Tram: A Token-level Retrieval-augmented Mechanism for Source Code Summarization

Tong Ye, Lingfei Wu, Tengfei Ma, Xuhong Zhang, Yangkai Du, Peiyu Liu, Shouling Ji, Wenhai Wang

1 Zhejiang University, 2 Anytime.AI, 3 Stony Brook University

{tongye, zhangxuhong, yangkaidu, liupeiyu, sji, zdzzlab}@zju.edu.cn
lwu@anytime-ai.com, tengfei.ma@stonybrook.edu

Abstract

Automatically generating human-readable text describing the functionality of a program is the intent of source code summarization. Although neural language models achieve significant performance in this field, they are limited by their inability to access external knowledge. To address this limitation, an emerging trend is combining neural models with external knowledge through retrieval methods. Previous methods have relied on the sentence-level retrieval paradigm on the encoder side. However, this paradigm is coarse-grained, noise-filled and cannot directly take advantage of the high-quality retrieved summary tokens on the decoder side. In this paper, we propose a fine-grained Token-level retrieval-augmented mechanism (Tram) on the decoder side rather than the encoder side to enhance the performance of neural models and produce more low-frequency tokens in generating summaries. Furthermore, to overcome the challenge of token-level retrieval in capturing contextual code semantics, we also propose integrating code semantics into individual summary tokens. The results of extensive experiments and human evaluation show that our token-level retrieval-augmented approach significantly improves performance and is more interpretable.

1 Introduction

With software functions becoming more comprehensive and complex, it becomes a heavy burden for developers to understand software. It has been reported that nearly 90% (Wan et al., 2018) of effort is used for maintenance, and much of this effort is spent on understanding the maintenance task and related software source codes. Source code summary as a natural language is indispensable in software since humans can easily read and understand it, as shown in Table 1. However, manually writing source code summaries is time-consuming and tedious. Besides, the source code summary is often outdated in continuous software iteration. Hence, automatically generating concise, human-readable source code summaries is critical and meaningful.

Table 1: A sample of source code summarization.

<table>
<thead>
<tr>
<th>Summary: evaluates the ( \cos ) of an interval. Token-level retrieval results at the next generation step ( &quot;\cos&quot; ): cos, tangent, sin, hyperbolic, ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>cos, tangent, sin, hyperbolic, ...</td>
</tr>
</tbody>
</table>

With the development of language models and the linguistic nature of source code, researchers explored Seq2Seq architecture, such as recurrent neural networks to generate summaries (Iyer et al., 2016; Loyola et al., 2017; Liang and Zhu, 2018). Soon afterward, transformer-based models (Ahmad et al., 2020; Wu et al., 2021; Gong et al., 2022) were proposed, outperforming previous RNN-based models by a large margin. Recently, many approaches have been proposed to leverage the structural properties of source code, such as Abstract Syntax Tree (AST) and Program Dependency Graph (PDG). Current structure-aware methods typically either fuse structural information in a hybrid manner (Hu et al., 2018; Shido et al., 2019; LeClair et al., 2020; Choi et al., 2021; Shi et al., 2022).
2021), or use a structured-guided way (Wu et al., 2021; Son et al., 2022; Gong et al., 2022; Guo et al., 2022b; Choi et al., 2023). Although these methods have shown promising results, they primarily focus on leveraging the information within the code to obtain richer code representation without fully utilizing the potential of the available human-written code-summary pairs.

In order to leverage external existing high-quality code and the corresponding summary instances, recent works (Zhang et al., 2020; Li et al., 2021; Liu et al., 2021; Parvez et al., 2021) have proposed a retrieval augmented approach. Their unified paradigm involves sentence-level retrieval, which uses text similarity metrics or code semantic similarity metrics to retrieve the most similar code snippet from a code repository for the given input code snippet. The retrieved code snippet and its corresponding summary are either directly concatenated with the input code snippet or semantically enhanced to augment the input code snippet on the encoder side.

However, the granularity of sentence-level retrieval methods poses challenges. Specifically, they can erroneously retrieve and incorporate code snippets that, while syntactically similar, are semantically distinct or those that only bear partial semantic resemblance. The unintended noise introduced through such mismatches can adversely affect the generation performance, especially for low-frequency tokens. Moreover, code summarization is essentially a generative task, the decoder autoregressively generates the summary tokens. However, previous sentence-level retrieval-augmented methods neglect to fuse the retrieved information on the decoder side, only doing so on the encoder side, which will result in the utilization pattern being indirect and insufficient.

These limitations have inspired us to explore a more fine-grained and sufficient retrieval approach on the summary generation process. In order to achieve the purpose of retrieving semantic similar summary tokens on the decoder side, we first construct a datastore to store the summary tokens and corresponding representations through a pre-trained base model offline. Meanwhile, to overcome the challenge of not fully utilizing code semantics on the encoder side when retrieving on the decoder side, we intelligently fuse summary token representation with code token representation and AST node representation with attention weight. This approach fully considers contextual code semantics associated with summary tokens. Then, at each generation step, the fused summary token representation is used to retrieve the top-$K$ most similar tokens. As illustrated in Table 1, the token-level retrieval results at the next token generation step “cos” are “cos, tangent, sin, hyperbolic, · · · ”. The retrieved top-$K$ tokens are expanded to a probability distribution, which we refer to as the retrieval-based distribution. The retrieval-based distribution is then fused with the vanilla distribution to form the final distribution. Additionally, our proposed token-level retrieval mechanism can be seamlessly integrated with existing sentence-level retrieval methods and code-related large pre-trained models.

To facilitate future research, we have made our code publicly available.$^1$ Overall, the main contributions of this paper can be outlined as follows:

1. We are the first to explore a Token-level retrieval-augmented mechanism (Tram) on the decoder side for source code summarization.
2. Our proposed retrieval-augmented mechanism is orthogonal to existing improvements, such as better code representation, additional sentence-level retrieval approaches, and pre-trained models.
3. Extensive experiments and human evaluation show that Tram significantly outperforms other baseline models, generates more low-frequency tokens and is more interpretable.

2 Related Works

Retrieval-based Source Code Summarization. Liu et al. (2021) retrieved the most similar code snippet by text similarity metric to enrich target code structure information for getting a better code representation encoder. This retrieval method only carries out from the perspective of text similarity and neglects code semantic similarity in the retrieval phase. Besides, the summary corresponding to the retrieved code snippet is just a simple concatenation to the encoder. Zhang et al. (2020); Parvez et al. (2021) used a pre-trained encoder to obtain code semantic representation, which was used to retrieve similar code snippets. The former only uses similar code snippets and discards the corresponding summaries; the latter directly splice the retrieved code snippet and the corresponding summary behind the target code; both are also aimed at better code representation on the encoder side.

$^1$https://github.com/tongye98/SourceCodeSummary
methods, Tram performs token-level retrieval augmentation at each step of the decoder that generates the next token.

**K-Nearest-Neighbor Machine Translation.** Recently, non-parametric methods have been successfully applied to neural machine translation (Khandelwal et al., 2021; Jiang et al., 2021; Zheng et al., 2021a,b). These approaches complement advanced NMT models with external memory to alleviate the performance degradation in domain adaptation. Compared to these works, we have fully accounted for the code’s inherent structure and have intelligently integrated code semantics into the retrieval process. Additionally, we demonstrate how Tram integrates with sentence-level retrieval methods.

## 3 Methodology

### 3.1 Overview

The overview architecture of Tram is shown in Figure 1. Initially, we introduce the base model, which is an encoder-decoder architecture that takes a code snippet and corresponding AST as input and generates a summary as output. Building upon the base model, we then construct a datastore that stores summary tokens and corresponding representations, where the representation is an intelligent combination of the decoder representation, code token representation, and AST node representation. Next, we develop a fine-grained token-level retrieval mechanism. This mechanism focuses on retrieving the top-\(K\) most similar tokens from the datastore and generating a retrieval-based distribution. The retrieval-based distribution is then fused with the vanilla base model distribution by a weight hyper-parameter \(\lambda\) to form the final distribution. Additionally, we detail the integration of both token-level and sentence-level retrieval. The combination of token-level retrieval and sentence-level retrieval enables a more comprehensive summarization process. In terms of integrating Tram with code pre-trained models, the implementation is broadly consistent and detailed in Appendix A.

### 3.2 Base Model

The base model serves as the foundation for the subsequent retrieval process. It is designed to construct the datastore and generate the base model distribution. Figure 2 illustrates the specific architecture of the base model, which consists of two encoders (SCEnc and ASTEnc) and a decoder.

**Source Code Encoder (SCEnc).** As shown in Figure 2, we utilize Transformer (Vaswani et al., 2017) as the encoder for the source code tokens. The Transformer consists of stacked multi-head attention and parameterized linear transformation layers. Each layer emphasizes on self-attention mechanism. Nevertheless, as pointed out in Ahmad et al. (2020), the code semantic representation is influenced by the mutual interactions between its tokens rather than their absolute positions. Therefore, we adopt the method of relative positional encoding, as proposed by Shaw et al. (2018).

Assuming the code snippet contains \(p\) tokens \([t_1, t_2, \ldots, t_p]\), after SCEnc, each token has a hidden representation, which is denoted as:

\[
[h_1, h_2, \ldots, h_p] = SCEnc([t_1, t_2, \ldots, t_p])
\]

**AST Encoder (ASTEnc).** Furthermore, the AST of the source code can be considered as a graph.
structure, making it suitable for representation and learning using Graph Neural Networks (GNNs). Taking advantage of the GAT’s (Veličković et al., 2018) exceptional performance and its ability to assign adaptive attention weights to different nodes, we employ GAT to represent each node in the AST. The graph encoder layer processes the AST by first aggregating the neighbors of the nodes with edge information. It then updates the nodes with the aggregated information from their neighborhoods.

After updating the node information, the node representations are put together into a ReLU activation followed by residual connection (He et al., 2016) and layer normalization (Ba et al., 2016).

Assuming the AST of the code snippet contains \( q \) nodes \([n_1, n_2, ..., n_q]\), after the ASTEnc, each node has a hidden representation, denoted as:

\[
[r_1, r_2, ..., r_q] = ASTEnc([n_1, n_2, ..., n_q])
\]

**Summary Decoder.** The summary decoder is designed with modified transformer decoding blocks. At time step \( t \), given the existing summary tokens \([s_1, s_2, ..., s_{t-1}]\), the decoding blocks first encode them by masked multi-head attention. After that, we expand the transformer block by leveraging two multi-head cross-attention modules to interact with the two encoders for summary decoding. One multi-head cross-attention module is performed over the code token features to get the first-stage decoded information, which will then be fed into the other over the learned AST node features for the second-stage decoding. Then the decoded summary vectors \([d_1, d_2, ..., d_{t-1}]\) are put into a feed-forward network for non-linear transformation.

**3.3 Datastore Construction**

Based on the base model, to achieve the goal of fine-grained token-level retrieval, we build the datastore that stores summary tokens and corresponding representations. At the stage of datastore establishment, we adopt the above pre-trained base model to go through all training instances in an offline manner. During this process, for each instance, the SCEnc and ASTEnc encode the code tokens and AST nodes into a sequence of hidden states: \([h_1, h_2, ..., h_p]\) and \([r_1, r_2, ..., r_p]\), the decoder generates the target summary autoregressively. At time step \( t \), the decoder takes existing summary token \([s_1, s_2, ..., s_{t-1}]\) as input, for the last token \( s_{t-1} \), the decoder’s first cross-attention module gets the attention score of the code tokens (called Attend-Code \([\alpha_1, \alpha_2, ..., \alpha_p]\)) and the second cross-attention module gets the attention score of the AST nodes (called Attend-Node \([\beta_1, \beta_2, ..., \beta_q]\)). We use Attend-Code and Attend-Node to perform weighted summation of the representations of code tokens and AST nodes, respectively, denoted as:

\[
\begin{align*}
[\alpha_1, \alpha_2, ..., \alpha_p] & \ast [h_1, h_2, ..., h_p]^T = H_t \\
[\beta_1, \beta_2, ..., \beta_q] & \ast [r_1, r_2, ..., r_p]^T = R_t
\end{align*}
\]

where \( H_t \) means weighted code token representation, \( R_t \) means weighted AST node representation.

After two cross-attention modules, the input token \( s_{t-1} \) is converted to token representation \( d_{t-1} \).

Because the goal at time step \( t \) is to generate the next token \( s_t \), we pick the token representation \( d_{t-1} \) to represent \( s_t \). To fully consider the contextual code semantics associated with the summary token, we concatenate \( H_t \), \( R_t \), and \( d_{t-1} \) to create the final and more comprehensive representation of \( s_t \). Besides, to facilitate efficient retrieval in the subsequent steps, we applied \( L_2 \) regularization to the representations in practice, denoted as:

\[
\begin{align*}
& k_t = Concat(H_t, R_t, d_{t-1}) \\
& \tilde{k}_t = L_2-Normalize(k_t)
\end{align*}
\]

where \( \tilde{k}_t \) is the final presentation of token \( s_t \). Finally, the ground-truth summary token \( s_t \) and corresponding representation \( \tilde{k}_t \) are inserted into datastore as a key-value pair, denoted as (key, value) = \((\tilde{k}_t, s_t)\), the whole datastore can be denoted as:

\[
\langle K, V \rangle = \{(\tilde{k}_t, s_t), \forall s_t \in S\}
\]

where \( S \) means all summary tokens in the training dataset. It is important to note that the datastore contains duplicate tokens because the same summary token can have different keys, representing different semantic representations due to variations in linguistic contexts.

**3.4 Token-level Retrieval**

During inference, at each decoding step \( t \), the current summary token representation \( d_{t-1} \) is combined with the corresponding \( H_t \) and \( R_t \) using the same concatenate and \( L_2 \) regularization operator as query \( q_t \). The query retrieves the top-\( K \) most similar summary tokens in the datastore according to cosine similarity distance. It is worth noting that we use cosine similarity instead of squared-\( L^2 \) distance because of the performance...
of the preliminary experiment. As an added bonus, cosine similarity can be seen as retrieval confidence. In practice, the retrieval over millions of key-value pairs is carried out using FAISS (Johnson et al., 2019), a library for fast nearest neighbor search in high-dimensional spaces. The retrieved key-value pairs \((k, v)\) and corresponding cosine similarity distance \(\alpha\) composed a triple set \(\mathcal{N} = \{(k_i, v_i, \alpha_i) \mid i = 1, 2, \ldots, K\}\). Inspired by KNN-MT (Khandelwal et al., 2021), the triple set can then be expanded and normalized to the retrieval-based distribution as follows:

\[
P_r(s_t|c, \hat{s}_{<t}) \propto \sum_{(k_i, v_i, \alpha_i) \in \mathcal{N}} \mathbb{I}_{v_i=v_t} \exp (g(k_i, \alpha_i))
\]

\[
g(k_i, \alpha_i) = \alpha_i \times T
\]

where \(g(\cdot)\) can be any Kernel Density Estimation (KDE); in practice, we use the product form; \(T\) is the temperature to regulate probability distribution.

3.5 Fused Distribution

The final prediction distribution can be seen as a combination of the vanilla base model output distribution and the retrieval-based distribution, which is interpolated by a hyper-parameter \(\lambda\):

\[
P(s_t|c, \hat{s}_{<t}) = \lambda \times P_r(s_t|c, \hat{s}_{<t}) + (1 - \lambda) \times P_m(s_t|c, \hat{s}_{<t})
\]

where \(P_m\) indicates the base model distribution.

3.6 Additional Sentence-level Retrieval

Our proposed token-level retrieval augmented method can also be seamlessly incorporated with additional sentence-level retrieval. Sentence-level retrieval here means using the target code snippet to retrieve the most semantically similar code snippet in the corpus through code semantic representations. Then we assign an additional but the same base model for the most similar code snippet to generate tokens autoregressively. At each generation step, the decoder of the additional base model (generating similar-code-based next token distribution) is synchronous with the original target code snippet decoder (generating base model next token distribution). Finally, the above two distributions, together with the “token-level retrieved next token distribution”, form the final distribution through a weighted sum, which is denoted as:

\[
P(s_t|c, \hat{s}_{<t}) = \lambda_1 \times P_r(s_t|c, \hat{s}_{<t}) + \lambda_2 \times \text{Sim} \times P_s(s_t|c, \hat{s}_{<t}) + (1 - \lambda_1 - \lambda_2) \times P_m(s_t|c, \hat{s}_{<t})
\]

where \(P_s\) is the additional base model produced distribution, \(\langle c \rangle\) is the most semantically similar code snippet to the target code snippet \(c\), and \(\text{Sim}\) is the corresponding similarity score.

4 Experiments

4.1 Experimental Setup

Datasets. We conduct the experiments on four public benchmarks of Java (Hu et al., 2018), Python (Wan et al., 2018), CCSD (C Code Summarization Dataset) (Liu et al., 2021), and Python‡ (Zhang et al., 2020). The partitioning of train/validation/test sets follows the original datasets. The statistics of the four datasets are shown in Table 2.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Java</th>
<th>Python</th>
<th>CCSD</th>
<th>Python‡</th>
</tr>
</thead>
<tbody>
<tr>
<td>Train</td>
<td>69,708</td>
<td>55,538</td>
<td>84,316</td>
<td>65,236</td>
</tr>
<tr>
<td>Validation</td>
<td>8,714</td>
<td>18,505</td>
<td>4,432</td>
<td>21,745</td>
</tr>
<tr>
<td>Test</td>
<td>8,714</td>
<td>18,502</td>
<td>4,203</td>
<td>21,745</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code: Avg. tokens</th>
<th>Java</th>
<th>Python</th>
<th>CCSD</th>
<th>Python‡</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary: Avg. tokens</td>
<td>73.76</td>
<td>49.42</td>
<td>68.59</td>
<td>150.82</td>
</tr>
</tbody>
</table>

Table 2: Statistics of the experimental datasets.

Out-of-Vocabulary. The vast operators and identifiers in program language may produce a much larger vocabulary than natural language, which can cause Out-of-Vocabulary problem. To avoid this problem, we apply CamelCase and snake_case tokenizers that are consistent with recent works (Gong et al., 2022; Wu et al., 2021; Ahmad et al., 2020) to reduce the vocabulary size of source code.

Metrics. Similar to recent work (Gong et al., 2022; Son et al., 2022), we evaluate the source code summarization performance using three widely-used metrics, BLEU (Papineni et al., 2002), METEOR (Banerjee and Lavie, 2005) and ROUGE-L (Lin, 2004). Furthermore, considering the essence of source code summarization to help humans better understand code, we also conduct a human evaluation study. The volunteers are asked to rank summaries generated from the anonymized approaches from 1 to 5 (i.e., 1: Poor, 2: Marginal, 3: Acceptable, 4: Good, 5: Excellent) based on Similarity, Relevance, and Fluency metrics. Further details on human evaluation can be found in Appendix C.

Training Details. We implement our approach based on JoeyNMT (Kreutzer et al., 2019). The batch size is set to 32 and Adam optimizer is used with an initial learning rate $10^{-4}$. To alleviate overfitting, we adopt early stopping with patience 15. For Faiss (Johnson et al., 2019) Index, we employ
IndexFlatIP and top-$K=16$ to maintain a balance between retrieval quality and retrieval speed in the large-scale datastore. It is worth noting that only the base model requires training, and once trained, all the parameters of the base model are fixed. For validation, we use greedy search, while for evaluation, we use beam search with beam size of 4.

### 4.2 Baselines

**Transformer-based.** Transformer (Ahmad et al., 2020) is the first attempt to use transformer architecture in this field. Soon, structure-aware methods were proposed. Among these are CAST (Shi et al., 2021) and mAST+GCN (Choi et al., 2021), which integrate structural information in a hybrid manner. SiT (Wu et al., 2021), SiT+PDG (Son et al., 2022), and CODESCRIBE (Guo et al., 2022b) utilize a structured-guided way. The detailed description of these baselines is shown in Appendix B.

**Retrieval-based.** Rencos (Zhang et al., 2020) is the first retrieval-based Seq2Seq model, which computes a joint probability conditioned on both the original source code and the retrieved most similar source code for a summary generation. HGNN (Liu et al., 2021) is the retrieval-based GNN model, which retrieval the most similar code and uses a Hybrid GNN by fusing static graph and dynamic graph to capture global code graph information.

### 4.3 Main Results

The main experiment results are shown in Table 3 and Table 4 in terms of three automatic evaluation metrics. The reason we have two tables is that transformer-based works compare their performance on the widely-used Java and Python benchmarks, while the retrieval-based works use two different benchmarks, namely CCSD and Python‡. Thus, our experiments are performed on all four datasets for a more thorough comparison. We calculate the metric values following the same scripts.

From Table 3, SiT + PDG and CODESCRIBE achieve better results than all previous works. However, it is worth noting that even our base model can achieve comparable performance to other models. This is due to the improved training method we used, Pre-LN (layer normalization inside the residual blocks), which is discussed in (Liu et al., 2020). This method enhances the stability of the training process and leads to better performance. Tram further boosts results with 1.39 BLEU points on Java and 1.53 BLEU points on Python. We also observe that the performance improvement for Python is better than that for Java. The main reason we speculate is that Java has a longer average code token length (from

---


---

Table 3: Comparison of the performance of our method with other baseline methods on Java and Python benchmarks in terms of BLEU, ROUGE-L, and METEOR. The results of baseline models are reported in their original papers. '-' refers to no corresponding value from the paper. HR refers to code token and AST node representation; SenRe refers to additional sentence-level retrieval. All of our results are the mean of 5 runs with different random seeds.
Table 4: Comparison of other retrieval methods. HR means code token and AST node representation; SenRe means additional sentence-level retrieval. All of our results are the mean of 5 runs with different random seeds.

<table>
<thead>
<tr>
<th>Model</th>
<th>CCSD BLEU</th>
<th>ROUGE-L</th>
<th>METEOR</th>
<th>Python BLEU</th>
<th>ROUGE-L</th>
<th>METEOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rencos (Zhang et al., 2020)</td>
<td>14.80</td>
<td>31.41</td>
<td>14.64</td>
<td>34.73</td>
<td>47.53</td>
<td>21.06</td>
</tr>
<tr>
<td>HGNN (Liu et al., 2021)</td>
<td>16.72</td>
<td>34.29</td>
<td>16.25</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5: Human Evaluation on Java and Python† datasets.

<table>
<thead>
<tr>
<th>Model</th>
<th>Java Similarity</th>
<th>Relevance</th>
<th>Fluency</th>
<th>Python† Similarity</th>
<th>Relevance</th>
<th>Fluency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rencos</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3.07</td>
<td>3.06</td>
<td>3.96</td>
</tr>
<tr>
<td>CODESCRIBE</td>
<td>3.67</td>
<td>3.72</td>
<td>4.16</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Base</td>
<td>3.62</td>
<td>3.64</td>
<td>4.10</td>
<td>3.20</td>
<td>3.24</td>
<td>4.03</td>
</tr>
<tr>
<td>Tram</td>
<td>3.83</td>
<td>3.89</td>
<td>4.23</td>
<td>3.33</td>
<td>3.44</td>
<td>4.14</td>
</tr>
</tbody>
</table>

Table 2) and richer code structure information.

In Table 4, we compare Tram with other retrieval-based models on CCSD and Python† benchmarks. Our base model is even superior to other retrieval-based methods; the main reason is that the backbone 3 are different. We reproduce Rencos architecture‡ in our base model for a fair comparison, which we denoted as “Base + Rencos”. Tram outperforms all other retrieval-based methods, further improving performance with 2.05 BLEU points and 1.47 BLEU points on CCSD and Python†, respectively. Furthermore, as shown in Table 3 and 4, enhancing Tram with additional sentence-level retrieval (refer as “Tram w/o HR”) and its integration with code pre-trained models ("Our Method on Pre-trained Models" section in Table 3) leads to a notable improvement in performance.

4.4 Ablation Study

To validate the effectiveness of intelligently fusing summary token representation with code token representation \( H_t \) and AST node representation \( R_t \), we conduct an ablation experiment where we eliminate the \( H_t \), \( R_t \), and directly use \( d_{t-1} \) to represent target summary token \( s_t \) for comparison (refer as “Tram w/o HR”). As shown in Table 3 and 4, the performance declined by 0.47, 0.60, 0.21, and 0.32 BLEU points for Java, Python, CCSD, and Python†, respectively. This decline in performance across all datasets demonstrated the importance of fusing code semantics into the summary token for effective token-level retrieval on the decoder side.

\[ \text{Figure 3: } \lambda \text{ and } T \text{ selections in Java and Python datasets.} \]

4.5 Human Evaluation

We perform a human evaluation (details provided in Appendix C) to assess the quality of the generated

3Other retrieval-based methods are RNN-based.  
‡HGNN code is not open source.
void scsi_netlink_init(void){
struct netlink_kernle_cfg cfg;
cfg.input = scsi_nl_rcv_msg;
cfg.groups = SCSI_NL_GRP_CNT;
scsi_nl_sock = netlink_kernel_create(&init_net, NETLINK_SCSI_TRANSPORT, &cfg);
if (!scsi_nl_sock){
    printk(KERN_ERR "%s: register of receive handler failed\n", __func__);
    return;
}
return;
}

Base: called by scsi netlink initialization to register the scsi netlink interface.
Rencos: called by scsi netlink interface to register the scsi netlink interface.
Tram: called by scsi subsystem to register the scsi transport netlink interface.
Human Written: called by scsi subsystem to initialize the scsi transport netlink interface.

Table 6: A Python instance. The bold red font is the keyword of generated summary. The Retrieval Results line is the visible retrieval results and corresponding probability after applying softmax on the keyword generation step.

<table>
<thead>
<tr>
<th>Token Frequency</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>10</th>
<th>50</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Base</td>
<td>126</td>
<td>75</td>
<td>45</td>
<td>27</td>
<td>28</td>
<td>16</td>
</tr>
<tr>
<td>Java Rencos</td>
<td>243</td>
<td>138</td>
<td>73</td>
<td>38</td>
<td>37</td>
<td>18</td>
</tr>
<tr>
<td>Java Tram</td>
<td>307</td>
<td>164</td>
<td>115</td>
<td>51</td>
<td>42</td>
<td>21</td>
</tr>
<tr>
<td>Python Base</td>
<td>452</td>
<td>376</td>
<td>272</td>
<td>176</td>
<td>84</td>
<td>82</td>
</tr>
<tr>
<td>Python Rencos</td>
<td>799</td>
<td>515</td>
<td>344</td>
<td>223</td>
<td>88</td>
<td>109</td>
</tr>
<tr>
<td>Python Tram</td>
<td>983</td>
<td>647</td>
<td>405</td>
<td>298</td>
<td>103</td>
<td>121</td>
</tr>
</tbody>
</table>

Table 7: Count of Accurately Generated Low-Frequency Tokens.

5 Analysis

5.1 Hyperparameters Analysis
Tram has two primary hyperparameters: $\lambda$ and $T$. $\lambda$ means the weight of the retrieval-based distribution component in the final distribution; the higher value indicates greater reliance on retrieval results, and vice versa. $T$ means temperature, which smooths the retrieval-based distribution. We plot the performance of Tram with different hyperparameter selections in Figure 3. The value of $\lambda$ has a significant impact on the final performance, and we find that different datasets have different optimal values (i.e., $\lambda = 0.5$ for Java and $\lambda = 0.6$ for Python). We also observe that $\lambda = 1$ outperforms $\lambda = 0$. The reason is related to the BLEU score (detailed cause analysis provided in Appendix D). Regarding $T$, if it is too small, the retrieval-based distribution cannot be adequately distinguished; while if it is too large, the retrieval-based distribution will concentrate on a single token. Our final results indicate that both extremes result in a performance decrease.

5.2 Token Frequency In-Depth Analysis
Compared to the coarse-grained retrieval approach at the sentence-level, the token-level retrieval can capture the top-$K$ most semantically relevant tokens at every step. This can increase the likelihood of generating those low-frequency tokens in the summary text. Since these low-frequency tokens and their corresponding representations are stored in the datastore, by retrieving the most semantically similar tokens at each generation step, these low-frequency tokens can be more easily and directly fetched from the datastore compared to purely model generated. We further conduct an in-depth statistical analysis of the generation quantity of low-frequency tokens. We first collect all the correctly generated tokens according to the ground-truth summaries. Then we count the frequencies of all these correct tokens in the training set and record the number of the correct and low-frequency tokens (frequency $\leq 100$). From Table 7, we can see that Tram can correctly predict more low-frequency tokens than Rencos (sentence-level retrieval) and Base (vanilla model generated) when the token frequency is small ($\leq 100$).

5.3 Datastore Quality and Robustness Analysis
To accurately assess the impact of datastore quality on Tram’s performance, we conduct robustness experiments where noise is intentionally introduced into the datastore. Specifically, we randomly shuffle a certain percentage of (representation, token) pairs, leading to misaligned pairings. These experi-
ments, conducted using Python and Java datasets, are based on the averages from five separate runs. We introduce noise levels of 5%, 10%, and 20%, corresponding to the proportion of misaligned pairs in the datastore. Table 8 presents the experimental results, indicating that even with a 10% noise level in the datastore, the BLEU score reduction is only up to 0.3 points. Furthermore, even under 20% noise conditions, the model maintains robust performance. These results suggest that the impact of datastore quality and the presence of noisy or poorly aligned pairs is relatively minimal, confirming the robustness of both the datastore and our Tram method.

5.4 Qualitative Analysis

We provide a python example in Table 6 to demonstrate the effectiveness and interpretability of Tram. The qualitative analysis reveals that, compared to other models, Tram enables visualization of the Retrieval Results and corresponding probability at each generation step, as depicted in the last line, making our approach more interpretable. More visualized instances can be found in Appendix E.

6 Conclusion

In this paper, we propose a novel token-level retrieval-augmented mechanism for source code summarization. By a well-designed fine-grained retrieval pattern, Tram can effectively incorporate external human-written code-summary pairs on the decoder side. Extensive experiments and human evaluation show that Tram not only significantly improves performance but also generates more low-frequency tokens and enhances interpretability.

Limitations

Our retrieval-augmented method (Tram) takes full advantage of external retrieval information, and the performance improvement relies on high-quality code-summary token-level pairs. However, there exists some noise in the datastore which will bias the final token distribution; therefore, dealing with noise deserves our deeper exploration. Furthermore, our experiments are only on high-resource programming language (Python, Java, C) scenarios; exploring how to apply our model in a low-resource programming language (Ruby, Go, etc.) is our future direction.

Acknowledgements

This work was partly supported by NSFC under Grant No. 62302443, the Fellowship of China National Postdoctoral Program for Innovative Talents (BX20230307), the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform). This research was also supported by the advanced computing resources provided by the Supercomputing Center of Hangzhou City University.

References


---

Table 8: Datastore Quality and Robustness Analysis at Different Noise Levels.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>BLEU</th>
<th>ROUGE-L</th>
<th>METEOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vanilla</td>
<td>35.97</td>
<td>49.92</td>
<td>22.09</td>
</tr>
<tr>
<td>Noise-5%</td>
<td>35.84</td>
<td>49.79</td>
<td>21.98</td>
</tr>
<tr>
<td>Noise-10%</td>
<td>35.68</td>
<td>49.67</td>
<td>21.85</td>
</tr>
<tr>
<td>Noise-20%</td>
<td>35.49</td>
<td>49.33</td>
<td>21.70</td>
</tr>
<tr>
<td>Java</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vanilla</td>
<td>48.32</td>
<td>58.13</td>
<td>29.56</td>
</tr>
<tr>
<td>Noise-5%</td>
<td>48.15</td>
<td>57.95</td>
<td>29.44</td>
</tr>
<tr>
<td>Noise-10%</td>
<td>48.07</td>
<td>57.90</td>
<td>29.37</td>
</tr>
<tr>
<td>Noise-20%</td>
<td>47.82</td>
<td>57.61</td>
<td>28.81</td>
</tr>
</tbody>
</table>


A Integration of Tram with Code Pre-trained Models

We need to clarify that our Tram can be integrated with generative code pre-trained models (encoder-decoder architecture), such as CodeT5 (Wang et al., 2021) and UniXcoder (Guo et al., 2022a), but is not suitable for code pre-trained models used for code understanding (encoder-only architecture),
like CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2021).

Specifically, the integration process is similar to the Methodology section and primarily consists of three steps:

1. We use Java (Hu et al., 2018) and Python (Wan et al., 2018) datasets to fine-tune the code pre-trained models, respectively, and treat the fine-tuned models as base models;
2. During the datastore establishment phase, the process aligns with that described in the Datastore Construction section. However, we have omitted the AST input to satisfy the input conditions of the code pre-trained models;
3. Token-level Retrieval: The retrieved top-$K$ tokens are expanded to a probability distribution (which we refer to as the retrieval-based distribution). Then we fused the retrieval-based distribution with the vanilla distribution built on the original vocabulary table of the code pre-trained models to obtain the final distribution.

B Details on Transformer-based Methods

Transformer (Ahmad et al., 2020) is the first attempt to use transformer architecture, equipped with relative positional encoding and copy mechanism (See et al., 2017), effectively capturing long-range dependencies of source code. CAST (Shi et al., 2021) hierarchically splits a large AST into a set of subtrees and utilizes a recursive neural network to encode the subtrees. The aim is to capture the rich information in ASTs. mAST + GCN (Choi et al., 2021) adopt the AST and graph convolution to model the structural information and the transformer to model the sequential information. SiT (Wu et al., 2021) incorporates a multi-view graph matrix into the transformer’s self-attention mechanism. SiT + PDG (Son et al., 2022) points program dependency graph is more effective for expressing the structural information than AST. CODESCRIBE (Guo et al., 2022b) model the hierarchical syntax structure of code by introducing a novel triplet position.

C Human Evaluation

In our human evaluation, we invited 3 PhD students and 5 master students with at least 2-5 years of software engineering experience as volunteers. We conduct a small-scale random dataset (i.e., 100 random Java samples and 100 random Python samples). The volunteers are asked to rank summaries generated from the anonymized approaches from 1 to 5 (i.e., 1: Poor, 2: Marginal, 3: Acceptable, 4: Good, 5: Excellent) based on the three following questions:

- **Similarity**: How similar of generated summary and ground truth?
- **Relevance**: Is the generated summary relevant to the source code?
- **Fluency**: Is the generated summary syntactically correct and fluent?

For each evaluation summary, the rating scale is from 1 to 5, where a higher score means better quality. Responses from all volunteers are collected and averaged.

D Cause Analysis: Performance

Superiority of $\lambda = 1$ over $\lambda = 0$

$\lambda$ means the weight of the retrieval-based distribution component in the final distribution. The reason is related to the BLEU score. The BLEU metric measures the similarity between two sentences by assessing the overlap of words between them. Model-generated sentences tend to produce more common words, leading to better fluency; in contrast, sentences generated through retrieval methods are more likely to include factual terms, which, when evaluated using the BLEU score, results in a higher score (Reiter, 2018). However, it may sacrifice the language quality.

For example, given the ground truth "**start a source file within a compilation unit.**", the retrieval-based generation with $\lambda = 1$: "**start file within a compilation unit.**", achieves a BLEU score of 48.78. This is higher than the model-based generation with $\lambda = 0$: "**start the source file within the unit.**", which scores a BLEU of 33.17. Indeed, neither $\lambda = 1$ or $\lambda = 0$ is good enough, and we need a trade-off between the retrieval and the model generation.

E Qualitative Examples

Table 9 shows a couple of qualitative examples to demonstrate the effectiveness and interpretability of Tram.
```c
void batadv_sysfs_del_meshif(struct net_device *dev)
{
    struct batadv_priv *bat_priv = netdev_priv(dev);
    struct batadv_attribute **bat_attr;
    for (bat_attr = batadv_mesh_attrs; *bat_attr; ++bat_attr)
        sysfs_remove_file(bat_priv->mesh_obj, &(*bat_attr)->attr);

    kobject_uevent(bat_priv->mesh_obj, KOBJ_REMOVE);
    kobject_del(bat_priv->mesh_obj);
    kobject_put(bat_priv->mesh_obj);
    bat_priv->mesh_obj = NULL;
}
```

**Base:** Remove mesh interface-related sysfs entries.

**Rencos:** Delete mesh junction sysfs attributes.

**Tram:** Remove soft interface specific sysfs entries.

**Human Written:** Remove soft interface specific sysfs entries.

**Retrieval Results:**
- "interface" (0.82), "portal" (0.11), "bridge" (0.04), "junction" (0.0086), "link" (0.0013) · · ·

```python
def category_structure(category, site):
    return {
        'description': category.title,
        'html.Url': ('%s://%s%s' % (PROTOCOL, site.domain,
                                    category.get_absolute_url())),
        'rss.Url': ('%s://%s%s' % (PROTOCOL, site.domain,
                                    reverse('zinnia:category_feed', args=[category.tree_path]))),
        'category.Id': category.pk,
        'parent.Id': ((category.parent and category.parent.pk) or 0),
        'category.Description': category.description,
        'category.Name': category.title
    }
```

**Base:** updates the structure.

**Rencos:** a post structure.

**Tram:** a category structure.

**Human Written:** a category structure.

**Retrieval Results:**
- "category" (0.43), "tag" (0.11), "post" (0.07), "helper" (0.06), "version" (0.06) · · ·

Table 9: Task samples. The first is a C instance; the second is a Python instance. The bold red font is the keyword of the generated summary. The **Retrieval Results** line is the visible retrieval results and corresponding probability after applying softmax on the keyword generation step.