.07	12.48	9.40	18.30	28.82
	VSM	36.16	24.47	18.62	15.25	22.04	34.58
	BM25	42.08	30.41	23.58	19.24	26.35	38.49
	NNGen	50.62	38.75	32.11	27.85	27.69	45.88
Deep learning	Transformer	46.39	33.37	25.42	19.97	25.22	44.01
	CODE-NN	49.60	37.18	29.53	24.17	26.85	47.21
	CodeBERT	48.65	37.02	29.84	24.83	27.16	47.36
	UniXcoder	49.99	38.52	31.80	27.25	29.03	48.24
	CoTexT	49.17	37.29	30.36	25.75	28.56	48.00
	PLBART	50.79	39.10	32.21	27.55	28.82	47.91
	CodeT5	51.75	40.04	33.25	28.70	29.49	48.36
Hybrid method	BASHEXPLAINER	51.74	40.41	33.73	29.13	28.78	48.81
	Hybrid-DeepCom	47.78	35.45	27.91	22.75	26.27	45.36
	Rencos	46.27	35.11	28.66	24.39	25.82	45.06
Our Method	**Bash2Com**	**54.74**	**43.79**	**37.19**	**32.57**	**30.26**	**51.80**

Table 3 Hyper-parameters setting of Bash2Com

Category	Hyper-parameter	Value
NP-GD	K	3
	α	0.3
CodeBERT	Decoder_layers	6
	Hidden_size	768
	Max_input_length	48
	Max_output_length	64
	Beam_search_size	10

4.4 Experimental settings

Our proposed method and baselines are implemented using the PyTorch framework. The code for BASHEXPLANER is implemented by the code provided by Yu et al. (2022). Specifically, we utilize the Transformers[9] and CodeBERT[10] to implement our proposed method. The model is trained using the AdamW optimizer for 50 epochs, with a learning rate of 2e-4. The weight α is 0.3, and the number of times K is set to 3 in NP-GD. To reduce the time cost of the model, we have implemented the technique of early stopping with a step size of 5 in this experiment. Table 3 illustrated the specific hyper-parameter setting of Bash2Com.

We run all the experiments on a computer with an Intel(R) Xeon(R) Silver 4210 CPU and a GeForce RTX3090 GPU with 24 GB memory. The running OS platform is Windows OS.

5 Experimental results

5.1 RQ1: How successful is our proposed method, Bash2Com, in generating Bash comments?

Automatic Evaluation RQ1 aims to compare the Bash2Com with fourteen state-of-the-art baselines, illustrated in Sect. 4.3. These baselines can be classified into three types: information retrieval methods, deep learning methods, and hybrid methods. Table 2 shows the overall results of the different methods concerning three evaluation measures (BLEU, ROUGE-L, and METEOR), and we mark the best one of each metric in bold. As seen in Table 2, our proposed method Bash2Com outperforms all considered baselines.

Firstly, we compare Bash2Com with information retrieval baselines. The state-of-the-art method is NNGen, which utilizes cosine similarity and BLEU values to retrieve similar code, so the NNGen method has better BLEU values. However, our method Bash2Com can perform better than NNGen. Specifically, Bash2Com can

[9] https://github.com/huggingface/transformers

[10] https://huggingface.co/microsoft/codebert-base

Table 4 Results of p-values and δ

Measure	BLEU-4	METEOR	ROUGE-L
p-values	0.0315	0.0028	0.0164
δ	0.682	0.795	0.811

improve performance by at least 8.14%, 13.01%, 15.82%, and 16.95% for BLEU-1/2/3/4 measures. Our method can also improve performance by 9.28%, and 12.90% for METEOR and ROUGE-L measures.

Secondly, we compare Bash2Com with deep learning baselines. The results show that the CodeT5 method performs best among the deep learning methods. However, for the BLEU-1/2/3/4, METEOR and ROUGE-L measures, Bash2Com can improve the performance by at least 5.78%, 9.37%, 11.85%, 13.48%, 2.61% and 7.11%. One possible reason is insufficient training data, which is challenging for deep learning models that require large amounts of training data. In contrast, Bash2Com using adversarial training is more suitable for handling small corpus, which can help to improve the model performance.

Thirdly, we compare Bash2Com with the hybrid baselines. We consider BASHEXPLAINER the baseline because it is the best-performing method among the hybrid methods and the state-of-the-art baseline for Bash code comment generation. Compared to BASHEXPLAINER, Bash2Com improves the performance by at least 5.80%, 8.36%, 10.19%, 11.81%, 5.14% and 6.13% in terms of BLEU-1/2/3/4, METEOR and ROUGE-L measures.

Finally, to further analyze whether there is a significant difference between the comparison results of Bash2Com and baselines, we consider the Wilcoxon signed-rank test (Rey and Neuhäuser 2011). Since BLEU-4 is the most meaningful in the evaluation measures of BLEU, we only perform a significance test for BLEU-4. In addition, we only consider the best-performing baseline BASHEXPLAINER. The p-values in Table 4 can reject the null hypothesis, which means significant differences exist between our method and baselines in all the measures considered. Then, we use Cliff's delta (δ) (Rey and Neuhäuser 2011), a non-parametric effect size measure, to quantify the magnitude of such differences between Bash2Com and the best-performing BASHEXPLAINER. According to the suggestions (Rey and Neuhäuser 2011), we classify difference magnitude into four levels according to the values of δ: "<0.147": Negligible (N), "0.147~0.33": Small (S), "0.33~0.474": Medium (M), and ">0.474": Large (L). As shown in Table 4, all δ values are larger than 0.474, which shows a large performance improvement of Bash2Com.

Human Study. Although automatic evaluation measures can calculate the textual dissimilarity between automatically generated comments and human-written comments, they cannot accurately distinguish the semantic similarity between them Iyer et al. (2016), Wei et al. (2020), Zhang et al. (2020), Hu et al. (2021). Therefore, we introduced the following three metrics for our human study:

Table 5 Comparison results of our human study (standard deviation in parentheses)

Method	Informativeness	Naturalness	Similarity
NNGen	1.81(1.22)	**3.63**(0.99)	1.49(1.03)
CodeBERT	2.18(1.14)	3.21(1.10)	1.71(1.29)
BASHEXPLAINER	2.76(1.34)	3.51(0.89)	2.38(1.45)
Bash2Com	**2.83**(1.09)	3.57(0.75)	**2.51**(1.17)

- Informativeness: Informativeness focuses only on the amount of information in the automatically generated comments without fluency.
- Similarity: Similarity is the degree of similarity between automatically generated and human comments.
- Naturalness: Naturalness considers the grammatical and fluency of automatically generated and human comments.

The score ranges from 0 to 4; the higher the score, the better the comment can meet the requirements. The final score is averaged.

For the human study, we selected NNGen, CodeT5, BASHEXPLAINER, and our proposed method Bash2Com as they are state-of-the-art methods with the best performance in their respective categories. We recruited ten volunteers: two PhDs and eight Masters - who have extensive experience in Linux development and maintenance to evaluate the automatically generated comments. Among them, the master-level volunteers have 1–3 years of experience, and the doctor-level volunteers have over 3 years of experience, which is beneficial for guaranteeing our human study quality. We randomly selected 100 code pairs from the corpus, including Bash code and comments, and provided manual comments for reference. The volunteers rated the comments generated by the four methods based on similarity, naturalness, and informativeness. To ensure fairness, we ensured that each volunteer did not know how the comments they received were generated. Additionally, to ensure the quality of the labels, we limited the number of Bash codes each volunteer marked to 25 per day.

Table 5 presents the human study results between Bash2Com and three representative baselines. The results show that our method Bash2Com improved over three baselines on Informativeness and Similarity but is slightly lower than NNGen by 0.06 points on the Naturalness metric. One possible reason is that NNGen is based on an information retrieval method, and the generated Bash code comments are obtained from the original corpus. Since professional developers write these comments, they are more natural. On the other hand, the comments generated by Bash-2Com are only 0.06 points below in the Naturalness metric, which indicates that Bash2Com is also effective in naturalness.

To assess the differences in volunteer scoring results, we consider Fleiss Kappa (Fleiss 1971) to measure the evaluation consistency of these volunteers. The result was 0.748, representing the consistency of the scoring results of the volunteers. Furthermore, we calculate p-values to examine whether the performance difference is statistically significant between Bash2Com and BASHEXPLAINER (the

Table 6 Comments generated by different methods for three Bash codes

ID	Example
1	**Bash Code:** cat $(whoami)
	Ground Truth: print current user's mail file
	NNGen: print current user name
	CodeT5: print file name of current user
	BASHEXPLAINER: print user name of current user
	Bash2Com: print file content of current user
2	**Bash Code:** sleep $(expr 'date -d 03/21/2014 12:30 +%s' - 'date +%s')
	Ground Truth: sleep until a specif date
	NNGen: print day between _regex and _regex
	CodeT5: sleep date _timespan ago
	BASHEXPLAINER: print _regex day in _timespan
	Bash2Com: sleep until _datetime
3	**Bash Code:** echo $(/usr/sbin/arp $(hostname) awk -F[()] { print $ 2 })
	Ground Truth: print ip address of current host
	NNGen: print current host name
	CodeT5: print ip address of host name
	BASHEXPLAINER: print _regex follow by current hostname
	Bash2Com: print ip address of current host

best-performing baseline) by Wilcoxon signed-rank test in terms of readability, comprehensibility, and naturalness. In our study, the hypothesis is set as follows, H0: There is no significant difference between Bash2Com and BASHEXPLAINER in terms of readability (comprehensibility, or naturalness). We set the significance level at 0.05, and all the *p*-values of all the considered metrics are lower than 0.05. These results show a significant difference between Bash2Com and BASHEXPLAINER in terms of readability, comprehensibility, and naturalness.

Case study To reveal the complementary between Bash2Com and baselines, we perform a case study by analyzing Bash code comments generated by Bash2Com and representative baselines in each category of baselines. The results are shown in Table 6. Although NNGen scored high in the automatic evaluation in these cases, the generated code comments differed significantly from the ground truth. One possible reason for this is the Bash code syntax flexibility. As the best-performing baseline BASHEXPLAINER, the comments generated in the second and third examples were poorly readable and understandable. A possible reason for this is that it considers information from the retrieved corpus, and this interference affects the model's performance. In addition, the comments generated by CodeT5 are somewhat distant from the ground truth. These examples show that Bash2Com can make up for the shortcomings of baselines.

Summary for RQ1: Bash2Com improves the performance by at least 11.81%, 5.14% and 6.13% over the best-performing baseline BASHEXPLAINER in terms of BLEU-4, METEOR and ROUGE-L. Moreover, human study shows

Table 7 Ablation study results for adversarial training methods in Bash2Com

Setting	BLEU-3	BLEU-4	METEOR	ROUGE-L
w/o NP-GD	32.83	28.28	27.95	49.18
with FGSM	28.70	25.22	26.12	47.99
with FGM	33.74	29.14	27.82	49.20
with PGD	34.42	29.91	28.65	49.57
Bash2Com	**37.19**	**32.57**	**30.26**	**51.80**

that Bash2Com outperforms three baselines in terms of informativeness, naturalness, and similarity.

5.2 RQ2: How much does our proposed component, NP-GD, contribute to the effectiveness of our proposed method, Bash2Com, in achieving its objectives?

RQ2 aims to investigate the impact of three different adversarial training methods on the Bash code comment generation task. In particular, we consider the popular adversarial training methods (i.e., FGSM Goodfellow et al. 2015, FGM Miyato et al. 2017 and PGD Madry et al. 2018). FGSM aims to make the disturbance direction δ by following the gradient direction and takes the max normalization of the gradient by the *sign* function. Unlike FGSM, which takes the same steps in each direction, FGM scales according to specific gradients to obtain better adversarial examples. FGM uses L_2 normalization, which divides the value of each gradient dimension by the L_2 parameter of the gradient.

Table 7 shows the comparison results of using different adversarial training methods. The experimental results show that utilizing NP-GD can significantly improve the model performance. Specifically, compared to Bash2Com without NP-GD, Bash2Com can improve the performance by 13.28%, 15.17%, 8.26% and 5.33% for BLEU-3, BLEU-4, METEOR, and ROUGE-L respectively. This result indicates that our data augmentation component NP-GD is significantly valuable for the Bash code comment generation task. At the same time, the experimental results also show that NP-GD can achieve the best performance among the different adversarial training methods. NP-GD can improve the performance by 8.05%, 8.89%, 5.62%, and 4.50% for BLEU-3, BLEU-4, METEOR, and ROUGE-L respectively, compared to the best-performing baseline method PGD. This result shows that our proposed NP-GD can fully use the advantages of L_1 normalization and L_2 normalization, which eventually improves the stability of the normalization process and makes it more suitable for the Bash code comment generation task.

Summary for RQ2: For the Bash code comment generation task, NP-GD can contribute more than traditional adversarial training methods.

Table 8 Ablation study results for Semantic-aware methods of Bash2Com

Setting	BLEU-3	BLEU-4	METEOR	ROUGE-L
w/o MASA	33.55	28.98	28.31	48.86
with GRU	34.44	29.83	28.75	49.34
with Bi-LSTM	34.83	30.24	28.82	49.81
Bash2Com	**37.19**	**32.57**	**30.26**	**51.80**

5.3 RQ3: How much does our proposed component, MASA, contribute to the effectiveness of our proposed method, Bash2Com, in achieving its objectives?

RQ3 aims to conduct an ablation study to demonstrate the effectiveness of semantic-aware component MASA utilized by our proposed method Bash2Com. Specifically, we set up four control methods:

(1) without MASA: To verify the necessity of a semantic-aware component, we directly utilize the output of the last layer of CodeBERT as the learned feature information into the decoder to generate Bash code comments;
(2) with GRU: To verify the ability of different RNN models to aggregate feature information in this method, we utilize GRU-Attention (Cho et al. 2014) to fuse the feature information output from each layer of CodeBERT into the decoder to generate Bash code comments;
(3) with Bi-LSTM: To verify the ability of different RNN models to aggregate feature information in this method, we utilize Bi-LSTM-Attention (Graves and Schmidhuber 2005) to fuse the feature information output from each layer of CodeBERT into the decoder to generate Bash code comments;
(4) with MASA: This is the component applied by Bash2Com, which utilizes LSTM + Attention to fuse the feature information output from each layer of CodeBERT into the decoder to generate Bash code comments.

Table 8 shows the results of the ablation experiments. After comparing these control methods, Bash2Com with MASA can achieve the best performance. Specifically, compared to no MASA, Bash2Com with MASA can improve the performance by 10.85%, 12.39%, 6.89% and 6.02% for BLEU-3, BLEU-4, METEOR, and ROUGE-L respectively. This result shows that LSTM can effectively fuse representative information from CodeBERT output, which helps to improve the quality of the generated Bash code comments. In addition, compared with other variants of RNNs, Bash2Com with MASA can improve the performance by at least 6.78%, 7.71%, 5.00%, and 4.00% for BLEU-3, BLEU-4, METEOR, and ROUGE-L respectively. This result indicates that MASA can fuse the representative information from CodeBERT output more effectively than other variants of RNNs in this ablation experiment, which can ultimately improve the quality of the generated Bash code comments.

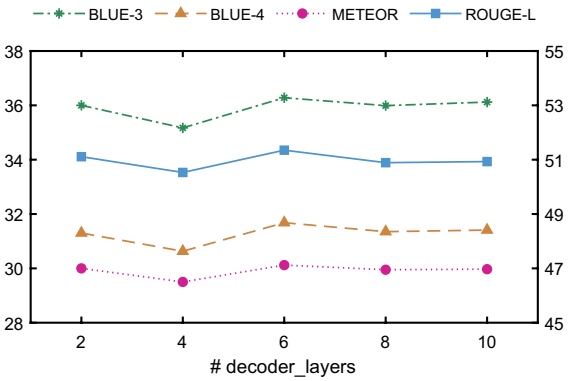

Fig. 4 Sensitivity Analysis on the number of decoder layers (the left axis is used for BLEU and METEOR, and the right axis is used for ROUGE-L)

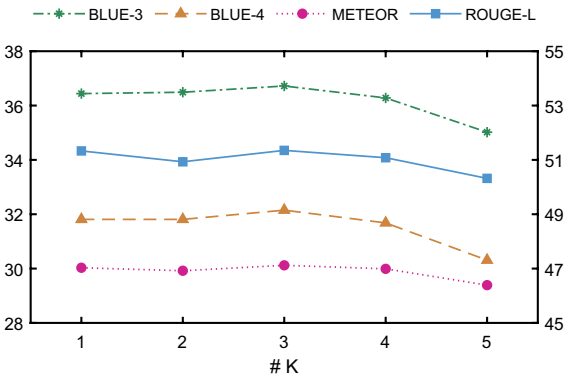

Fig. 5 Sensitivity Analysis on the number of adversarial examples (the left axis is used for BLEU and METEOR, and the right axis is used for ROUGE-L)

Summary for RQ3: The utilization of MASA by Bash2Com is shown to enhance model performance, with the LSTM model selected by MASA demonstrating superior performance compared to other RNN variants.

6 Discussion

6.1 Analysis on the hyper-parameters setting

In this subsection, we perform a sensitivity analysis on the parameters of Bash2Com. We mainly focus on two parameters (i.e., the number of decoder layers *decoder_layers* and the number of adversarial examples K). The results of the sensitivity analysis are shown in Fig. 4 and Fig. 5, where all hyperparameters except the hyperparameter of the current analysis are set to the optimal setting. Since the numerical differences in evaluation measures, the left axis is used for BLEU and

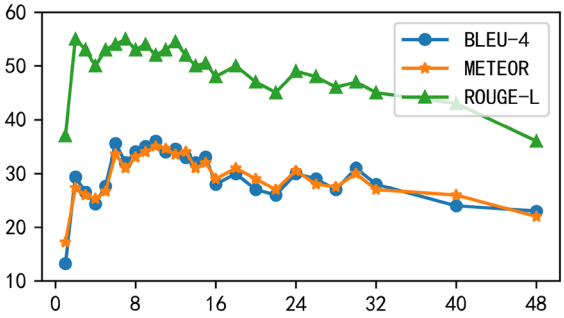

Fig. 6 Analysis on the impact of different code lengths

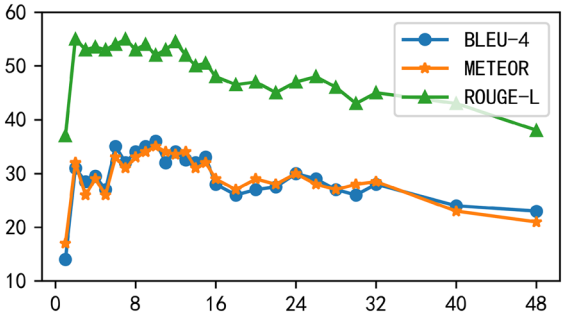

Fig. 7 Analysis on the impact of different comments lengths

METEOR, and the right axis is used for ROUGE-L. The optimal settings are set as follows: *decoder_layers* is 6, and *K* is 3.

In Fig. 4, we find that the best performance is achieved for all four evaluation metrics when the number of layers of the decoder is 6. When the number of layers of the decoder is increased from 2 to 6, the scores of the four evaluation measures show a decreasing trend followed by an increasing trend. When the number of layers of the decoder is larger than 6, the scores of the four evaluation measures show a flat decreasing trend. When the number of generated adversarial examples changes from 1 to 5, the four evaluation measures shown in Fig. 5 show an overall trend of rising first and then declining, and we set the value of this parameter to 3 since it can achieve the best performance. In our experiments, to balance the performance and cost of the model, we set the number of decoder layers to 6 and the number of generated adversarial examples to 3.

6.2 Impact analysis of code lengths and comment lengths

In this subsection, we analyze the performance of Bash2Com in different code lengths and comment lengths with BLEU-4, METEOR, and ROUGE-L. The results of the influence results can be found in Fig. 6 and Fig. 7. Based on the experimental results, we find that the performance of Bash2Com is consistent with the statistical distribution of the corpus.

Table 9 Comparison results of ChatGPT and Bash2Com

Method	BLEU	METEOR	ROUGE-L	CIDEr
ChatGPT(zero-shot)	10.92	6.91	10.91	10.02
ChatGPT(one-shot)	18.51	10.64	17.49	17.75
Bash2Com	**42.07**	**30.26**	**51.80**	**26.14**

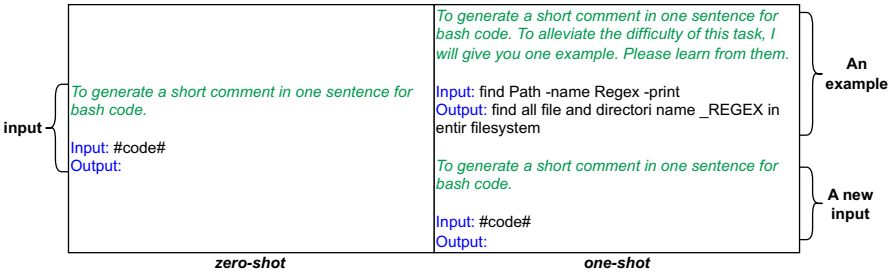

Fig. 8 Prompts of zero-shot and one-shot

According to the corpus statistics in Table 1, we know that the length of most Bash code and the corresponding comments are mainly around 8 or 11 words in the corpus. In addition, 90.8% (80.3%) of Bash code (comments) is less than 16 words in length. Based on these training sets, Bash2Com is stable in length from 1 to 16. Moreover, because there are data with a length of more than 32, the performance of the model declines. This result indicates that our data augmentation technology is valuable for the Bash code comment generation task. Therefore, collecting more corpus according to our length analysis in the future can further strengthen our Bash code comment generation method Bash2Com.

6.3 Performance analysis of ChatGPT

Large language models (LLMs) have been used for many mainstream software engineering tasks, which refer to a class of artificial intelligence models that use an enormous amount of parameters and are designed to process and generate human-like text based on large-scale language datasets (Zhao et al. 2023). In our study, we also designed related experiments to verify the performance of LLMs in the Bash code comments generation task. The model used in our experiment is GPT3.5, a widely used version of ChatGPT.[11]

Specifically, we conduct two sets of experiments as shown in Fig. 8. The left half is based on zero-shot learning, which involves generating Bash comments without any prompt information. The right half is based on one-shot learning, employing the prompt learning method (Gu et al. 2022), which provides a group of prompt information. This

[11] https://chat.openai.com/

method can be utilized as an expert question-answer form to leverage ChatGPT's capabilities in related fields more effectively. Prompts cover the basic input–output structure, and italicized sentences are instructions for ChatGPT, indicating the objectives and related restrictions of ChatGPT. Then, the prompt ends with a new requirement and is fed into ChatGPT. We hope ChatGPT to learn from examples and generate a comment for a new requirement. Additionally, we limit the comment length. This setting can avoid generating excessively long comments, which may achieve low scores in automatic evaluation.

Table 9 shows the comparison results of Bash2Com and ChatGPT. The BLEU is the average value of BLEU-1/2/3/4, and we added CIDEr (Vedantam et al. 2015) to evaluate the model performance further. The results show that our method Bash2Com improved over ChatGPT. Specifically, in the automatic index evaluation, ChatGPT(one-shot) performs better than the native ChatGPT(zero-shot), but both are behind Bash2Com. This result indicates that ChatGPT needs advanced professional knowledge to play its performance, but Bash code, as a small language, does not have too many corpora to train the GPT model. ChatGPT can be optimized through prompt learning technology, but it may need more data support, and we will also pay attention to it in future research.

6.4 Limitations of Bash2Com

Our automatic evaluation shows that our proposed method Bash2Com outperformed the baselines. However, we also notice that Bash2Com may generate comments of lower quality than the ground truth. Therefore, we utilize BLEU-4 as the filtering metric to filter out the data by threshold (BLEU-4<0.01). After using this filtering strategy, we collect 190 pieces of data and these filtered data is available on our homepage.[12] After analyzing all these data manually, we identify three main challenges for our proposed approach.

The first challenge is that when technicians write comments, synonyms are diversified. For example, "search system for file _file" and "find all files under _file and sort them" contain some synonyms, but the mixed words cause Bash2Com prediction error. One possible solution is to standardize comment writing and avoid mixing synonymous words.

The second challenge arises when some common fields of Bash code are overwritten. For example, some Bash code snippets find or remove files under a certain path. The original Bash code is "find *ask* : *ask/www/askapache*" or "rm *ask* : *ask/www/askapache*", but it became "find Path" or "rm File" after preprocessing. This operation is helpful for Bash2Com to learn more important semantic information in Bash code. However, it will also make some code inputs too short, making Bash2Com difficult to learn semantics.

The third challenge is that the current model's performance can still be improved. Therefore, further augmenting the training data and improving the diversity of the contained Bash code and comments may alleviate this issue.

[12] https://github.com/syhstudy/Bash2Com/blob/master/limitation_data.csv

Finally, we analyze the complementarity from the limitation to reveal further perils and pitfalls of Bash2Com and baselines. For example, the Bash code is “find Path -name Regex" and the ground truth is “find all text file in current folder". The generated comment by Bash2Com is “find all text file of current folder", BASHEXPLAINER is “find all text file in folder", and NNGen is “find file in current folder". Based on this case, we observe that the comment generated by Bash2Com demonstrates good informativeness and similarity. However, there is a decrease in naturalness due to the substitution of “in" with “of". On the other hand, BASHEXPLAINER and NNGen generate comments with slightly lower informativeness (missing determiners). However, they exhibit naturalness and fluency.

6.5 Threats to validity

Internal threats The first internal threat is the potential faults in implementing Bash-2Com. To mitigate this threat, we utilize mature libraries(such as PyTorch and transformers) to implement the methods. The second internal threat is the randomness of adversarial training to generate adversarial examples. To mitigate this threat, we guarantee the replicability of our method by fixing random seeds. The third internal threat is the baselines considered in RQ1. For these baselines, we use the scripts shared by Yu et al.[13] Then, we reproduce other baseline models according to original studies and achieve similar performance.

External threats The main external threat to this study is the corpus. To mitigate this problem, we performed data augmentation using adversarial training. Also, we follow the previous study’s settings (Yu et al. 2022) in the division of the corpus to ensure a fair comparison.

Construct threats Construct threats mainly refer to the selection of automatic assessment measures. To mitigate these threats, we chose three performance measures: BLEU (Papineni et al. 2002), METEOR (Banerjee and Lavie 2005), and ROUGE-L (ROUGE 2004). These evaluation measures are widely utilized in the field of neural machine translation and automatic comment generation (Lin et al. 2023; Wei et al. 2020; Li et al. 2021; Yang et al. 2023; Cao et al. 2021; Zhang et al. 2020; Yang et al. 2021; Liu et al. 2022; Li et al. 2022). We also conduct a human study and compute the p-value by using the Wilcoxon signed-rank test to evaluate the readability, comprehensibility, and naturalness of Bash comments generated by our proposed method and representative baselines.

Conclusion threats We only split the corpus once because of the high computational cost of deep learning. This setting is consistent with the previous study on Bash code comment generation (Yu et al. 2022). To mitigate the conclusion threat, we also randomly split the corpus three times by different random seeds, and we show detailed comparison results on the project homepage,[14] which also confirms the effectiveness of our proposed method when compared with baselines. The second conclusion threat is related to the experience of volunteers. To alleviate this threat, we recruited ten volunteers. For master-level volunteers, they have 1–3 years of experience in Linux development and maintenance. For doctor-level volunteers,

[13] https://github.com/NTDXYG/BASHEXPLAINER

[14] https://github.com/syhstudy/Bash2Com/blob/master/README_add.md

they have over 3 years of experience. However, we acknowledge there still may exist some misjudgments caused by the relatively limited experience of these volunteers and this is a potential threat to our study's conclusion validity.

7 Related work

7.1 Code comment generation

Code comment generation (Alon et al. 2018; Yang et al. 2022; Wu et al. 2021) is a task that generates comments for a piece of the source code. We classify these methods into the following three categories: Information retrieval-based, Deep learning-based, and hybrid methods.

Information retrieval-based methods were first investigated, often achieving better high-reusable code performance. Haiduc et al. (2010b) first proposed VSM (Vector Space Model) and LSI (Latent Semantic Index) to retrieve relevant terms from a corpus to generate comments on classes and methods. Then Eddy et al. (2013) extended it to a thematic model hPAM to build code comments. Recently, Yang et al. (2022) proposed the method CCGIR based on semantic similarity, lexical similarity, and syntactic similarity of smart contract code.

Regarding deep learning-based methods, researchers discovered that leveraging code structure information could enhance the quality of model generation. LeClair et al. (2020) proposed treating the words in the code and the serialized AST structure as separate data sources. Furthermore, they also introduced a graph-based neural architecture method (LeClair et al. 2019) that better matches the default structure of the AST to generate these comments. Meanwhile, Wu et al. (2021) proposed a structure-guided Transformer, which incorporates multi-perspective structural cues to guide the encoding process. Then, Wei et al. (2019) and Ye et al. (2020) used the dual learning framework to explore the relationship between Code summarization and Code generation to improve the model performance. Haque et al. (2020) modeled other subroutines within the same file and employed attention mechanisms to identify the words and concepts used in the summary. Recently, Mastropaolo et al. (2022) discovered that pre-trained models can achieve success in code summarization tasks.

Some recent studies have proposed a hybrid method that combines multiple inputs. Wei et al. (2020) proposed the Re^{2}Com method, which retrieves similar code snippets and then reuses existing comments as examples to guide comment generation. Zhang et al. (2020) proposed the Rencos method, which considers syntactic and semantic similarity. Rencos retrieves the two most similar code fragments from the corpus, generates fusion vectors, and then decodes the fusion vectors to generate comments.

However, most of the existing comment-generation methods focus on popular programming languages. In this study, we focus on a specific programming language Bash, and propose a novel Bash comment generation method Bash-2Com. Aiming at the challenges faced by Bash code comment generation, such as insufficient corpus, we propose a data enhancement component named NP-GD.

Moreover, we further propose a semantic awareness component, MASA, to learn the semantic representation of Bash code better. Both automatic evaluation and human study verify the effectiveness of our study.

7.2 Data augmentation in NLP

In natural language processing, typical data augmentation methods can be categorized as rule-based and gradient-based. For the former, Wei and Zou (2019) proposed the EDA method, which includes some easy data augmentation operations. Xie et al. (2020) proposed the UDA algorithm related to noncore word replacement technology, that is, to replace a certain proportion of nonimportant words in the text with unimportant words in the dictionary to generate new text. Anaby-Tavor et al. (2020) proposed a GPT-based data augmentation technology, LAMBDA, which first pre-trained and then fine-tunes the model on a small number of corpus for different tasks.

For the gradient-based data augmentation technology (e.g., adversarial training), Miyato et al. (2017) introduced adversarial training and virtual adversarial training (Miyato et al. 2018) into the NLP domain to improve the performance of classification models. Ebrahimi et al. (2018) proposed using character/word substitution (white-box method) to generate adversarial examples. Zhu et al. (2020) utilized a gradient-based method to add adversarial perturbations to the embedding of input sentences, allowing the model to improve performance in the field of natural language understanding. Recently, Zhang et al. (2020) also found that model performance can significantly improve in code comment generation by generating adversarial examples.

To the best of our knowledge, no research has focused on the performance of adversarial training in bash code comment generation. We consider adversarial training in our method Bash2Com to fill this gap. Moreover, to better adapt to our task, we designed a customized adversarial training method NP-GD, which uses L_1 normalization to reduce the effect of large values on the vectors and then applies L_2 normalization to ensure that the resulting vectors have a consistent length and sum to 1.

7.3 Bash code comment generation

Lin et al. (2018) were the first to study the field of Bash code and constructed a corpus of Bash code using natural language descriptions. Yu et al. (2022) further enriched the corpus based on the study of Lin et al. (2017). They combined the NL2Bash study with the corpus shared in the NLC2CMD competition to construct a corpus with 10,592 Bash codes and corresponding comments. They also propose an automatic generation model BASHEXPLAINER for Bash code comments based on a two-stage training strategy.

However, Yu et al. (2022) only used information retrieval methods to enhance input representation. Motivated by the Bash code comment generation task, we

introduce the idea of data augmentation and multi-layer semantic awareness, making the model fine-tuned on a larger corpus and learning the multi-level semantic representation.

8 Conclusion

In this study, we propose a novel Bash comment generation method Bash2Com, which includes two components NP-GD and MASA. Specifically, NP-GD is designed to enhance the data representation. In NP-GD, the data is augmented by acting on the adversarial training of the embedding layer. MASA is designed to represent better information learned by each layer in CodeBERT. In MASA, the representation information of each layer of CodeBERT is fully utilized using LSTM, and weights are assigned to each layer using an attention mechanism. Finally, the Transformer decoder utilizes the output vector to generate the corresponding Bash comment. We evaluate the effectiveness of the Bash2Com through extensive experiments, which show that our proposed method outperforms state-of-the-art baselines.

In the future, we want to evaluate the effectiveness of our proposed method by extending it to more low-resource source code understanding tasks. Moreover, we also want to further improve our proposed method's performance by considering more advanced code representation and adversarial training methods.

References

Ahmad, W., Chakraborty, S., Ray, B., Chang, K.-W.: A transformer-based approach for source code summarization. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pp. 4998–5007 (2020)

Ahmad, W., Chakraborty, S., Ray, B., Chang, K.-W.: Unified pre-training for program understanding and generation. In: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 2655–2668 (2021)

Alon, U., Brody, S., Levy, O., Yahav, E.: code2seq: Generating sequences from structured representations of code. In: International Conference on Learning Representations (2018)

Anaby-Tavor, A., Carmeli, B., Goldbraich, E., Kantor, A., Kour, G., Shlomov, S., Tepper, N., Zwerdling, N.: Do not have enough data? deep learning to the rescue! In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, pp. 7383–7390 (2020)

Banerjee, S., Lavie, A.: Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In: Proceedings of the Acl Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation And/or Summarization, pp. 65–72 (2005)

Cao, K., Chen, C., Baltes, S., Treude, C., Chen, X.: Automated query reformulation for efficient search based on query logs from stack overflow. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering, pp. 1273–1285 (2021)

Chen, F., Fard, F.H., Lo, D., Bryksin, T.: On the transferability of pre-trained language models for low-resource programming languages. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, pp. 401–412 (2022)

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y.: Learning phrase representations using rnn encoder–decoder for statistical machine translation.

In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, p. 1724 (2014)
Clark, K., Luong, M.-T., Le, Q.V., Manning, C.D.: Electra: Pre-training text encoders as discriminators rather than generators. In: International Conference on Learning Representations (2019)
Dong, X.L., Zhu, Y., Fu, Z., Xu, D., de Melo, G.: Data augmentation with adversarial training for cross-lingual nli. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, pp. 5158–5167 (2021)
Ebrahimi, J., Rao, A., Lowd, D., Dou, D.: Hotflip: White-box adversarial examples for text classification. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, pp. 31–36 (2018)
Eddy, B.P., Robinson, J.A., Kraft, N.A., Carver, J.C.: Evaluating source code summarization techniques: Replication and expansion. In: 2013 21st International Conference on Program Comprehension, pp. 13–22 (2013)
Fatima, S., Ghaleb, T.A., Briand, L.: Flakify: a black-box, language model-based predictor for flaky tests. IEEE Trans. Softw. Eng. **49**(4), 1912–1927 (2022)
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. ACL Anthology, 1536–1547 (2020)
Fleiss, J.L.: Measuring nominal scale agreement among many raters. Psychol. Bull. **76**(5), 378 (1971)
Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. In: 3th International Conference on Learning Representations (2015)
Graves, A., Schmidhuber, J.: Framewise phoneme classification with bidirectional LSTM and other neural network architectures. Neural Netw. **18**(5–6), 602–610 (2005)
Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., Yin, J.: Unixcoder: Unified cross-modal pre-training for code representation. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, pp. 7212–7225 (2022)
Gu, J., Salza, P., Gall, H.C.: Assemble foundation models for automatic code summarization. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering, pp. 935–946 (2022)
Gu, Y., Han, X., Liu, Z., Huang, M.: Ppt: Pre-trained prompt tuning for few-shot learning. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, pp. 8410–8423 (2022)
Haiduc, S., Aponte, J., Marcus, A.: Supporting program comprehension with source code summarization. In: 2010 Acm/ieee 32nd International Conference on Software Engineering, vol. 2, pp. 223–226 (2010)
Haiduc, S., Aponte, J., Moreno, L., Marcus, A.: On the use of automated text summarization techniques for summarizing source code. In: 2010 17th Working Conference on Reverse Engineering, pp. 35–44 (2010)
Haque, S., LeClair, A., Wu, L., McMillan, C.: Improved automatic summarization of subroutines via attention to file context. In: Proceedings of the 17th International Conference on Mining Software Repositories, pp. 300–310 (2020)
He, H.: Understanding source code comments at large-scale. In: ESEC/FSE 2019: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1217–1219. Association for Computing Machinery, New York, NY, USA (2019)
Hochreiter, S., Schmidhuber, J.: Lstm can solve hard long time lag problems. Advances in neural information processing systems **9** (1996)
Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., Brockschmidt, M.: Codesearchnet challenge: Evaluating the state of semantic code search. arXiv preprint arXiv:1909.09436 (2019)
Hu, X., Li, G., Xia, X., Lo, D., Jin, Z.: Deep code comment generation. In: 2018 IEEE/ACM 26th International Conference on Program Comprehension, pp. 200–20010 (2018)
Hu, X., Li, G., Xia, X., Lo, D., Jin, Z.: Deep code comment generation with hybrid lexical and syntactical information. Empir. Softw. Eng. **25**(3), 2179–2217 (2020)
Hu, X., Gao, Z., Xia, X., Lo, D., Yang, X.: Automating user notice generation for smart contract functions. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering, pp. 5–17 (2021)

Irsan, I.C., Zhang, T., Thung, F., Kim, K., Lo, D.: Picaso: Enhancing api recommendations with relevant stack overflow posts. arXiv preprint arXiv:2303.12299 (2023)
Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics: Long Papers, pp. 2073–2083 (2016)
Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, pp. 2073–2083 (2016)
Jawahar, G., Sagot, B., Seddah, D.: What does bert learn about the structure of language? In: 57th Annual Meeting of the Association for Computational Linguistics (2019)
Jiang, H., He, P., Chen, W., Liu, X., Gao, J., Zhao, T.: Smart: Robust and efficient fine-tuning for pre-trained natural language models through principled regularized optimization. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pp. 2177–2190 (2020)
Kan, J.-W., Chien, W.-C., Wang, S.-D.: Grid structure attention for natural language interface to bash commands. In: 2020 International Computer Symposium, pp. 67–72 (2020)
Kenton, J.D.M.-W.C., Toutanova, L.K.: Bert: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of NAACL-HLT, pp. 4171–4186 (2019)
Kondratyuk, D., Straka, M.: 75 languages, 1 model: Parsing universal dependencies universally. In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, pp. 2779–2795 (2019)
Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., Soricut, R.: Albert: A lite bert for self-supervised learning of language representations. In: International Conference on Learning Representations (2019)
LeClair, A., Jiang, S., McMillan, C.: A neural model for generating natural language summaries of program subroutines. In: 2019 IEEE/ACM 41st International Conference on Software Engineering, pp. 795–806 (2019)
LeClair, A., Haque, S., Wu, L., McMillan, C.: Improved code summarization via a graph neural network. In: Proceedings of the 28th International Conference on Program Comprehension, pp. 184–195 (2020)
Lin, X.V., Wang, C., Pang, D., Vu, K., Ernst, M.D.: Program synthesis from natural language using recurrent neural networks. University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-17-03-01 (2017)
Lin, X.V., Wang, C., Pang, D., Vu, K., Zettlemoyer, L., Ernst, M.D.: Program synthesis from natural language using recurrent neural networks. In: Technical Report UW-CSE-17-03-01, University of Washington Department of Computer Science and Engineering (2017)
Lin, X.V., Wang, C., Zettlemoyer, L., Ernst, M.D.: Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. In: Proceedings of the Eleventh International Conference on Language Resources and Evaluation (2018)
Lin, H., Chen, X., Chen, X., Cui, Z., Miao, Y., Zhou, S., Wang, J., Su, Z.: Gen-fl: quality prediction-based filter for automated issue title generation. J. Syst. Softw. **195**, 111513 (2023)
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)
Liu, K., Yang, G., Chen, X., Zhou, Y.: El-codebert: Better exploiting codebert to support source code-related classification tasks. In: Proceedings of the 13th Asia-Pacific Symposium on Internetware, pp. 147–155 (2022)
Liu, K., Yang, G., Chen, X., Yu, C.: SOTitle: A Transformer-based Post Title Generation Approach for Stack Overflow, pp. 577–588 (2022)
Li, J., Liang, X., Wei, Y., Xu, T., Feng, J., Yan, S.: Perceptual generative adversarial networks for small object detection. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1222–1230 (2017)
Li, Z., Wu, Y., Peng, B., Chen, X., Sun, Z., Liu, Y., Paul, D.: Setransformer: A transformer-based code semantic parser for code comment generation. In: IEEE Transactions on Reliability (2022)
Li, Z., Wu, Y., Peng, B., Chen, X., Sun, Z., Liu, Y., Yu, D.: Secnn: a semantic CNN parser for code comment generation. J. Syst. Softw. **181**, 111036 (2021)

Madry, A., Makelov, A., Schmidt, L., Tsipras, D., Vladu, A.: Towards deep learning models resistant to adversarial attacks. In: 6th International Conference on Learning Representations (2018)
Mastropaolo, A., Cooper, N., Palacio, D.N., Scalabrino, S., Poshyvanyk, D., Oliveto, R., Bavota, G.: Using transfer learning for code-related tasks. IEEE Trans. Softw. Eng. **49**(4), 1580–1598 (2022)
Miyato, T., Dai, A.M., Goodfellow, I.: Adversarial training methods for semi-supervised text classification. In: 5th International Conference on Learning Representations (2017)
Miyato, T., Maeda, S.-I., Koyama, M., Ishii, S.: Virtual adversarial training: a regularization method for supervised and semi-supervised learning. IEEE Trans. Pattern Anal. Mach. Intell. **41**(8), 1979–1993 (2018)
Moreno-Barea, F.J., Strazzera, F., Jerez, J.M., Urda, D., Franco, L.: Forward noise adjustment scheme for data augmentation. In: 2018 IEEE Symposium Series on Computational Intelligence, pp. 728–734 (2018)
Morris, J., Lifland, E., Yoo, J.Y., Grigsby, J., Jin, D., Qi, Y.: Textattack: A framework for adversarial attacks, data augmentation, and adversarial training in nlp. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pp. 119–126 (2020)
Newham, C.: Learning the bash shell: uniX shell programming. O'Reilly Media Inc (2005)
Papineni, K., Roukos, S., Ward, T., Zhu, W.-J.: 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)
Phan, L., Tran, H., Le, D., Nguyen, H., Annibal, J., Peltekian, A., Ye, Y.: Cotext: Multi-task learning with code-text transformer. In: Proceedings of the 1st Workshop on Natural Language Processing for Programming, pp. 40–47 (2021)
Provilkov, I., Emelianenko, D., Voita, E.: Bpe-dropout: Simple and effective subword regularization. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pp. 1882–1892 (2020)
Rey, D., Neuhäuser, M.: Wilcoxon-signed-rank test. In: International Encyclopedia of Statistical Science, pp. 1658–1659 (2011)
ROUGE, L.C.: A package for automatic evaluation of summaries. In: Proceedings of Workshop on Text Summarization of ACL (2004)
Schmidt, M., Fung, G., Rosales, R.: Fast optimization methods for l1 regularization: A comparative study and two new approaches. In: Machine Learning: ECML 2007: 18th European Conference on Machine Learning, Warsaw, Poland, September 17-21, 2007. Proceedings 18, pp. 286–297 (2007)
Shi, E., Wang, Y., Gu, W., Du, L., Zhang, H., Han, S., Zhang, D., Sun, H.: Cocosoda: Effective contrastive learning for code search. In: 2023 IEEE/ACM 45th International Conference on Software Engineering, pp. 2198–2210 (2023)
Shrivastava, A., Pfister, T., Tuzel, O., Susskind, J., Wang, W., Webb, R.: Learning from simulated and unsupervised images through adversarial training. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2107–2116 (2017)
Simon-Gabriel, C.-J., Ollivier, Y., Bottou, L., Schölkopf, B., Lopez-Paz, D.: First-order adversarial vulnerability of neural networks and input dimension. In: International Conference on Machine Learning, pp. 5809–5817 (2019)
Su, T.-C., Cheng, H.-C.: Sesamebert: Attention for anywhere. In: 2020 IEEE 7th International Conference on Data Science and Advanced Analytics, pp. 363–369 (2020)
Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. Advances in neural information processing systems **27** (2014)
Trizna, D.: Shell language processing: Unix command parsing for machine learning. arXiv preprint arXiv:2107.02438 (2021)
Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł, Polosukhin, I.: Attention is all you need. Adva. Neural Inf. Process. Syst. **30**, 5998–6008 (2017)
Vedantam, R., Lawrence Zitnick, C., Parikh, D.: Cider: Consensus-based image description evaluation. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4566–4575 (2015)
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)

Wei, J., Zou, K.: Eda: Easy data augmentation techniques for boosting performance on text classification tasks. In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, pp. 6382–6388 (2019)
Wei, B., Li, G., Xia, X., Fu, Z., Jin, Z.: Code generation as a dual task of code summarization. Advances in neural information processing systems **32** (2019)
Wei, B., Li, Y., Li, G., Xia, X., Jin, Z.: Retrieve and refine: exemplar-based neural comment generation. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering, pp. 349–360 (2020)
Wiseman, S., Rush, A.M.: Sequence-to-sequence learning as beam-search optimization. In: Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, pp. 1296–1306 (2016)
Wu, H., Zhao, H., Zhang, M.: Code summarization with structure-induced transformer. In: Findings of the Association for Computational Linguistics, pp. 1078–1090 (2021)
Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A.E., Li, S.: Measuring program comprehension: a large-scale field study with professionals. IEEE Trans. Softw. Eng. **44**(10), 951–976 (2017)
Xie, Q., Dai, Z., Hovy, E., Luong, T., Le, Q.: Unsupervised data augmentation for consistency training. Adv. Neural Inf. Process. Syst. **33**, 6256–6268 (2020)
Xie, C., Wang, J., Zhang, Z., Zhou, Y., Xie, L., Yuille, A.: Adversarial examples for semantic segmentation and object detection. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 1369–1378 (2017)
Yang, G., Chen, X., Cao, J., Xu, S., Cui, Z., Yu, C., Liu, K.: Comformer: Code comment generation via transformer and fusion method-based hybrid code representation. In: 2021 8th International Conference on Dependable Systems and Their Applications, pp. 30–41 (2021)
Yang, G., Zhou, Y., Chen, X., Yu, C.: Fine-grained Pseudo-code Generation Method via Code Feature Extraction and Transformer, pp. 213–222 (2021)
Yang, G., Chen, X., Zhou, Y., Yu, C.: Dualsc: Automatic generation and summarization of shellcode via transformer and dual learning. In: IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022, pp. 361–372 (2022)
Yang, G., Liu, K., Chen, X., Zhou, Y., Yu, C., Lin, H.: Ccgir: information retrieval-based code comment generation method for smart contracts. Knowledge-Based Syst. **237**, 107858 (2022)
Yang, G., Zhou, Y., Chen, X., Zhang, X., Han, T., Chen, T.: Exploitgen: template-augmented exploit code generation based on codebert. J. Syst. Softw. **197**, 111577 (2023)
Ye, W., Xie, R., Zhang, J., Hu, T., Wang, X., Zhang, S.: Leveraging code generation to improve code retrieval and summarization via dual learning. In: Proceedings of The Web Conference 2020, pp. 2309–2319 (2020)
Yu, C., Yang, G., Chen, X., Liu, K., Zhou, Y.: Bashexplainer: Retrieval-augmented bash code comment generation based on fine-tuned codebert. In: 2022 IEEE International Conference on Software Maintenance and Evolution, pp. 82–93 (2022)
Zhang, J., Wang, X., Zhang, H., Sun, H., Liu, X.: Retrieval-based neural source code summarization. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering, pp. 1385–1397 (2020)
Zhang, X., Zhou, Y., Han, T., Chen, T.: Training deep code comment generation models via data augmentation. In: 12th Asia-Pacific Symposium on Internetware, pp. 185–188 (2020)
Zhao, W.X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., et al.: A survey of large language models. arXiv preprint arXiv:2303.18223 (2023)
Zhu, C., Cheng, Y., Gan, Z., Sun, S., Goldstein, T., Liu, J.: Freelb: Enhanced adversarial training for natural language understanding. In: 8th International Conference on Learning Representations (2020)

Authors and Affiliations

Yiheng Shen[1] · Xiaolin Ju[1] · Xiang Chen[1] · Guang Yang[2]

✉ Xiaolin Ju
ju.xl@ntu.edu.cn

✉ Xiang Chen
xchencs@ntu.edu.cn

Yiheng Shen
yiheng.s@outlook.com

Guang Yang
novelyg@outlook.com

[1] School of Information Science and Technology, Nantong University, Nantong 226019, Jiangsu, China

[2] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, Jiangsu, China