Automatic Comment Generation via Multi-Pass Deliberation

Fangwen Mu†‡
fangwen2020@iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

Xiao Chen†‡
chenxiaochen2021@iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

Lin Shi†‡§
shilin@iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

Song Wang
wangsong@eecs.yorku.ca
Lassonde School of Engineering, York University
Toronto, Canada

Qing Wang†‡§$
wq@iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

ABSTRACT
Deliberation is a common and natural behavior in human daily life. For example, when writing papers or articles, we usually first write drafts, and then iteratively polish them until satisfied. In light of such a human cognitive process, we propose DECOM, which is a multi-pass deliberation framework for automatic comment generation. DECOM consists of multiple Deliberation Models and one Evaluation Model. Given a code snippet, we first extract keywords from the code and retrieve a similar code fragment from a pre-defined corpus. Then, we treat the comment of the retrieved code as the initial draft and input it with the code and keywords into DECOM to start the iterative deliberation process. At each deliberation, the deliberation model polishes the draft and generates a new comment. The evaluation model measures the quality of the newly generated comment to determine whether to end the iterative process or not. When the iterative process is terminated, the best-generated comment will be selected as the target comment. Our approach is evaluated on two real-world datasets in Java (87K) and Python (108K), and experiment results show that our approach outperforms the state-of-the-art baselines. A human evaluation study also confirms the comments generated by DECOM tend to be more readable, informative, and useful.

CCS CONCEPTS
- Software and its engineering → Software notations and tools; - Computing methodologies → Artificial intelligence.

KEYWORDS
Comment generation, Information retrieval, Deep neural network

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ASE ’22, October 10–14, 2022, Rochester, MI, USA
© 2022 Copyright held by the owner/author(s).
https://doi.org/10.1145/3551349.3556917

1 INTRODUCTION
With software growing in size and complexity, developers tend to averagely spend around 59% of their effort on program comprehension during software development and maintenance [32, 62]. Source code comments provide concise natural language descriptions of code snippets, which not only greatly reduce the effort for developers to understand the code, but also play a vital role in software maintenance and evolution. However, manually commenting code is time-consuming, and code comments are often missing or outdated in software projects [13, 30]. Therefore, the code comment generation task, which aims at automatically generating a high-quality comment for a given code snippet, has long attracted the interest of many researchers.

Most of the existing approaches treat comment generation as a machine translation task and adopt a one-pass encoder-decoder process, i.e., first encode the input code into a sequence of semantic features, then decode the features to a natural language comment [26, 28, 34, 64]. Although the encoder-decoder framework has achieved remarkable performance on the comment generation task, it still suffers from two major limitations. The first one is that such models adopt a regular one-pass decoding process that sequentially generates comments word by word. They directly use the generated comment as the final output, which results in their inability to correct the mispredicted words. The words mistakenly predicted in the early steps may lead to error accumulation under the constraint of the language model. Taking the auto-generated comments in Figure 1 as an example, the one-pass model Re³com [58] incorrectly predicts the sixth word “probability” as “bytes”, which leads the model to keep exploring the words related to “bytes” when predicting the consequential words. As a result, the related words “written to this stream” are mistakenly generated, which results in a typical example of error accumulation. The second limitation is that they generate comments sequentially. Thus, such sequential one-pass models cannot leverage the global information of the generated comment to further polish its local content. As the example shown
public CategoricalTable copy(){
    Map<Value,Double> newTable=new HashMap<Value,Double>();
    if(variable==null){
        variable = 1;
    }
    if(table.isEmpty()){  
        return new CategoricalTable{variable};
    }
    for(Value v : table.keySet()){  
        newTable.put(v,table.get(v));
    }
         return new CategoricalTable{variable,newTable};
}

Ground Truth: returns a copy of the probability table
Initial Draft: constructs a new multivariate table from a univariate table

One-Pass Decoding:
Rencos: prunes all table values that have a probability lower than the threshold
Re2com: returns a copy of the bytes written to this stream
EditSum: creates a new copy of the probability table

Multi-Pass Deliberation:
First-Pass: creates a new copy of the given distribution
Second-Pass: returns a copy of the table from this table
Third-Pass: returns a copy of the probability table

Closeness to copy
Closeness to copy
Closeness to copy

Figure 1: A motivation example of multi-pass deliberation.

In Figure 1, the one-pass model EditSum [34] generates two consecutive prepositional phrases after the word “copy”. Although either of them is reasonable in their local contexts (“of the given table” and “of the table”), putting them together degrades the comment fluency and makes the developers hard to understand.

To alleviate these challenges, we introduce the Deliberation mechanism [63] in the comment generation task, aiming to further enhance the performance. Deliberation is a common and natural behavior in human daily life. When writing papers or articles, we usually first write drafts, and then iteratively polish them until satisfied. Figure 1 illustrates an example of applying multi-pass deliberation on comment generation. Based on the initial draft “constructs a new multivariate table from a univariate table”, the first-pass deliberation will generate the comment “creates a new copy of the given distribution”, refine it in the second and the third pass guided by the closeness to the similarity with the give code snippet. In the end, we could obtain the most satisfying comment “returns a copy of the probability table”.

In light of such a human cognitive process, we propose a novel multi-pass deliberation framework for automatic comment generation, named DECOM, which contains multiple Deliberation Models and one Evaluation Model. Given a code snippet, initially, we retrieve the most similar code from a pre-defined corpus and treat its comment as the initial draft. We also extract the identifier names from the input code as keywords, since these user-defined words usually contain more semantic information that users want to express [10, 48, 53]. Then, we input the code, the keywords, and the initial draft into DECOM to start the iterative deliberation process. At each deliberation, the deliberation model polishes the draft and generates a new comment. The evaluation model calculates the quality score of the newly generated comment. This multi-pass process terminates when (1) the quality score of the new comments is no longer higher than the previous ones, or (2) the maximum number of deliberations is reached. To evaluate our approach, we conduct experiments on two real-world datasets in Java (87K) and Python (108K), and the results show that our approach outperforms the state-of-the-art baselines by 8.3%, 6.0%, 13.3%, and 10.5% with respect to BLEU-4, ROUGE-L, METEOR, and CIDEr on Java dataset. On Python dataset, DECOM improves the performance on BLEU-4, ROUGE-L, METEOR, and CIDEr by 5.8%, 3.8%, 6.6%, and 6.3%, respectively. We also conduct a human evaluation to assess the generated comments on three aspects: naturalness, informativeness, and usefulness, showing that DECOM can generate useful and relevant comments.

Our main contributions are outlined as follows:

- **Technique**: a multi-pass deliberation framework for comment generation, named DECOM, which is inspired by the human cognitive process, and can effectively generate comments in an iterative way. To the best of our knowledge, this is the first work that employs multi-pass deliberation to enhance the performance of comment generation.
- **Evaluation**: an experimental evaluation of the performance of DECOM against state-of-the-art baselines, which shows that DECOM outperforms all baselines, together with a human evaluation, which further confirms the readability, informativeness, and usefulness of DECOM.
- **Data**: publicly accessible dataset and source code [4] to facilitate the replication of our study and its application in extensive contexts.

In the rest of this paper, Section 2 elaborates the approach. Section 3 presents the experimental setup. Section 4 demonstrates the results and analysis. Section 5 describes the human evaluation. Section 6 discusses indications and threats to validity. Section 7 introduces the related work. Section 8 concludes our work.

## 2 APPROACH

In this section, we present our DECOM, a multi-pass deliberation framework that performs an iterative polishing process to refine the draft to a better comment. Figure 2 illustrates an overview of DECOM, which consists of three main stages: (1) **Data initialization**, for extracting the keywords from the input code and retrieving the similar code-comment pair from the retrieval corpus; (2) **Model training**, for leveraging a two-step training strategy to optimize DECOM; and (3) **Model prediction**, for generating the target comment of the new source code. Below, we provide details for each stage in DECOM.

### 2.1 Data Initialization

Given a code snippet $x$, this stage aims to extract the keywords $t$ from $x$, and retrieve the initial draft $z^0$ from the retrieval corpus.

**Extract keywords from code.** A code snippet contains many different types of tokens, such as reserve words (if, for), identifier names (set_value, SortList), and operators (+, *). Among them, identifier names defined by users usually contain more semantic information that users want to express [10, 48, 53]. For example, a method’s name is a typical identifier name, which is used to describe the overall functionality of the code and can be considered as a shorter version of its code comment. Thus, to enable the model to attend more on the identifier names and capture semantic information from them, we extract these words from code. First, we use the javalang [3] and tokenize [5] libraries to extract identifier names from the Java and Python code snippets, respectively. Then,
we further split the extracted identifier names into sub-tokens by CamelCase or snake_case to obtain the smaller semantic units and reduce data sparsity. These sub-tokens are treated as the keywords of the code.

Retrieve the initial draft. To obtain the initial draft, following the previous studies [58], we use the lexical similarity-based retrieval method to identify the top similar code-comment pair for the given code \( x \). Specifically, we first take the training set of the benchmark dataset as the retrieval corpus. Then, for each code in the retrieval corpus, we adopt the BM25 [45] metric to calculate the similarity between it and the given code \( x \). The BM25 is a bag-of-words retrieval metric to measure the relevance of documents to a given search query in IR and is also widely used in code clone detection and code search tasks [29, 39, 46]. Finally, we extract the code with the highest similarity score as the retrieved result, and use the comment of the code as the initial draft \( z^0 \). Since the size of our training sets is quite large, we leverage the open-source search engine Lucene [1] to speed up the retrieval process. We follow the settings of Lucene from Re²Com [58] to run our experiments.

2.2 Model Training

DECOM contains \( K \) deliberation models and one evaluation model, where \( K \) is the maximum deliberation number. To reduce computation cost and facilitate the sharing of information between models, all \( K \) deliberation models share three encoders with others and share the code encoder and comment encoder with the evaluation model. Each deliberation model has its own decoder, which can avoid these models generating highly similar comments. We employ a two-step training strategy to train DECOM as shown in Figure 2. In the first step, we locally train the \( K \) deliberation models: we first jointly train the first deliberation model and the evaluation model. Then we freeze the shared encoders and train the other deliberation models one by one. In the second step, we fine-tune DECOM by jointly optimizing all trained models.

2.2.1 Deliberation Model. Each deliberation model consists of three different encoders (i.e. code encoder, keyword encoder, and comment encoder) and a decoder. The details of them are illustrated in the following.

Figure 2: The overall architecture of DECOM

Figure 3: The detailed structure of the Deliberation model.
Encoders. The code encoder, keyword encoder, and comment encoder aim to encode the source code $x$, keywords $t$, and the previous comment $z^{k-1}$ as vectors, thus enabling the deliberation model to obtain the semantic information from both source-side (code and keywords) and target-side (past comment). We construct the three encoders by following the structure of the vanilla Transformer Encoder [54]. As shown in Figure 3, each encoder is composed of a stack of $N$ identical Transformer Encoder blocks. Each block contains two sub-layers: The first sub-layer is a multi-head self-attention layer (MHAtt), which employs multiple attention heads to capture the information from different representation sub-spaces at different positions. The second sub-layer is a two-layer Feed-Forward Network (FFN) with a ReLU activation function in between. The residual connection is employed around the two sublayers, followed by layer normalization (LayerNorm) [7]. Since the three encoders have the same structure, we only introduce the code encoder for simplicity.

Given a code snippet $x = [x_1, x_2, ..., x_{|x|}]$, where $|x|$ is the number of words in the code. The code encoder first embeds each word of the code into a d dimensional word vector:

$$\vec{x} = W_e^T \cdot x + PE_i$$

where $W_e$ is a trainable embedding matrix, and $PE_i$ is the position encoding of the $i$-th word. Following previous study [54], we use the $\sin$ and $\cos$ function of different frequencies to compute the position encoding:

$$PE_{i,2j} = \sin(\frac{j}{10000^{2j/d}})$$
$$PE_{i,2j+1} = \cos(\frac{j}{10000^{2j/d}})$$

where $i$ is the position of the word and $j$ denotes the $j$-th dimension of the embedding vector.

Then, the code encoder inputs the sequence of word embeddings into $N$ identical encoder blocks to calculate the hidden states of the code. For the $i$-th block of the code encoder, suppose that the input is $H_{i-1}$, the output $H_i$ is calculated as follows:

$$H_{i,1} = \text{LayerNorm}\left(H_{i-1} + \text{MHAtt}(H_{i-1}, H_{i-1}, H_{i-1})\right)$$
$$H_i = \text{LayerNorm}\left(H_{i,1} + \text{FFN}(H_{i,1})\right)$$

where $H_{i,1}$ is the hidden states of the first sub-block. Initially, the word embedding vectors $[\vec{x}_1, \vec{x}_2, ..., \vec{x}_{|x|}]$ are fed into the first block, and the $N^{th}$ block outputs the final hidden states of the input code $H = [h_1, h_2, ..., h_{|x|}]$. Similarly, DECOM can encode the keywords $t$ and the past comment $z^{k-1}$ into hidden states $P$ and $R^{k-1}$, respectively.

There are two points worth noting: (1) In the first-pass deliberation, DECOM takes the comment of the retrieved code as the initial draft $z^0$, for each turn after this, DECOM uses the comment generated in the previous turn as the draft. (2) source code $x$ and keywords $t$ do not change in the iterative deliberation process, so to save computational resources and time, we compute their hidden states $H$ and $P$ only once, and reuse them in subsequent iterations.

Decoder. The decoder aims to improve the quality of the previously generated comment $z^{k-1}$ by jointly leveraging its context and the semantics of the source code $x$ and the keywords $t$. As shown in Figure 3, the decoder is also composed of a stack of $N$ identical decoder blocks, and each block consists of four sub-layers. In addition to the first and the last sub-layers introduced in the part of Encoders in section 2.2.1, the decoder block inserts two multi-head cross attention sub-layers in between, which are used to capture the information from the outputs of the three encoders.

In the $k$-th pass deliberation ($k \geq 1$), given the hidden states $H$, $P$, $R^{k-1}$. The $i$-th block of the decoder first gets the hidden states of the first sub-layer $S_{i,1}$ using Eq. (4). Then, in the second sub-layer, the block separately performs multi-head attention over the hidden states of the source code $H$ and the keywords $P$:

$$a_i = \text{MHAtt}(S_{i,1}, H, H)$$
$$b_i = \text{MHAtt}(S_{i,1}, P, P)$$

Besides, to effectively leverage the information from source-side, we utilize the gate mechanism [23] to adaptively incorporate the $a_i$ containing source code features and the $b_i$ containing keywords features:

$$\beta = \text{Sigmoid}(W_{gate}(a_i ; b_i))$$
$$S_{i,2} = \text{LayerNorm}\left(S_{i,1} + \beta \cdot a_i + (1 - \beta) \cdot b_i\right)$$

where $\beta$ is the degree of integration between source code and keywords. A larger value of $\beta$ (ranges from 0 to 1) indicates that the model should pay more attention to the information in the source code. $W_{gate}$ is a trainable parameter matrix, $[\cdot, \cdot]$ is concatenation operation, and $S_{i,2}$ is the hidden states of the second sub-layer. In the third sub-layer, the block obtains the $S_{i,3}$ by performing multi-head attention over the hidden states of the previous comment $R^{k-1}$:

$$S_{i,3} = \text{LayerNorm}\left(S_{i,2} + \text{MHAtt}(S_{i,2}, R^{k-1}, R^{k-1})\right)$$

Based on this equation, the decoder can capture the important clues from the global information of the past comment for further refinement. Then, according to Eq. (5), the $i$-th block uses the $S_{i,3}$ to compute the output of the last sub-layer $S_i$. After the calculation of $N$ decoder blocks, the decoder gets the hidden states of the last decoder block $S$. For the $j$-th decoding step, the probability of the $j^{th}$ token $z^j$ can be calculated by projecting the $j^{th}$ state $s_j$ in $S$ via a linear layer followed by a Softmax function:

$$p(z^j | z^1, z^2, ..., z^{j-1}) = \text{Softmax}(W_o^T \cdot s_j + b_o)$$

where $W_o$ is the parameter matrix and $b_o$ is the bias. Ultimately, we use the Argmax function to generate the new comment $z^k$:

$$z^k = \text{Argmax}\{p(z^k_1); p(z^k_2); \cdots; p(z^k_{l(k)})\}$$

where the $l(k)$ is the length of the $k$-th generated comment.

2.2.2 Evaluation Model. The evaluation model aims to estimate the quality of the generated comments and calculate their quality scores. As shown in Figure 4, the evaluation model contains a shared code encoder, a shared comment encoder, and an evaluator.

Given the new comment $z^k$ generated by the $k$-th deliberation model, the comment encoder encodes the $z^k$ into hidden states $R^k$. To obtain the representation of the comment, the evaluator first uses Mean Pooling to average the hidden states $R^k = [r^k_0, r^k_1, ..., r^k_{l(k)}]$ to get the aggregated features $i^k_{mean}$. Then, it utilizes a two-layer
from scratch is unstable in practice, which is mainly because of the cold start [49] problem. To mitigate this problem, we use a two-step training strategy as shown in Figure 2.

Step 1: Locally train the K Deliberation models. We first jointly train the first deliberation model and the evaluation model by minimizing the following loss function:

$$L(\theta^1, \theta_e) = L_{delib}(\theta^1) + \alpha_e L_{eval}(\theta_e)$$

(18)

where the $\alpha_e$ is a hyperparameter, which is set to be 0.1 in our experiments to control the weight of the evaluation loss. Then we freeze the three shared encoders, and iteratively train the subsequent deliberation models using the Eq. (16) until the last deliberation model is trained.

Step 2: Globally train the K Deliberation models. One of the drawbacks of the first-step training is that the deliberation models is optimized independently and the model components cannot share the information. To address this, we further fine-tune DECOM by jointly training all K deliberation models and the evaluation model:

$$L(\theta^1, ..., \theta^K, \theta_e) = L_{delib}(\theta^1) + ... + L_{delib}(\theta^K) +$$

$$\alpha_e L_{eval}(\theta_e)$$

(19)

Note that in this step, all parameters are unfrozen and are updated at the same time.

2.3 Model Prediction

The prediction stage aims to generate a concise and useful comment for a given code snippet. As shown in Figure 2, given a new code snippet $x$, we first perform data initialization introduced earlier to obtain the keywords $t$ and the initial draft $z^0$. Then, we input them into DECOM to generate the target comment automatically. The comment generation process involves multiple deliberation processes. During the $k^{th}$ deliberation, the $k^{th}$ deliberation model polishes the previously generated comment $z^{k-1}$ and generates a new comment $z^k$. The evaluation model estimates the quality of the new comment $z^k$ by calculating the cosine similarity between this and the source code $x$. The deliberation process is performed iteratively unless either of the following two conditions is satisfied: (1) the quality score of the new comment is no longer higher than the previous ones; (2) a certain number of deliberations $K > 0$ is reached. In the former case, we adopt the previous comment as the target comment. In the latter case, the last generated comment is accepted.

3 EXPERIMENTAL SETUP

3.1 Dataset

Since most of the related studies [6, 12, 16, 64, 65] for comment generation tasks are evaluated on JCDSD [27] and PCSD [9] benchmark datasets, in this study, we also select these two datasets in our experiments. JCDSD has 87,136 code-comment pairs collected from more than 9K Java Github repositories created from 2015 to 2016 with at least 20 stars. It first extracted Java methods and Javadocs, and treated the first sentence of the Javadoc as the ground-truth comment. Since most of the related studies [6, 12, 16, 64, 65] for comment generation tasks are evaluated on JCDSD [27] and PCSD [9] benchmark datasets, in this study, we also select these two datasets in our experiments. JCDSD has 87,136 code-comment pairs collected from more than 9K Java Github repositories created from 2015 to 2016 with at least 20 stars. It first extracted Java methods and Javadocs, and treated the first sentence of the Javadoc as the ground-truth comment. Since most of the related studies [6, 12, 16, 64, 65] for comment generation tasks are evaluated on JCDSD [27] and PCSD [9] benchmark datasets, in this study, we also select these two datasets in our experiments. JCDSD has 87,136 code-comment pairs collected from more than 9K Java Github repositories created from 2015 to 2016 with at least 20 stars. It first extracted Java methods and Javadocs, and treated the first sentence of the Javadoc as the ground-truth comment. Since most of the related studies [6, 12, 16, 64, 65] for comment generation tasks are evaluated on JCDSD [27] and PCSD [9] benchmark datasets, in this study, we also select these two datasets in our experiments. JCDSD has 87,136 code-comment pairs collected from more than 9K Java Github repositories created from 2015 to 2016 with at least 20 stars. It first extracted Java methods and Javadocs, and treated the first sentence of the Javadoc as the ground-truth comment. Since most of the related studies [6, 12, 16, 64, 65] for comment generation tasks are evaluated on JCDSD [27] and PCSD [9] benchmark datasets, in this study, we also select these two datasets in our experiments. JCDSD has 87,136 code-comment pairs collected from more than 9K Java Github repositories created from 2015 to 2016 with at least 20 stars. It first extracted Java methods and Javadocs, and treated the first sentence of the Javadoc as the ground-truth comment. Since most of the related studies [6, 12, 16, 64, 65] for comment generation tasks are evaluated on JCDSD [27] and PCSD [9] benchmark datasets, in this study, we also select these two datasets in our experiments. JCDSD has 87,136 code-comment pairs collected from more than 9K Java Github repositories created from 2015 to 2016 with at least 20 stars. It first extracted Java methods and Javadocs, and treated the first sentence of the Javadoc as the ground-truth comment. Since most of the related studies [6, 12, 16, 64, 65] for comment generation tasks are evaluated on JCDSD [27] and PCSD [9] benchmark datasets, in this study, we also select these two datasets in our experiments. JCDSD has 87,136 code-comment pairs collected from more than 9K Java Github repositories created from 2015 to 2016 with at least 20 stars. It first extracted Java methods and Javadocs, and treated the first sentence of the Javadoc as the ground-truth comment. Since most of the related studies [6, 12, 16, 64, 65] for comment generation tasks are evaluated on JCDSD [27] and PCSD [9] benchmark datasets, in this study, we also select these two datasets in our experiments. JCDSD has 87,136 code-comment pairs collected from more than 9K Java Github repositories created from 2015 to 2016 with at least 20 stars. It first extracted Java methods and Javadocs, and treated the first sentence of the Javadoc as the ground-truth comment.
For the sake of fairness, we preprocess the JCSD and PCSD strictly following Rencos [64]. Specifically, we first split datasets into a training set, validation set, and test set in a consistent proportion of 8 : 1 : 1 for the Java dataset and 6 : 2 : 2 for the Python dataset. We use the javalang [3] and tokenize [5] libraries to tokenize the code snippet for JCSD and PCSD, respectively. We further split code tokens of the form CamelCase and snake_case to respective sub-tokens. In common with [64], we remove the exactly duplicated code-comment pairs in the test set for JCSD. The specific statistics of the two preprocessed datasets are shown in Table 1.

![Table 1: Statistic of Datasets](image)

3.2 Evaluation Metrics
We evaluate the performance of different approaches using common metrics including BLEU [43], ROUGE-L [37], METEOR [8], and CIDEr [55]. BLEU measures the n-gram precision by computing the overlap ratios of n-grams and applying brevity penalty on short translation hypotheses. BLEU-1/2/3/4 corresponds to the scores of unigram, 2-grams, 3-grams, and 4-grams, respectively. ROUGE-L is defined as the length of the longest common subsequence between generated sentence and reference, and based on recall scores. METEOR is based on the harmonic mean of unigram precision and recall, with recall weighted higher than precision. CIDEr considers the frequency of n-grams in the reference sentences by computing the TF-IDF weighting for each n-gram. CIDEr\_n score for n-gram is computed using the average cosine similarity between the candidate sentence and the reference.

3.3 Implementation Details
Following previous studies [64], we set the length limits (in terms of #words) of code and comment (i.e., 300 and 30 for JCSD, 100 and 50 for PCSD). To save the computing resource, we limit the maximum vocabulary size of source code and comment to 50K for both datasets. The out-of-vocabulary words are replaced by 'UNK'. The word embedding size of both code and comment is set to 512. Following previous studies [64], we set the length limits (in terms of #words) of code and comment (i.e., 300 and 30 for JCSD, 100 and 50 for PCSD). To save the computing resource, we limit the maximum vocabulary size of source code and comment to 50K for both datasets. The out-of-vocabulary words are replaced by 'UNK'. We set the dimensions of hidden states to 512, the number of heads to 8, and the number of blocks to 6, respectively. The maximum deliberation number \(K\) is set to be 3. We set the mini-batch size to 32 and train our approach using the Adam [31] optimizer. In the first-step training, we set the learning rate to 1e-4, and for the second-step training, we use a smaller learning rate (1e-5) to fine-tune DECOM. To avoid the over-fitting problem, we apply dropout [22] with 0.2. The maximum number of epochs is set to 100 for each step of training. We also use the strategy of early stopping, when the validation performance does not improve for 20 consecutive epochs, the training process will be stopped. To reduce training time, we use the greedy search to generate comments at the training stage. During the prediction stage, we use the beam search [59] and set the beam size to 5 for choosing the best result. Our approach is implemented based on the Pytorch [2] framework. The experimental environment is a desktop computer equipped with an NVIDIA GeForce RTX 3060 GPU, intel core i5 CPU, and 12GB RAM, running on Ubuntu OS.

4 RESULTS
We address the following three research questions to evaluate the performance of DECOM:

- **RQ1**: How does the DECOM perform compared to the state-of-the-art comment generation baselines?
- **RQ2**: How does each individual component in DECOM contribute to the overall performance?
- **RQ3**: What’s the performance of DECOM on the data with different code or comment length?

4.1 RQ1: Comparison with Baselines
4.1.1 Baselines. We compare our approach with three categories of existing work on the comment generation task. We exactly adopt the hyperparameter settings reported in the original paper for all baselines. For a fair comparison, we use the same maximum code and comment length for all approaches, and evaluate their performance using the same training/testing datasets.

- IR-based baselines. **LSI** [14] is an IR technique to analyze the latent meaning or concepts of documents. The similarity between the code and the comment is computed based on the LSI-reduced vectors and cosine distance, and we set the vector dimension to be 500. **VSM** [47] is also a commonly used IR technique in comment generation tasks. For a given code snippet, we represent the code as a vector using TF-IDF, and extract the comment of the most similar code based on cosine similarity. **NNGen** [38] is a nearest-neighbors approach for generating commit messages. It first embeds code into vectors based on the bag of words and the term frequency. Then, it retrieves the nearest neighbors of the code. Finally, it outputs the message of the code with the highest BLEU score.

- NMT-based approaches. **CODE-NN** [28] is the first learning-based model for comment generation. It maps the source code sequence into word embeddings, then uses the LSTM and the attention mechanism to generate comments. **TL-CodeSum** [27] is a multi-encoder neural model that encodes API sequences along with code token sequences and generates comments from source code with transferred API knowledge. **Hybrid-DRL** [56] incorporates ASTs and sequential content of code as a vector using TF-IDF, and extract the comment of the most similar code based on cosine similarity. **EditSum** [34] is the most recent hybrid approach. It leverages the advantages of three types of methods based on neural networks, templates, and IR to improve the performance.
first retrieves the most similar code snippet, and treats the corresponding comment as a prototype. Then, it combines the pattern in the prototype and semantic information of the input code to generate the target comment.

4.1.2 Results. Table 2 shows the comparison results between the performance of DECOM and other baselines, and the best performance is highlighted in bold. Overall, our approach achieves the best performance on all evaluation metrics, followed by Rencos, EditSum, and Re^2. On the Java dataset, DECOM achieves 22.3, 44.5, 19.6, and 2.442 points on BLEU-4, ROUGE-L, METEOR, and CIDEr. Compared with the best baseline (Rencos), DECOM improves the performance of BLEU-4, ROUGE-L, METEOR, and CIDEr by 8.3%, 6.0%, 13.3%, and 6.9%, respectively. On the Python dataset, DECOM achieves 19.1, 41.2, 21.9, and 2.603 points on BLEU-4, ROUGE-L, METEOR, and CIDEr. Compared with the best baseline (Rencos), DECOM also achieves 5.8%, 3.8%, 6.6%, and 6.9% improvements on BLEU-4, ROUGE-L, METEOR, and CIDEr, respectively.

Answering RQ1: DECOM outperforms the state-of-the-art baselines in terms of all seven metrics on both two datasets. Compared to the best baseline Rencos, DECOM improves the performance of BLEU-4, ROUGE-L, METEOR, and CIDEr by 8.3%, 6.0%, 13.3%, and 10.5% on JCSD dataset, by 5.8%, 6.0%, 13.3%, and 10.5% on PCSD dataset, respectively.

4.2 RQ2: Component Analysis

4.2.1 Variants. To evaluate the contribution of core components, we obtain two variants: (1) DECOM w/o Multi-pass Deliberation, which removes the multi-pass deliberation and adopts the one-pass process to generate comments. (2) DECOM w/o Evaluation Model, which removes the evaluation model and takes the comment generated by the last ($K^{th}$) deliberation model as the result. We train the two variants with the same experimental setup as DECOM and evaluate their performance on the test sets of JCSD and PCSD, respectively.

4.2.2 Results. Table 3 presents the performances of DECOM and its two variants. We can see that, removing the two components makes the performance degrade substantially. Specifically, when comparing DECOM and DECOM w/o Multi-pass Deliberation, removing the multi-pass deliberation will lead to a dramatic decrease in the average BLEU-4 (by 6.8%), ROUGE-L (by 3.4%), METEOR (by 5.2%), and CIDEr (by 6.9%) across both datasets. When comparing DECOM and DECOM w/o Evaluation Model, we find that removing the evaluation model will lead to the performance decline in the average BLEU-4 (by 4.8%), ROUGE-L (by 1.6%), METEOR (by 3.5%), and CIDEr (by 4.5%). We can also observe that, removing the multi-pass deliberation will lead to a larger degree of performance decline than removing the evaluation model.

Answering RQ2: Both the multi-pass deliberation and the evaluation model components have positive contributions to the performance of DECOM, where the multi-pass deliberation component contributes more to increasing the performance.

4.3 RQ3: Performance for Different Lengths

4.3.1 Methodology. To answer this question, we analyze the performance of DECOM and best three baselines (i.e. Re^2, Rencos, and EditSum) on different lengths (i.e., number of tokens) of code and comments. We calculate the BLEU-1 score of each sample on the test set of both datasets and average the scores by the length of code and comments, respectively. (Note that, based on our observations, all the seven evaluation metrics show similar trends. For simplicity, we show BLEU-1 only).

4.3.2 Results. Figure 5 presents the performance of DECOM and the three baselines on JCSD and PCSD datasets with code and comments of different lengths, where the red lines denote the performance of DECOM. Overall, we can observe that the performance of DECOM generally outperforms the three baselines with different code and comment lengths on both datasets. Specifically, as the length of the input code increases, DECOM almost keeps a stable
improvement over the other three approaches. The performance of DECOM is nearly the best on all the lengths of Java and Python code snippets. In particular, DECOM can achieve much higher performance than others when the length of the Java code snippet is over 200 words. This shows that DECOM can better understand the semantics of the long code snippets by sharing the information between deliberation models and the evaluation model. For the output comments, we can see that when the output comments are becoming complicated with a relatively long length, the performance of all the approaches decrease, which indicates that the longer the comment, the harder to generate it completely. However, DECOM still has a substantial improvement over the other baselines (as shown in Figure 5(c)), showing that our approach has the ability to generate long and concise comments.

Answering RQ3: DECOM generally outperforms the best three baselines on different lengths of the input code snippets and the output comments, indicating its robustness. In particular for Java, DECOM can achieve much higher performance than others when the code snippets and comments are long.

5 HUMAN EVALUATION

Although the evaluation metrics (i.e., BLEU, ROUGE-L, METEOR, and CIDEr) can measure the lexical gap between the generated comments and the references, it can hardly reflect the semantic gap. Therefore, we perform a human evaluation to further assess the quality of comments generated by different approaches.

5.1 Procedure

We recruited six participants, including three Ph.D. students, one master student, and two senior researchers, who are not co-authors of this paper. They all have at least three years of both Java and Python development experience, and four of them have more than six years of development experience. We randomly select 100 code snippets from the test dataset (50 from JCSD and 50 from PCSD). By applying the best three baselines (i.e., Re2com, Rencos, and EditSum) and DECOM, we obtain a total of 400 generated comments. The 400 code-comment pairs are divided into three groups, and each group is used to create a questionnaire. We randomly list the code-comment pairs on the questionnaire and remove their labels to ensure that the participants are not aware of where the comments are generated from. Each questionnaire is evaluated by two participants, and the final result of a generated comment is the average of two participants. Each participant is asked to rate each generated comment from the three aspects: (1) Naturalness reflects the fluency of generated comments from the perspective of grammar; (2) Informativeness reflects the information richness of generated comments; and (3) Usefulness reflects how can generated comments help developers. All three scores are integers, ranging from 1 to 5 (1 for poor, 2 for marginal, 3 for acceptable, 4 for good, and 5 for excellent).

5.2 Results

Figure 6 exhibits the results of human evaluation by showing the violin plots depicting the naturalness, informativeness, and usefulness of different models, and Table 4 shows the statistic results. Each violin plot contains two parts, i.e., the left and right parts reflect the evaluation results of models on the JCSD dataset and PCSD dataset. The box plots in the violin plots present the distribution of data and the red triangles mean the average scores of the three aspects. Overall, DECOM is better than all baselines in three aspects.
average score for naturalness, informativeness, and usefulness of our approach are 4.24, 3.43, and 3.25, respectively, on the JCSD dataset. On the PCSD dataset, our approach gets the average score of 4.05, 2.96, and 2.87 in terms of naturalness, informativeness, and usefulness. We can see that, the comments generated in the PCSD dataset receive lower scores in human evaluation, while receiving higher scores in evaluation metrics (see Table 2). This is mainly because the PCSD dataset contains shorter comments (see Table 1), thus mistakenly generating fewer keywords may lead to a lower degree of human satisfaction. While the shorter comments are more probable with these N-gram matching metrics [44].

Specifically, in terms of naturalness, our approach achieves average scores above 4 on both JCSD and PCSD datasets, which shows that DECOM can generate fluent and readable comments. Besides, in terms of informativeness and usefulness, DECOM is the only approach with an average score of more than 3 points on the JCSD dataset. It indicates that the comments generated by DECOM tend to be more informative and useful than other baselines.

### 6 DISCUSSION

#### 6.1 Qualitative Analysis

For qualitative analysis of our approach, we present two cases generated by the best three baselines together with DECOM. The cases are selected from the test sets of Java and Python datasets respectively, as shown in Figure 7. Overall, the comments generated by DECOM tend to be more accurate and more readable than the other three baselines. In case 1, the aim of the Java code is to predict the keyword “displays” as “locates”, “writes”, and “locates”, respectively, resulting in the semantics of the generated comments being different from the ground truth. In contrast, the comment generated by DECOM is exactly the same as the ground truth, indicating that our approach can understand the intention of code concisely. In case 2, we can see that, our approach also performs better than other baselines, and the comment generated by DECOM has a high semantic similarity with the ground truth.

We believe that the performance advantage of DECOM mainly comes from two aspects: (1) DECOM can observe the entire previously generated comment and leverage its global information to polish it. While other baselines can only leverage the previously generated words. (2) DECOM employs an evaluation model that can determine the opportunity when the deliberation process should end, as well as learn the semantic relationship between source code and target comments. Besides, the evaluation model shares its two encoders with the deliberation models, which facilitates the information sharing among these models, and enables DECOM to learn a better representation for code and comments.

#### 6.2 Parameter Analysis

Figure 8 illustrates the impact of the maximum number of deliberations $K$ on the performance of DECOM trained on the PCSD dataset, as well as the time cost (Note that, since the JCSD dataset has quite similar results to the PCSD dataset, we only exhibit the results on the PCSD dataset).

We can see that enlarging the maximum deliberation number $K$ generally increases the performance of DECOM. When enlarging $K$ from 1 to 5, the BLEU-1 score increases by 6.0%. We also note that DECOM with $K = 2$ substantially outperforms DECOM with $K = 1$ (i.e one-pass model), which indicates that the deliberation process can greatly improve the comment quality by polishing the previously generated comment. Moreover, for $K$ larger than 3, the
performances slowly increase but the time cost rises exponentially. For example, when enlarging $K$ from 3 to 5, the BLEU-1 score increases by 1% (0.5 points), while the training time increases by 65% (26 hours). Thus, we consider $K = 3$ to be a trade-off choice between effectiveness and efficiency.

6.3 Threats to Validity

There are four main threats to the validity of our approach.

The first threat is that DECOM uses the lexical similarity based method to retrieve the top similar code-comment pair, which may cause the retrieved comment (initial draft) to be semantically different from the target comment. However, the threats can be largely relieved as DECOM generates the target comment by iteratively polishing the previous comments. Specifically, DECOM can correct and refine the retrieved comment in subsequent iterations by leveraging its global information and semantic features of the source code. Thus, even though the dissimilar comment is retrieved, DECOM still can guarantee its performance is not affected.

The second threat to validity is the datasets we use. We only evaluate DECOM on the Java and Python datasets. However, DECOM uses language-agnostic features that can be easily extracted from any programming language. Therefore, we believe that our approach has good generalizability and can perform well on the datasets of other programming languages, such as C# and Ruby.

The third threat relates to the suitability of evaluation metrics. First, recent researchers have raised concern over the use of BLEU [18], warning the community that the way BLEU is used and interpreted can greatly affect its reliability. To mitigate that threat, we also adopt other metrics, i.e., ROUGE, METEOR, and CIDEr, when evaluating performance. Second, there is also a threat related to our human evaluation. We cannot guarantee that each score assigned to every generated comment is fair. To mitigate this threat, each comment is evaluated by six human evaluators, and we use the average score of the two evaluators as the final score.

The fourth threat relates to the errors in the implementation of baselines. To mitigate this issue, we directly use the publicly available code of CODE-NN, TL-CodeSum, Hybrid-DRL, Rencos, and Re$^2$-com to implement baselines. However, the code of EditSum [34] is not available, so we tried our best to understand the paper and re-implement the approach carefully. While we have verified our implementation can achieve similar results as the original EditSum on the same dataset used in its paper.

7 RELATED WORK

7.1 Automatic Comment Generation

The automatic comment generation task is now a rapidly-growing research topic in the community of software engineering and natural language processing.

Early studies typically utilize template-based approaches and information retrieval (IR) based approaches to generate comments. The basic idea of the template-based approach [40, 41, 50] is to extract the keywords from the code snippets and fill them into the predefined templates. Due to the limitations of manually designing templates, these methods are usually time-consuming and have poor generalization. The IR-based approaches [14, 15, 19, 20, 38, 47, 60, 61] aim to use IR techniques to extract keywords from the source code and compose them into term-based comments for a given code snippet. For example, Wong et al. [60] generated a comment for a given code snippet by retrieving the replicated code samples from software repositories with clone detection techniques. However, the IR-based approaches ignore the semantic relationship between source code and natural language, so the comments they generate are poorly readable. Recently, many learning-based methods have been proposed, which train the neural models from a large-scale code-comment corpus to automatically generate comments [11, 26–28, 33, 34, 56–58, 64]. Iyer et al. [28] first treated the comment generation task as an end-to-end translation problem and introduced NMT techniques into code comment generation. Hu et al. [26] converted the Java methods into AST sequence to learn the structural information, and applied a seq2seq model to generate comments. Wei et al. [58] proposed an exemplar-based comment generation method that utilized the comment of the similar code snippet as an exemplar to assist in generating the target comment. Zhang et al. [64] proposed a seq2seq approach that retrieved two similar code snippets for a given code to improve the quality of the generated comment. Further, Li et al. [34] treated the comment of the similar code retrieved from a parallel corpus as a prototype. Based on the semantic differences between input code and similar code, they proposed a seq2seq network to update the prototype and generate comments.

Different from the existing research, we propose a novel framework for automatic comment generation, which performs multiple deliberation processes to iteratively polish the generated comments. DECOM also contains an evaluation model that not only determines whether to end the deliberation process, but also learns the semantic relationship between source code and target comments. The experimental results also prove the superiority of our approach.

7.2 Deliberation Networks

The Deliberation mechanism aims to refine the existing results for further improvement. It has been successfully applied to various domains, such as machine translation [17, 21, 35], question generation [42], image captioning [36], speech recognition [24, 25, 52].

Xia et al. [63] first proposed a deliberation network for sequence generation tasks, which consists of two decoders: a first-pass decoder for generating a draft, and a second-pass decoder for polishing the generated draft to a better sequence. Geng et al. [17] proposed a novel architecture to introduce the deliberation mechanism into the neural machine translation model. It leveraged the policy network...
to determine whether to end the translation process adaptively. Nema et al. [42] utilized the deliberation network to address the automatic question generation task. They proposed a novel approach called Refine Network, which contains two decoders. The second decoder used dual attention to capture information from both (i) the original passage and (ii) the question (initial draft) generated by the first decoder, thereby refining the question generated by the first decoder to make it more correct and complete. Lian et al. [36] proposed a universal two-pass decoding framework for the image captioning task, which contains a drafting model and a deliberation model. The drafting model first generated a draft caption according to an input image, and a deliberation model then refined the draft caption to a better image description. Hu et al. [25] employed the deliberation network for the speech recognition task. They combined acoustics and first-pass text hypotheses for second-pass decoding based on the deliberation network and obtained significant improvements.

The findings of previous work motivate the work presented in this paper. Our study is different from the previous work as we focus on enhancing the performance of the comment generation task by incorporating its own characteristics into the deliberation paradigm. Specifically, we combine the two characteristics of the comment generation task into the deliberation network: (1) since code reuse is widespread in software development, we use retrieval techniques to retrieve the most similar comment to provide an explicit hint about the comment expression; (2) since user-defined identifier names usually contain semantic information, we extract the keywords from the source code to strengthen the semantic features of the source code. To the best of our knowledge, this is the first work that treats the comment generation process as the process of writing and polishing, and utilizes multi-pass deliberation automatically generate comments.

8 CONCLUSION

In this paper, we propose a novel multi-pass deliberation framework for automatic comment generation, named DECOM, which is inspired by human cognitive processes. DECOM relies on multiple deliberation models and one evaluation model to iteratively perform the deliberation process. For each process, the deliberation model refines the previously generated comment into a better one. The evaluation model estimates the quality of the new generated comment, and compares its quality score to the previous one to determine whether to end the iterative process. We use a two-step training strategy to train our framework. The evaluation results show that our approach significantly outperforms all other baselines on both Java and Python datasets. A human evaluation study also confirms the comments generated by DECOM tend to be more readable, informative, and useful. In future work, we plan to incorporate the reinforcement learning techniques (e.g. policy network) into the framework to adaptively choose the suitable deliberation processes, thereby enhancing the performance.

ACKNOWLEDGMENTS

We sincerely appreciate anonymous reviewers for their constructive and insightful suggestions for improving this manuscript. This work is supported by the National Key Research and Development Program of China under Grant No. 2018YFB1403400, the National Science Foundation of China under Grant No. 61802374, 62002348, 62072442, 61422092020 and Youth Innovation Promotion Association Chinese Academy of Sciences.

REFERENCES


