LEANCODE: Understanding Models Better for Code Simplification of Pre-trained Large Language Models

Yan Wang[1]*, Ling Ding[1]*, Tien N. Nguyen[2]*, Shaohua Wang[1]†, Yanan Zheng[3]†

[1]Central University of Finance and Economics,
[2]University of Texas at Dallas, [3]Yale University

{dayanking,imlingding}@gmail.com, tien.n.nguyen@utdallas.edu,
davidshwang@ieee.org, yanan.zheng@yale.edu

Abstract

Large Language Models for code often entail significant computational complexity, which grows significantly with the length of the input code sequence. We propose LEANCODE for code simplification to reduce training and prediction time, leveraging *code contexts in utilizing attention scores* to represent the tokens' importance. We advocate for the selective removal of tokens based on the average *context-aware* attention scores rather than average scores across all inputs. LEANCODE uses the attention scores of 'CLS' tokens within the encoder for classification tasks, such as code search. It also employs the encoder-decoder attention scores to determine token significance for sequence-to-sequence tasks like code summarization. Our evaluation shows LEANCODE's superiority over the SOTAs DIETCODE and SLIMCODE, with improvements of 60% and 16% for code search, and 29% and 27% for code summarization, respectively.

1 Introduction

Pre-trained Large Language Models (PLLMs) demand significant computational resources, often constraining input word or code token lengths. For example, when using CodeBERT (Feng et al., 2020a) locally, there's a limitation of 512 tokens. CodeT5 (Wang et al., 2021b), CodeGen (Nijkamp et al., 2022), and GPT-4 (ChatGPT) also entail high computational overheads and costs, particularly with longer input code sequences. Code simplification of PLLMs is a practical way to reduce training and prediction time, while maintaining performance of a PLLM as much as possible. Given various pre-trained models and downstream tasks, it is intuitive that not all input tokens play critical roles in downstream-tasks. To tackle this challenge, the state-of-the art approaches, like DietCode (Zhang et al., 2022) and SlimCode (Wang et al., 2024), were proposed to simply the input program of a PLLM.

First, DietCode (Zhang et al., 2022) computes the average self-attention score for each code token across various contexts(global attention scores), treating it as the representative importance score for the token across the entire dataset. It then employs this score to determine whether to eliminate the code token from all inputs. However, due to the inherent nature of a PLLM, the same code token may be associated with different self-attention weights across different contexts. In our experiments, we observed a wide range of self-attention weights assigned to the same code token depending on *the contexts in which it appears*. Consequently, assigning *an average score* to each token for its importance is not appropriate.

Second, the way of DietCode using attention scores cannot directly reflect the importance of tokens in downstream tasks. Each input is represented by a special token labeled 'CLS'. This 'CLS' token's vector is computed based on the vectors of the constituent code tokens within that code snippet and description, determined by their self-attention scores. The vector of the 'CLS' token is fed into the final fully-connected layer for classification after the encoder. Consequently, tokens with higher CLS-attention scores hold greater importance for the classification task compared to individual token self-attention scores. However, DietCode assesses the self-attention scores of all input tokens for classification purposes, lacking the focus on the 'CLS' token, whose vector essentially encapsulates all necessary information for the classification.

Third, similarly, for code summarization, a transformer-based encoder-decoder architecture translates a given code snippet into a description. The self-attention mechanism within the encoder enables each token in the input to interact with

* Equal contribution.
† Corresponding authors.

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 1551–1567

all other tokens, effectively capturing dependencies and contextual information across different positions within the sequence, particularly adept at capturing long-range dependencies. DietCode only uses the attention scores of this encoder to signify the importance of code tokens and discarded these encoder-decoder attention scores.

Alternatively, SLIMCODE (Wang et al., 2024) is the state-of-the-art method based on human knowledge and uses a set of rules to determine the importance for different input code tokens. Specifically, it categorizes tokens into 8 priority levels based on the nature of the code tokens. For example, tokens in method signatures receive the highest priority, while symbol tokens (e.g., brackets, separators, and operators) have the lowest. However, performing better than DietCode, SLIMCODE still has the following issues (Wang et al., 2024). The manually selected priority levels with only 8 tiers result in a large number of tokens having the same priority, thus making token removal lack a solid basis. Secondly, the simplified code is fed into the model to complete downstream tasks, but model cognition can differ from human cognition. Thus, tokens considered important by human knowledge may not necessarily be considered important in model knowledge, leading to unexpected results.

In this paper, we posit that 1) model knowledge is more suitable for code simplification. 2) when leveraging attention scores to indicate the importance of code tokens in code simplification, one needs to consider their appearance **contexts**, the **CLS** and **encoder-decoder attention** are closely tied to downstream tasks, making their scores more appropriate for determining token importance in such tasks. We introduce a novel approach to code simplification, named LEANCODE. LEANCODE's overarching goal is to streamline computation time, including those for training and inference, in downstream tasks like code search and summarization, while preserving performance to the fullest extent possible. First, we advocate for the removal of a specific instance of token based on the attention score unique to that context of that occurrence, rather than relying on average scores across all inputs. We use the *statement type to represent the context* of that occurrence of t. Second, we propose integrating the self-attention scores of 'CLS' tokens for classification tasks, such as code search. Finally, for sequence-to-sequence tasks, we consider encoder-decoder attention scores to ascertain the significance of input tokens.

The contributions of this paper are as follows:

1. We carried out a systematic analysis of the significant tokens learned by both 'CLS' and encoder-decoder attentions, comparing them with the tokens learned only by encoder self-attention.

2. We present a new context-aware, code simplification, which is used in discriminative and sequence-to-sequence generation tasks.

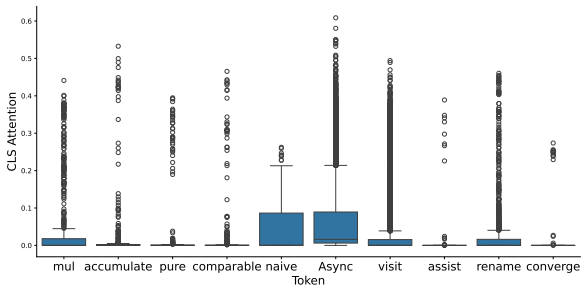
3. We evaluate LEANCODE in two downstream tasks and the results show its superiority over DIETCODE and SLIMCODE, with improvements of up to 60% and 16% for code search, and 29% and 27% for code summarization, respectively.

4. We evaluate LeanCode's cross-model transfer capability by feeding simplified code, generated using CodeT5, into the GPT-4o model to assess downstream task performance.

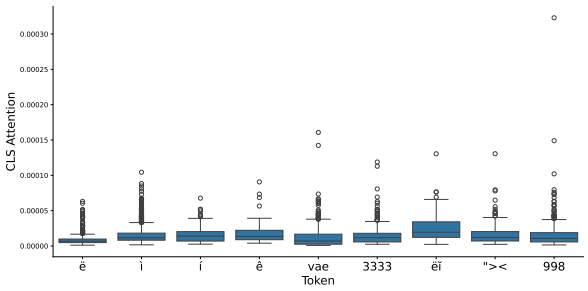
2 Preliminary Empirical Study

We conducted an empirical study on code search and summarization tasks to further investigate the significant tokens identified by the CLS and the Encoder-Decoder attentions. Our main focus is on the token level. The importance of statements and functions can be represented using tokens. We utilize the same models and datasets as DIETCODE and SLIMCODE. Specifically, for the code-search classification task, we use CodeBERT (an encoder-only model) and the encoder of CodeT5 (an encoder–decoder model), each augmented with an additional fully connected layer. For code summarization, we employ CodeBERT with a Transformer decoder as well as the complete CodeT5 model. For all experiments, we measure token importance using CLS (or Encoder-Decoder) attention scores from the final layer of the encoders of CodeBERT and CodeT5 (or the decoders of transformer and CodeT5), since both models compute scores across multiple layers. Token weights from the final layer provide the accurate representations of contextual relationships. Both tasks utilize the CodeSearchNet Corpus (Husain et al., 2019).

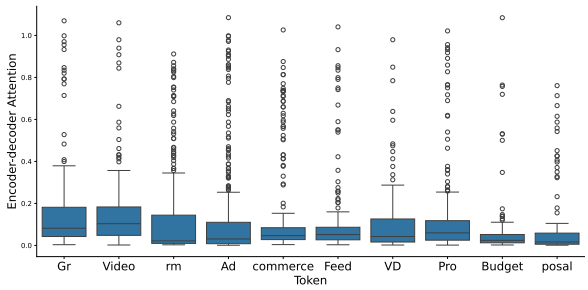
(RQ-1) What important tokens do CLS attentions emphasize on? For CodeBERT on code search and summarization, Fig. 1 show the top 10 and bottom 10 tokens with the highest and lowest variances in attention scores provided by CodeBERT. As seen, the tokens with the highest variance typically encompass richer semantic meanings, such as 'accumulate', 'pure', and 'commerce'. Conversely, tokens with the lowest variance tend to



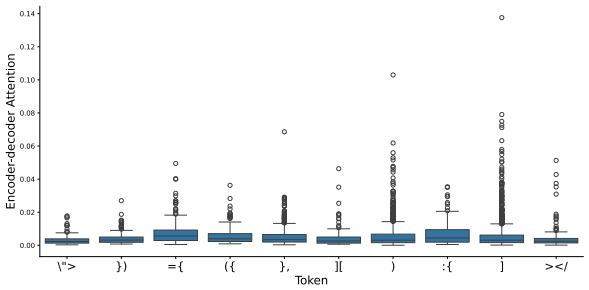
(a) Top 10 tokens with highest variance in CLS attention for code search.



(b) Bottom 10 tokens with lowest variance in CLS attention for code search.



(c) Top 10 tokens with highest variance in encoder-decoder attention for code summarization.



(d) Bottom 10 tokens with lowest variance in encoder-decoder attention for code summarization.

Figure 1: The variance of the top and bottom 10 tokens of CodeBERT for code search and summarization.

resemble simple symbols, including numbers, separators, and brackets. Upon examining tokens with high variance, we refer back to the original source code to analyze their positions. It is observed that *the tokens receive high attention scores when they appear in method signatures, function invocations, or as variables in return statements. Conversely, they attain low scores when used in conditions, expressions, and similar contexts*. Given the substantial variance observed, *relying solely on the average of all of its attention scores to gauge a token's importance across diverse statement types is not reasonable*. The attention scores of individual tokens are notably influenced by their positions in the code. Thus, categorizing statements and calculating the average attention score of each token in its contexts, i.e., distinct categories of statements, should be employed, named **context-aware, category-local attention average.** This aims to diminish such variance and bolster accuracy.

(RQ-2) What tokens encoder-decoder attentions emphasize on? Table 1 shows the average of the Encoder-Decoder attention scores of tokens based on statement classes (Zhang et al., 2022). Unlike CLS attention, each token in the input can have multiple Encoder-Decoder (EnDe) attention scores, i.e., for each generated token, the decoder will calculate an attention score for each token in the input. Thus, the largest attention score is selected as the attention score. The EnDe attention scores are generated in conjunction with the description. In the instances where the description contains intricate function details, these tokens garner high attention scores, facilitating the establishment of bi-modal mappings. For code search, the significance of detailed information within the code (e.g., `Throw` statements) is lower compared to the broader functional description (e.g., 'Method signature').

(RQ-3) Do the averages of self-attention scores reflect the CLS attentions and the Encoder-Decoder attentions? Our answer is 'No'. *The accumulated attention scores from the self-attention (as used in* DietCode*) is for pre-training and cannot reflect and substitute for those from the CLS and Encoder-Decoder attentions. i.e., the self-attention is used for pre-trained tasks and vectored general representations, not directly for downstream tasks*. For elaboration, these attention schemes are for different tasks. The self attention is for pre-training tasks, while CLS attention is for fine-tuning downstream discriminative tasks, and the Encoder-Decoder attention is for downstream sequence-to-sequence generation tasks. In fact, the encoders of CodeBERT and CodeT5 have been trained in multiple pre-trained tasks. Thus, the averages of self-attention scores cannot replace the CLS and Encoder-Decoder attentions. The latter attentions are directly applied to downstream tasks.

Table 1: (RQ-2) Statistics of encoder-decoder attention scores based on 0.16M training dataset. (Max/Min: the maximum/minimum of encoder-decoder attention scores in each category; Global/Global_variance: the average/variance of the global attention scores of tokens for each category; Category-local/Local_variance: the averages/variance of category-local attention scores.)

Category	Max	Min	**Global**	**Global_variance**	**Category-local**	**Local_variance**
Annotation	7.94	0.32	2.61	13.76	1.55	0.09
Arithmetic	37.44	0.07	2.69	15.41	2.30	2.52
Variable Declaration	65.54	0.09	2.87	15.63	2.69	7.97
Function Invocation	63.97	0.01	2.86	15.94	2.80	8.54
Return	55.23	0.10	3.08	17.61	4.76	16.02
Switch	30.03	0.07	2.71	16.36	2.41	2.63
Break	28.02	0.04	2.64	16.43	2.67	1.21
Setter	69.06	0.03	2.85	17.25	2.33	5.10
Synchronized	78.09	0.04	2.84	17.08	3.11	3.03
Try	78.27	0.03	2.82	17.31	2.46	2.69
Catch	34.99	0.07	3.01	19.80	2.44	4.18
Method Signature	91.69	0.14	3.29	18.21	5.91	30.92
Finally	10.49	0.74	2.38	7.78	2.99	1.74
Getter	68.49	0.03	2.88	16.59	2.58	6.42
Throw	87.67	0.06	2.80	16.04	3.10	8.13
Case	23.25	0.03	2.75	16.11	1.80	1.55
While	67.68	0.04	2.70	15.52	2.41	3.14
Continue	9.85	0.27	2.49	12.64	1.73	0.37
If Condition	57.88	0.05	2.84	15.84	2.50	5.97
For	60.62	0.03	2.91	17.21	2.99	6.89
Logging	65.63	0.04	2.77	15.53	2.89	8.42

3 LEANCODE: Code Simplification

3.1 Code Simplification Problem Formulation

Given a dataset $D = \{d_1, ..., d_m\}$ with m snippets. Each snippet d_j contains a sequence of n_j tokens. Thus, each code snippet d_j can be denoted as $d_j = \{t_1, \cdots, t_{n_j}\}$ and the index of each token records its position. w_i denotes the importance of each token t_i and x_i is a binary indicator showing whether t_i should be removed or not. With the simplification ratio $SimplifiedRatio$, the total number of tokens to be removed for d_j is $\mathcal{X} = SimplifiedRatio \times n_j$. Now, we formulate code simplification as the combinatorial optimization problem as following:

$$\text{minimize} \sum_{i=1}^{n_j} w_i x_i, \text{ such that} \sum_{i=1}^{n_j} x_i = \mathcal{X}. \quad (1)$$

$x_i \in \{0, 1\}$, code simplification aims to minimize the weighted sum of $w_i x_i$ that satisfies the number of tokens to be removed for each d_j.

3.2 LEANCODE Algorithm

3.2.1 Computation

As mentioned, there are three methods to measure token importance through attention scores: *dynamic, global, and category-local methods*. Regarding the dynamic method, the CLS and Encoder-Decoder attention scores of the same token will be different for different inputs, which are dynamically generated and can reflect the importance of corresponding tokens in context. However, it is inefficient and impractical to assign a dynamic attention score to each input token in the test dataset using the models. Calculating the dynamic attention scores requires multiple transformer blocks (for example, CodeBERT and the encoder of CodeT5 have a stack of 12 transformer blocks), making it time-consuming. Moreover, once we have obtained the dynamic attention scores, the downstream tasks are nearly complete. Thus, it does not make sense and is unnecessary to reduce the code and redo the downstream tasks.

Regarding the global attention average of each token in the training dataset, DietCode uses it to replace the dynamic attention score in testing dataset, which is computed in Equation (2):

$$\mu_t = \frac{\sum_{j=1}^{m} \sum_{t \in d'_j} s_t}{n_t} \quad (2)$$

where $t \in d'_j$ means that a token t is in d'_j in a training dataset D' and s_t is the common self-attention score. n_t is the number of the occurrence of token t in the training dataset D'.

Our empirical study reveals that significant tokens often have high variances in global attention averages. As a result, we propose the category-local attention average for each token, defined as

$$\mu_t^c = \frac{\sum_{j=1}^{m} \sum_{t \in p_k, p_k \in d'_j, L(p_k) \in c} s_t}{n_t^c}. \quad (3)$$

In this equation, p_k is a statement of a code snippet d'_j. $L(p_k)$ is the label (category) of the statement

Algorithm 1 LEANCODE: Code Simplification Algorithm

INPUT: A dataset $D = \{d_1, \ldots, d_m\}$, token scores S, $SimplifiedRatio$
OUTPUT: A simplified code dataset D'
PROCEDURE:

1: $D^c \leftarrow D$
2: **for** $j = 1$ **to** m **do**
3: $\quad removedTokens \leftarrow \{\}$
4: $\quad \mathcal{X} \leftarrow SimplifiedRatio \times n_j$
5: $\quad removedTokenNum \leftarrow 0$
6: $\quad$ **while** $removedTokenNum < \mathcal{X}$ **do**
7: $\quad\quad$ Add {index:token with lowest s_t} ($\in d_j^c$, $\notin removedTokens$) into $removedTokens$
8: $\quad\quad removedTokenNum$ updates
9: $\quad d_j^c = d_j^c / removedTokens[1 : \mathcal{X}]$
10: **return** D^c

p_k. n_t^c refers to the number of occurrences of token t in the statements belonging to the category $c \in C$. Finally, s_t can be CLS attention or Encoder-Decoder attention scores. The definitions of those scores are provided in Section B.

3.2.2 Removal Algorithm

Algorithm 1 displays LEANCODE algorithm. Unlike DietCode's removing the less critical statements and proceeding to remove less important tokens from other statements, it exclusively focuses on token-level removal, without initially discarding entire statements. Deleting entire statements would result in the loss of important tokens. Our Algorithm (1) initializes a copy of the original dataset D as the returned simplified dataset (line 1). Next, it iteratively removes the tokens of each snippet in D^c one by one (lines 2–9) based on their attention scores stored in the dictionary $S = \{t, c, \mu_t^c\}$ where c is the category of the statement that token t belongs to, μ_t^c is the category-local attention average of the token. At line 3, `removedTokens` records the pair of the index and the respective token with current lowest score ({index:token}) in d_j^c at each turn. The number of removed tokens is set (line 4). In line 5, `removedTokenNum` is the number of currently selected tokens to be removed. At each iteration, LEANCODE repeatedly selects the remaining token (not in `removedTokens`) with the lowest score in d_j^c until the number of removed tokens is reached (lines 6–8). Finally, our algorithm returns the simplified dataset D'.

4 Empirical Evaluation

We evaluate our LEANCODE on different code tasks using three PLLMs and GPT-4o.

Downstream Tasks: We choose code search and summarization as the tasks (also used in DIETCODE and SLIMCODE), which are commonly used in evaluating LLMs in text and code analysis (Ahmed and Devanbu, 2022; Feng et al., 2020a; Jiang et al., 2021; Wang et al., 2021a; Niu et al., 2022; Liu et al., 2019). The goal of code search is to find relevant code snippets from a codebase given a query and code summarization is to generate a natural language summary for a given code.

Models Under Study: We opted for 3 popular models, CodeBERT (Feng et al., 2020a), CodeT5 (Wang et al., 2021b) and GPT-4o (Islam and Moushi, 2024). Unlike DIETCODE (calculating attention scores based on all layers), for 'CLS' and Encoder-Decoder attention scores, we obtain them from the last encoder and decoder blocks of the respective model.

CodeBERT-based code search and summarization: We added a fully connected layer on top of the CodeBERT model for binary classification to perform code search. As in DietCode, we added a Transformer decoder for code summarization.

CodeT5-based code search and summarization: For code search, its encoder is separately extracted and joint with a fully connected layer for the classification task. We use CodeT5 directly for code summarization.

GPT4-based code search and summarization: Since GPT-4o is accessible only through programming APIs, we cannot directly access its model. We are limited to using prompts to obtain classification and summrization results and corresponding analyses from GPT-4o in a predefined format.

Baselines: We chose **DIETCODE** (Zhang et al., 2022) and **SLIMCODE** (Wang et al., 2024), the SOTA code simplification methods. DietCode (Zhang et al., 2022) is based on token weights learned by models and SlimCode (Wang et al., 2024) is based on the nature of tokens.

Datasets: We used code search and summarization datasets from CodeBERT (Feng et al., 2020a) (Details in Table 9 in Appendix). These are the extensions of CodeSearchNet (Husain et al., 2019), which is a collection of datasets and benchmarks for code retrieval using texts. It consists of +2 millions pairs of (comment, code) that were extracted from Github, covering six languages (Python, PHP,

Go, Java, JavaScript, and Ruby). Since DietCode (Zhang et al., 2022) reported similar trends for different languages, we conducted experiments only on Java.

Metrics: We use the simplification ratio to measure the degree of simplification of a code snippet. Given a code snippet $Code$ and its simplified one $Scode$, $SimplifiedRatio = \frac{|Code|-|Scode|}{|Code|} \times 100$. $|Code|$ and $|Scode|$ are the numbers of tokens in $Code$ and $Scode$. The efficiency of simplified code is measured by the ***time cost*** it takes for model inference. We use ***BLEU-4*** score and ***MRR*** (Mean Reciprocal Rank) for code summarization and search, respectively. For code search by GPT-4o, we use Precision instead of MRR due to the latter's high computational requirements. *Precision* is highly correlated with MRR in measuring model effectiveness. Similar to SLIMCODE, we randomly replace the code description in 400 sample pairs of texts-code and check if the replacement content matches the code part. The dataset consists of an equal number of matching and non-matching samples.

Implementation: DietCode is realized using the code in (Zhang et al., 2022). We set up CodeBERT and CodeT5 with default hyper-parameters. For optimization, they used Adam optimizer with learning rates of 1×10^{-5} and 5×10^{-5} for downstream tasks. We used a server with 2 CPUs of Intel(R) Xeon(R) Golden 2.40GHz and 2 Nvidia A100s.

4.1 Effectiveness of LEANCODE

Tables 2 and 3 present the results. We first use the training dataset (complete code snippets) to train the relevant models, and then use the testing dataset (complete code snippets) to perform inference on a model to obtain the 'Base' results. Next, we apply code simplification to the samples in the testing dataset using DietCode, SlimCode, and LEANCODE according to the simplifiedRatio. Finally, we input the simplified code into the trained models to perform inference, obtaining results for the cases of the removal percentages of 10%-50%. As seen in columns 'R-M' and 'R-B', the performance of both methods declines in downstream tasks as the $SimplifiedRatio$ gradually increases. However, even when 50% of the code is removed, the decline at most in the effect of LEANCODE for code search and code summarization are only *5.48%* and *11.01%*. Meanwhile, LEANCODE enhances DIETCODE (SLIMCODE) up to 60.37% (15.82%) and 14.12%(6.28%) in terms of MRR and BLEU scores with CodeBERT. Similarly, it improves by 25.84% (10.14%) and 29.36% (27.04%) in terms of MRR and BLEU scores with CodeT5.

We observe two interesting points: 1) Intuitively, one would expect the performance of the removed code for downstream tasks to be worse than the original code. However, LEANCODE and SLIMCODE can achieve better results on code search. 2) Compared with LEANCODE, SLIMCODE experiences a sharp decline in performance when the $simplifiedRatio$ exceeds 30%.

For the 1^{st}observation, if the code contains more valuable information, the downstream tasks will have better results. Not all code snippets can be fully inserted into models, as the maximum lengths for original codes are limited to 512 tokens. Thus, even if some tokens are removed, tokens towards the end of the code snippets may still be included in the input. If the newly input tokens provide more information than the removed tokens, then the above observation may occur. Specifically, LEANCODE often removes low-quality tokens with lower attention scores (e.g., symbol-like tokens) while allowing high-quality tokens to enter the model input.

For the 2^{nd} observation, the main reason lies in the deletion of increasingly important tokens as the process progresses. At this stage, a model's ability to precisely discern token importance becomes critical for maintaining the task performance. SLIMCODE, with only 8 levels of token importance, struggles to differentiate between crucial tokens when $simplifiedRatio$ surpasses 30%.

Figure 2 presents an example of our LEANCODE. Subfigure (a) shows the original code, which computes a Bessel function using a recurrence formula. Subfigures (b) and (c) display the results of LEANCODE after 30% token removal for code search and summarization, respectively. Both approaches preserved critical elements, such as function signatures and return statements. For code search, structural details like loops, arithmetic operations, and variable names were retained to support precise keyword matching. For summarization, non-essential details such as loop bodies, variable declarations, and calculations were removed, while return statements were emphasized. This demonstrates how return statements play a key role in capturing code intent, enabling concise and clear summaries.

4.2 Efficiency of LEANCODE

Tables 2 and 3 show the results on the positive correlations between the simplified ratio and inference time. For instance, for code summarization using

Table 2: Results of Code Search for DIETCODE , SlimCode and LEANCODE. (10%-50%: removing 10%-50% tokens, R-M: Reduced MRR, Time: Inference time, R-T: Reduced Inference time)

Ratio	DIETCODE				SlimCode				LEANCODE				Inference			
	CodeBERT		CodeT5		CodeBERT		CodeT5		CodeBERT		CodeT5		CodeBERT		CodeT5	
	MRR	R-M	MRR	R-M	MRR	R-M	MRR	R-M	MRR	R-M	MRR	R-M	Time	R-T	Time	R-T
Base	0.726	—	0.747	—	0.726	—	0.747	—	0.726	—	0.747	—	41m	—	40m	—
10%	0.663	8.67%↓	0.699	6.42%↓	0.731	0.68%↑	0.738	1.2%↓	0.728	0.27%↑	0.743	0.53%↓	38m	7.31%↓	36m	10%↓
20%	0.598	17.63%↓	0.669	10.44%↓	0.726	0.00%↓	0.733	1.87%↓	0.719	0.96%↓	0.736	1.47%↓	35m	14.63%↓	33m	17.5%↓
30%	0.529	27.13%↓	0.624	16.46%↓	0.70	3.58%↓	0.723	3.21%↓	0.716	1.37%↓	0.724	3.07%↓	32m	21.95%↓	31m	22.5%↓
40%	0.502	30.85%↓	0.602	19.41%↓	0.632	12.94%↓	0.679	9.1%↓	0.697	3.99%↓	0.714	4.41%↓	29m	29.27%↓	28m	30%↓
50%	0.429	40.90%↓	0.561	24.89%↓	0.594	18.18%↓	0.641	14.19%↓	0.688	5.23%↓	0.706	5.48%↓	26m	36.59%↓	25m	37.5%↓

Table 3: Results of Code Summarization for DIETCODE , SlimCode and LEANCODE. (10%-50%: removing 10%-50% tokens, R-B: Reduced BLEU, Time: Inference time, R-T: Reduced Inference time)

Ratio	DIETCODE				SlimCode				LEANCODE				Inference			
	CodeBERT		CodeT5		CodeBERT		CodeT5		CodeBERT		CodeT5		CodeBERT		CodeT5	
	BLUE	R-B	BLUE	R-B	BLUE	R-B	BLUE	R-B	BLUE	R-B	BLUE	R-B	Time	R-T	Time	R-T
Base	18.25	—	20.55	—	18.25	—	20.55	—	18.25	—	20.55	—	29m	—	22m	—
10%	16.44	9.91%↓	17.27	15.96%↓	17.86	2.13%↓	20.01	2.62%↓	18.08	0.93%↓	20.32	1.11%↓	27m	6.89%↓	20m	9.09%↓
20%	15.68	14.08%↓	16.48	19.80%↓	17.35	4.93%↓	18.68	4.23%↓	17.73	2.84%↓	20.18	1.80%↓	26m	10.34%↓	18m	18.18%↓
30%	15.05	17.53%↓	15.74	23.40%↓	16.8	7.94%↓	18.74	8.80%↓	17.23	5.58%↓	19.82	3.55%↓	24m	17.24%↓	16m	27.27%↓
40%	14.66	19.67%↓	15.11	26.47%↓	15.95	12.60%↓	16.35	20.43%↓	16.71	8.43%↓	19.27	6.22%↓	23m	20.68%↓	15m	31.81%↓
50%	14.23	22.02%↓	14.27	30.55%↓	15.28	16.27%↓	14.53	29.29%↓	16.24	11.01%↓	18.46	10.17%↓	23m	24.13%↓	13m	40.90%↓

CodeT5, *the inference time is reduced by 40.9% when the simplification ratio is set to 50%.*

For both tasks, as the simplified ratio increases, *the inference time follows a near-linear descent* on both of models. Specifically, for code search, the ratio of $SimplifiedRatio$ and reduce time is about 0.7, meanwhile, for code summarization, the ratio is about 0.5 on CodeBERT and 0.75 on CodeT5, respectively. The ratio on CodeBERT for code summarization is relative lower than others since directly concatenating CodeBERT with a Transformer decoder would result in the model lacking some optimization for inference.

In addition to inference time, each method requires training and pruning time. DietCode and LeanCode also need extra computation for per-token attention scores. While all methods share identical training times due to using the same training dataset and model, calculating token attention scores adds about 5% to the total training time. For example, in code summarization, standard training across 8 epochs takes 37.5 minutes per epoch, totaling 300 minutes. When attention scores are collected only during the final epoch, the first seven epochs remain unchanged, but the last epoch increases to 53 minutes. This results in a total time of 315.5 minutes,i.e., representing a 5% increase in training time.

We conducted additional pruning experiments for LeanCode and obtained pruning time results for SlimCode and DietCode from SlimCode's paper. The results are shown in Table 4. From these results, we observed: 1) DietCode's pruning time is significantly higher than SlimCode and LeanCode due to its two-step process—selecting high-quality statements first, then removing tokens from the remaining ones. This approach leads to complex knapsack optimization problems, resulting in a lower reduction ratio and increased pruning time; 2) LeanCode's pruning time is approximately 2–4 times longer than SlimCode's, primarily due to LeanCode's more complex pruning process, which includes additional steps like tokenizing and statement class matching; 3) The pruning times for LeanCode and SlimCode remain within a comparable range.

Table 4: Pruning times for DietCode(**DC**), SlimCode(**SC**), and LeanCode(**LC**) on code search and code summarization training datasets(10%-50% removal for each code snippet)

Ratio	Code Search			Code Summarization		
	DC	SC	LC	DC	SC	LC
10%	9h24m	17m	46m33s	1h40m	45s	3m32s
20%	8h28m	17m	46m39s	1h30m	53s	3m37s
30%	7h37m	20m	47m15s	1h19m	59s	3m41s
40%	6h45m	21m	47m35s	1h11m	66s	3m45s
50%	5h59m	21m	47m43s	1h02m	69s	3m50s

4.3 Model Transferability

We assess whether code simplified by LEANCODE using one model can be applied to another while maintaining previous results and conclusions for a given downstream task. We replicate the analysis procedures in the Section 4.1 with GPT-4o. As GPT-4o only provides access via APIs, we used

```
1. public static double Y ( int n , double x ) {
2.   double by , bym , byp , tox ;
3.   if ( n == 0 ) return Y0 ( x ) ;
4.   if ( n == 1 ) return Y ( x ) ;
5.   tox = 2.0 / x ;
6.   by = Y( x ) ;
7.   bym = Y0 ( x ) ;
8.   for ( int j = 1 ; j < n ; j ++ ) {
9.     byp = j * tox * by - bym ;
10.    bym = by ;
11.    by = byp ;
12.  }
13.  return by ;
14. }
```

(a) original source code

```
1. public static double Y ( int n , double x ) {
2.   double by bym byp to ;
3.   ( ) return Y0 ( ) ;
4.   ( ) return Y ( ) ;
5.   tox = 2 / x ;
6.   by Y ( ) ;
7.   bym Y0 ( ) ;
8.   for ( ) byp = j * tox * by - bym ;
9.     bym = by ;
10.    by = byp ;
11.  }
12.  return by ;
13. }
```

(b) remove 30% tokens with LeanCode for code search

```
1. public static double Y ( int n , double x ) {
2.   , , byp , ;
3.   if ( n ) return Y0 ( x ) ;
4.   if ( n ) return Y ( x ) ;
5.   = 2 x ;
6.   by = Y ( x ) ;
7.   bym = Y0 ( x ) ;
8.   ( ; < n ; )byp = by bym ;
9.     bym = by ;
10.    by = byp ;
11.  }
12.  return by ;
13. }
```

(c) remove 30% tokens with LeanCode for code summarization

Figure 2: The example of LEANCODE for code simplification.

Table 5: Results of removal methods on GPT-4 for Code Search (IT: Input Tokens, R-IT: Reduced Input Tokens (%), OT:Output Tokens, R-OT:Reduced Output Tokens (%); TT:Total Tokens, R-P:Reduced Precision (%)

Removal method	IT	R-IT	OT	R-OT	TT	R-TT	Precision	R-P
Base	102385	—	24535	—	127920	—	0.82	—
DietCode (10%)	98105	4.18%↓	24619	0.34%↑	122724	4.06%↓	0.776	5.37%↓
DietCode (20%)	93268	8.23%↓	24508	0.11%↓	117776	7.92%↓	0.813	0.85%↓
DietCode (30%)	86672	14.61%↓	24211	1.32%↓	110883	13.32%↓	0.775	5.49%↓
DietCode (40%)	80196	20.98%↓	24536	0.00%↑	104732	18.14%↓	0.78	4.88%↓
DietCode (50%)	73759	27.76%↓	24169	1.49%↓	97928	23.44%↓	0.776	5.37%↓
SlimCode (10%)	96845	5.41%↓	25343	3.29%↑	122188	4.48%↓	0.79	3.66%↓
SlimCode (20%)	91736	10.40%↓	25217	2.99%↑	116953	8.57%↓	0.808	1.46%↓
SlimCode (30%)	85778	16.22%↓	24779	0.99%↑	110557	13.57%↓	0.789	3.78%↓
SlimCode (40%)	79686	22.17%↓	24734	0.81%↑	104420	18.37%↓	0.796	2.93%↓
SlimCode (50%)	73215	28.49%↓	24548	0.05%↑	97763	23.57%↓	0.763	6.95%↓
LeanCode (10%)	96978	5.28%↓	25843	5.33%↑	122821	3.99%↓	0.795	3.05%↓
LeanCode (20%)	91173	10.95%↓	25043	2.07%↑	116216	9.15%↓	0.798	2.68%↓
LeanCode (30%)	85202	16.78%↓	25454	3.74%↑	110656	13.50%↓	0.828	0.49%↑
LeanCode (40%)	79102	22.74%↓	25162	2.55%↑	104264	18.49%↓	0.793	3.54%↓
LeanCode (50%)	72887	28.81%↓	24741	0.84%↑	97628	23.68%↓	0.81	1.22%↓

a program to interact with GPT-4o. We used the following prompts: 1) for code search, *"Please check whether the incomplete code snippet is semantically consistent with the given text. Please analyze step-by-step first, and then answer in the following format."*, and 2) for code summarization, *"Write a short sentence to describe the function of the incomplete code snippet. Answer in the following format."* Finally, we analyzed the results of GPT-4o to calculate the number of total tokens, precision, and BLEU-4.

Tables 5 and 6 present the results of the code simplification methods. We use the total number of tokens (the input and output tokens), instead of the prediction time for two reasons: 1) the fee charged for the usage of computing resource of GPT-4o is based on the total number of tokens (OPENAI-Pricing); 2) GPT-4 only offers API interfaces, making it difficult to accurately measure time due to multiple influencing factors. It's worth noting that the input tokens include not only the code snippet but also the prompts. From the results, we made the following empirical observations:

a) Saving Computational Resources. As the simplified ratio increases, the total number of tokens decrease for all methods on both tasks in GPT-4o. This finding substantiates that code simplification leads to a reduction in resources required for Prompt-based LLM. This can be due the fact that GPT-4o requires a significant amount of analytical content by following a sequence of statements through reasoning, as opposed to solely producing binary responses. It shows that in the absence of generating analytical texts, the classification performance of GPT-4o is notably deficient (OpenAI, 2023). However, the total number of tokens decreases since the reduction in input tokens outweighs the increase in output tokens.

b) Performance on Code Summarization. Firstly, LEANCODE still slightly outperforms the baselines. We observe that for all methods, the BLEU-4 scores in Table 6 are approximately half of those in Ta-

Table 6: Results of removal methods on GPT-4 for Code Summarization (IT: Input Tokens, R-IT: Reduced Input Tokens (%), OT: Output Tokens, R-OT: Reduced Output Tokens (%); TT: Total Tokens, R-B: Reduced BLEU (%))

Removal method	IT	R-IT	OT	R-OT	TT	R-TT	BLEU	R-B
Base	78246	—	7668	—	85914	—	10.59	—
DietCode (10%)	75217	3.87%↓	7340	4.28%↓	82557	3.90%↓	10.80	1.98%↑
DietCode (20%)	72245	7.67%↓	7583	1.11%↓	79828	7.08%↓	10.21	3.59%↓
DietCode (30%)	66844	14.58%↓	7846	232%↑	74690	13.06%↓	10.12	4.44%↓
DietCode (40%)	61534	21.36%↓	8015	4.52%↑	69549	19.06%↓	9.95	6.04%↓
DietCode (50%)	56162	28.21%↓	7748	1.04%↑	63910	25.59%↓	9.69	8.50%↓
SlimCode (10%)	74912	4.26%↓	7426	3.16%↓	82338	4.16%↓	10.71	1.14%↑
SlimCode (20%)	70734	9.96%↓	7715	0.61%↑	78449	8.69%↓	10.89	2.83%↑
SlimCode (30%)	65648	16.10%↓	7633	0.46%↓	73281	14.70%↓	10.71	1.14%↑
SlimCode (40%)	60585	22.57%↓	7773	1.37%↑	68358	20.43%↓	10.62	0.28%↑
SlimCode (50%)	55335	29.28%↓	7496	2.24%↓	62831	26.86%↓	10.60	0.09%↑
LeanCode (10%)	74938	4.23%↓	7487	2.36%↓	82425	4.06%↓	11.11	4.91%↑
LeanCode (20%)	70207	10.28%↓	8278	7.95%↑	78485	8.66%↓	10.69	1.09%↑
LeanCode (30%)	65322	16.51%↓	7607	0.80%↓	72929	15.11%↓	10.77	1.70%↑
LeanCode (40%)	60296	22.93%↓	8163	6.45%↑	68459	20.32%↓	10.90	2.93%↑
LeanCode (50%)	55261	29.37%↓	8442	10.08%↑	63703	25.86%↓	10.70	1.04%↑

ble 3. This indicates that the descriptions from GPT-4o deviate more largely from the ground truth compared to those from CodeT5. GPT-4o lacks the capability to produce descriptions closely resembling the ground truth, as it did not undergo fine-tuning. The overlap of words between the generated description and the ground truth is considerably lower when compared to CodeT5.

c) Performance on Code Search. From Table 5, the base GPT-4o yielded good results, even without model fine-tuning. Interestingly, as the simplified ratio increases, Precision scores do not show a consistent decline. Moreover, all methods produced closely clustered results, with LEANCODE maintaining a slight edge. A plausible explanation is that GPT-4o was trained on a similar dataset. Consequently, GPT-4o can identify crucial tokens for generating judgments, resulting in better inference.

5 Related Work

Pre-trained Models. Pre-trained models have significantly advanced code models (Feng et al., 2020b; Guo et al., 2021; Karampatsis and Sutton, 2020; Guo et al., 2022). They excel in various tasks such as code generation (Wang et al., 2021b; Clement et al., 2020; Wang et al., 2023b), defect detection (Wang et al., 2023a), code summarization (Ahmed and Devanbu, 2022; Jiang et al., 2021), and code search (Wang et al., 2021a; Niu et al., 2022). Researchers have investigated pre-trained models for code understanding (Ahmad et al., 2021; Mastropaolo et al., 2021). For instance, Karmakar and Robbes (Karmakar and Robbes, 2021) conducted four probing tasks on pre-trained models to assess their ability to learn various aspects of source code. Wan *et al.* (Wan et al., 2022) reported that Transformer attention mechanisms can capture high-level structures within source code. Moreover, Autofocus (Bui et al., 2019) is a method to determine the most relevant code by measuring statement relevance using attention weights from a GGNN (Allamanis et al., 2018).

Program Simplification. SIVAND (Rabin et al., 2021) and P2IM (Suneja et al., 2021) typically build upon the delta debugging prototype (Zeller and Hildebrandt, 2002), which involves treating a code snippet and an auxiliary model (e.g., code2vec) as inputs. The model segments the code snippet into fragments and use each as inputs. If a fragment achieves a satisfactory score, it is further divided. This process continues until the subset's performance fails to meet the desired score, resulting in the smallest snippet that satisfies the goal.

6 Conclusion

In this study, we introduce LEANCODE, a novel code simplification approach that harnesses code contexts to utilize attention scores of pre-trained models for representing the importance levels of each token of input. We advocate for the selective removal of tokens based on the average **context-aware** attention scores. rather than relying on average scores across all inputs. We evaluated LEANCODE in code search and code summarization tasks, experimental results show its superiority over the SOTA DietCode and SlimCode, achieving improvements of up to 60% and 16% for code search and 29% and 27% for code summarization.

7 Limitations

LEANCODE still has the following limitations"

Programming Language: One limitation of our current model is its exclusive application to Java, which restricts its use and effectiveness in other programming languages. Although related literature reports similar effects in other languages, this limitation highlights the need for future work to expand the model's capabilities across multiple programming languages.

External Validity: Our experiments were on three models (CodeBERT, CodeT5, GPT-4o), and two tasks of code search (text-to-code) and code summarization (code-to-text). Despite the consistent findings, for generalizability, future experiments are needed on a wider variety of models with other paradigms and in other code-related tasks. Our dataset might not be representative. However, for a fair comparison, we used the same datasets and two tasks as in DIETCODE and SLIMCODE.

Internal Validity: When measuring the time, we acknowledge that there might be other external factors involving hardware (e.g., GPUs), operating system delays, etc. However, inference time for the models, CodeBERT and CodeT5 is stable in the same controlled environment.

Acknowledgments

The third author, Tien N. Nguyen, was supported in part by the US National Science Foundation (NSF) grant CNS-2120386 and the National Security Agency (NSA) grant NCAE-C-002-2021.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*.

Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1443–1455, New York, NY, USA. Association for Computing Machinery.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2019. Autofocus: interpreting attention-based neural networks by code perturbation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 38–41. IEEE.

ChatGPT. OpenAI. https://openai.com/.

Colin B Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. Pymt5: multi-mode translation of natural language and python code with transformers. *arXiv preprint arXiv:2010.03150*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020a. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020b. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Raisa Islam and Owana Marzia Moushi. 2024. Gpt-4o: The cutting-edge advancement in multimodal llm. *Authorea Preprints*.

Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. Treebert: A tree-based pre-trained model for programming language. In *Uncertainty in Artificial Intelligence*, pages 54–63. PMLR.

Rafael-Michael Karampatsis and Charles Sutton. 2020. Scelmo: Source code embeddings from language models. *arXiv preprint arXiv:2004.13214*.

Anjan Karmakar and Romain Robbes. 2021. What do pre-trained code models know about code? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1332–1336. IEEE.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 336–347.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint*.

Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng. 2022. Deep learning meets software engineering: A survey on pre-trained models of source code. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 5546–5555. ijcai.org.

OpenAI. 2023. GPT-4 technical report. *CoRR*, abs/2303.08774.

OPENAI-Pricing. Openai-pricing. https://openai.com/api/pricing/.

Md Rafiqul Islam Rabin, Vincent J. Hellendoorn, and Mohammad Amin Alipour. 2021. Understanding neural code intelligence through program simplification. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 441–452, New York, NY, USA. Association for Computing Machinery.

Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim A. Laredo, and Alessandro Morari. 2021. Probing model signal-awareness via prediction-preserving input minimization. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 945–955, New York, NY, USA. Association for Computing Machinery.

Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What do they capture? a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2377–2388, New York, NY, USA. Association for Computing Machinery.

Wenbo Wang, Tien N. Nguyen, Shaohua Wang, Yi Li, Jiyuan Zhang, and Aashish Yadavally. 2023a. Deepvd: Toward class-separation features for neural network vulnerability detection. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 2249–2261. IEEE Press.

Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021a. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556*.

Yan Wang, Xiaoning Li, Tien N Nguyen, Shaohua Wang, Chao Ni, and Ling Ding. 2024. Natural is the best: Model-agnostic code simplification for pre-trained large language models. *Proceedings of the ACM on Software Engineering*, 1(FSE):586–608.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023b. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021b. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

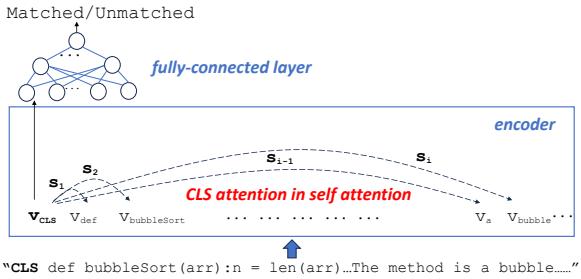
Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200.

Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Diet code is healthy: Simplifying programs for pre-trained models of code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 1073–1084, New York, NY, USA. Association for Computing Machinery.

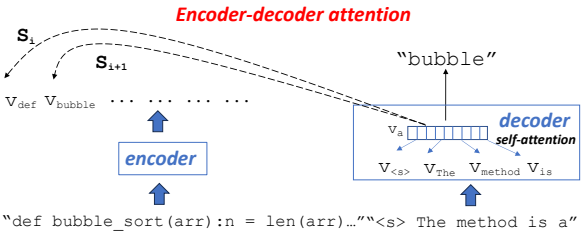
Appendix

A DietCode discarded encoder-decoder attention scores

For code summarization like task, DietCode still uses the attention scores of this encoder to signify the importance of code tokens. However, it overlooks a crucial aspect: *the encoder-decoder attention*. Fig. 3b illustrates this encoder-decoder attention mechanism during the generation for the code in Fig. 4. As shown, the vector v_{bubble} should receive more attention from the decoder, and the product of v_{bubble} and normalized attention score s_{i+1} is integrated into the final vector v_a to generate the token 'bubble' (after the previous tokens



(a) Example of the CLS-attention. It is the self-attention of the 'CLS' token that is calculated by the weighted sum of attention over all tokens in the input sequence to derive the representation of the CLS token. The input is the concatenation of a code snippet and a description, the output is 'matched' or 'unmatched', v is the vector of each token and ss are the attention scores of 'CLS' token on other tokens based on Equation 5.



(b) Example of Encoder-Decoder attention. The attention is computed by paying attention to the encoder's output while also maintaining self-attention within the decoder layers to generate context-aware representations for the target sequence. The vector 'v_a' first performs self-attention with the generated vectors $v_{<s>}$, v_{The}, v_{method}, v_{is} then absorbs the vectors of input tokens through encoder-decoder's weights to generate next token "bubble".

Figure 3: CLS and Encoder-Decoder attention scores based on the example in Fig. 4.

have already been generated). Thus, input vectors with higher decoder-attention scores typically hold greater importance for sequence-to-sequence tasks and can effectively discern the tokens' significance. However, DietCode uses only self-attention scores and discarded these encoder-decoder attention scores.

B An Example of CLS and encoder-decoder attentions

Let us use an example to illustrate the problem and to motivate our work (Sections 2 and 3).

The state-of-the-art DIETCODE (Zhang et al., 2022) uses the accumulated attention scores to indicate the importance of each input token for downstream tasks. For classification tasks, DIETCODE uses a pre-trained encoder (e.g., CodeBERT or CodeT5) in conjunction with a fully-connected layer for code search. For generation tasks, e.g., code summarization, it uses a sequence-to-sequence structure that combines either the encoder of CodeBERT or CodeT5 with a Transformer decoder or CodeT5 decoder. For both tasks, the calculation of the importance of a token is only based on the self-attention scores of the encoder in DIETCODE. Specifically, the accumulated self attention score of an input token is calculated in Equation 4. Given an input sequence $X = [x_1, x_2, \ldots, x_n]$ where x_i is a d-dimensional vector. First, Query ($Q = [q_1, q_2, \ldots, q_n]$), Key ($K = [k_1, k_2, \ldots, k_n]$), and Value ($V = [v_1, v_2, \ldots, v_n]$) vectors for a token are generated via three linear mappings. These mappings are implemented by the weight matrices W^Q, W^K, and W^V, which are learnable parameters. For any token x_i, its key vector is dot-producted with every query vector in the sequence, yielding an accumulated attention score that is scaled down by the square root of d as shown in Equation (4).

$$s_i = \frac{\sum_{j=1}^{n} q_j \cdot k_i}{\sqrt{d}}. \tag{4}$$

Each accumulated attention score s_i measures how the corresponding token gains attention from other tokens in the input sequence. The example of the accumulated self-attention based for the example in Fig. 4 is shown in Fig. 6.

For the classification tasks, e.g., code search, *only the 'CLS' token is sent into the fully-connected layer for label prediction* as shown in Fig. 3a. Thus, **the tokens to which the 'CLS' token pays attention could play more crucial roles in the classification task than other tokens**. For the CLS attention score s_i for corresponding token, it can be calculated in Equation 5:

$$s_i = \frac{q_{cls} \cdot k_i}{\sqrt{d}}, (1 \leq i \leq n) \tag{5}$$

In code search, the model assesses the correspondence between the code and its accompanying description. Via pre-training or fine-tuning, the model acquires an understanding of the interrelation among tokens in bi-modal data via self-attention mechanisms. For instance, the code token *"def"* exhibits considerable attention toward tokens like *"the"*, *"method"*, *"is"*, *"a"*, and *"basic"* within the description. Conversely, the token *"method"* in the description concentrates its attention scores on the tokens within the method signature, such as *"def"*, *"bubble"*, and *"method"*. Meanwhile, tokens such as `+`, `=`, and `if` lack clear respective tokens in

The code of bubble sort:

```
def bubbleSort(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range(n - 1 - i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

The description: "The method is a basic implementation of the bubble sort algorithm in Python. Bubble sort works by repeatedly swapping adjacent elements if they are in the wrong order until the entire array is sorted."

Figure 4: An Example of Bubble-Sort Code

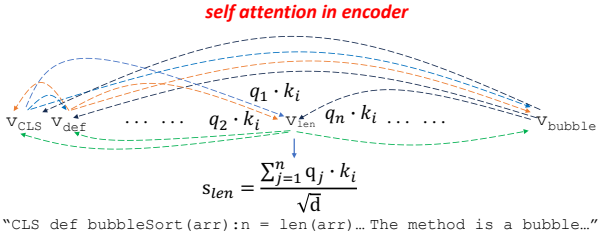


Figure 5: Example of accumulated self attention in DIETCODE on Fig. 4. Dotted lines with different colors represent the self-attentions of different tokens. s_{len} is the accumulated self-attention score of the token *'len'*.

Figure 6: An Example of Accumulated Self-Attention in DIETCODE on Fig. 4.

the text. Thus, the removal of such tokens may not notably impact the matching outcomes, as the models heavily rely on mapping information.

Figures 7a) and b) represent CLS attention scores on the tokens in the code and the description, respectively. The token <s> serves as the 'CLS' token. The deeper the color, the higher the value of the attention score. As seen in Fig. 7a), the CLS attention (the self attention of the token 'CLS') assigns greater importance to the tokens with more bi-modal mappings to form its vector representation. The **'CLS' token allocates significant attention to most tokens in the textual description, while emphasizing on the code tokens primarily within the method signature**.

Similarly, for code summarization, the bi-modal mappings are apparent. In Fig. 7c), when the next generated token is the token *"bubble"* (part of the method name) in the description, the encoder-decoder attention should emphasize on the tokens of method signature in the code. Fig. 7c) shows the encoder-decoder attention. As seen, **all tokens in the method signature are paid much attention than others**. More specific, with the current query vector q_t at position t in the decoder, the Encoder-Decoder score for each input token is calculated via Equation (6):

$$s_i = \frac{q_t \cdot k_i}{\sqrt{d}}, (1 \leq i \leq n) \tag{6}$$

Thus, maintaining the mappings between bi-modal data is crucial for bi-modal tasks, which can be accomplished using Encoder-Decoder attention. However, DIETCODE leverages the attention across all tokens, without giving emphasis to bi-modal mappings.

C Detailed Results and Analysis of 3 Research Questions in Preliminary Empirical Study

C.1 (RQ-1) What important tokens do CLS attentions emphasize on?

In Table 7, the five leftmost columns show the categories, the maximum of CLS attention scores, the minimum of CLS attention scores, the averages and variances of the global attention scores of tokens for each category. The global attention scores are used to calculate the average attention score of each token within a statement category. The last two columns of Table 7 display our proposed averages of category-local attention scores, and the

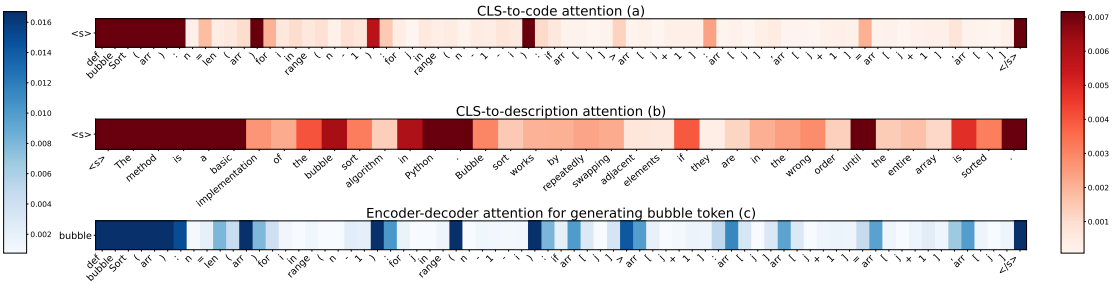


Figure 7: 'CLS' and encoder-decoder attentions on the input of the example of bubble sort. (<s> is the CLS token. Deeper color shows higher attention score. Fig. 7a) and b) represent the CLS attentions on the code tokens and the description, and Fig. 7c) shows the encoder-decoder attention for the generation of the "bubble" token.)

average variances of CLS attention scores of tokens grouped by statement categories. There are 21 categories used by DIETCODE, collectively representing over 95% of the statements in the dataset. In LEANCODE, we use these categories as contexts for tokens. They offer relatively straightforward setups for bi-modal mappings.

Analyzing the columns reveals that the category-local attention average of tokens within method signatures significantly surpasses that of others, even exceeding the second largest class ('return' statements) by a factor of eight. The `Return` class typically comprises the names of returned variables, which often convey crucial functional information. Logging and Annotation classes rank third and fourth, respectively, as they frequently encompass function-related data. Variable Declaration and Function Invocation are next, predominantly due to inclusion of variables or callee names.

Conversely, the `Break`, `Case`, and `Continue` exhibit the lowest category-local attention average. This is attributed to the fact that `Break` and `Continue` statements have keyword information, while `Case` statements, though potentially containing conditions, may be too detailed to establish meaningful bi-modal mappings with the description. Moreover, the average variance is directly proportional to the average attention scores, indicating that higher attention scores correspond to increased variance.

Comparing our proposed averages (last two columns) with the global averages (the 4^{th} and 5^{th} columns) in Table 7, except Method signature, the averages of attention scores/variances for the context-aware, category-local attentions as we propose are reduced from 0.55/5 times to 3.3/844 times in comparison to the global averages within each statement category. The variance of Method Signature increases 3.3 times when the average of attention scores goes up more than 4 times. Thus, **the categories of the statements successfully reflect the context of each token**.

C.2 (RQ-2) What important tokens do encoder-decoder attentions emphasize about?

Table 8 shows the average of the Encoder-Decoder attention scores of tokens based on statement types, called **Encoder-Decoder category-local attention average**. Unlike CLS attention, each token in the input can have multiple Encoder-Decoder attention scores, i.e., for each generated token, the decoder calculates an attention score for each token in the input. For example, Fig. 3b shows the Encoder-Decoder attention scores for the input at the generation of the token "bubble". Thus, the largest attention score is chosen as the attention score.

As depicted in Table 8, the categories with the highest and lowest importance remain consistent as 'Method signature', `Return`, `Continue`, and `Case`, respectively. However, the disparity between them has diminished. Certain categories, e.g., `Synchronized`, `For`, and `Throw`, have surpassed 'Logging', 'Variable Declaration', and 'Function Invocation', securing the 3^{rd}, 4^{th}, and 5^{th} places in importance. This shift indicates a redistribution of importance across categories. The Encoder-Decoder attention scores are generated in conjunction with the description. In the instances where the description contains intricate function details, these tokens garner high attention scores, facilitating the establishment of bi-modal mappings. For code search, the significance of details within the code (e.g., `Throw` statements) is lower compared to the broader functional description (e.g., 'Method signature').

Similar to Table 7, except 'Method' signature, comparing the 4^{th} and 5^{th} columns with the last

Table 7: (RQ-1) Statistics of CLS attention scores on 0.9M training dataset. (Max/Min: the max/min of CLS attention scores in each category; Global/Global_variance: the average/variance of global attention scores for each category; Category-local/Local_variance: the averages/variance of category-local attention scores.)

Category	Max	Min	**Global**	**Global_variance**	**Category-local**	**Local_variance**
Annotation	13.64774	0.00369	0.22834	1.09668	0.14712	0.25699
Arithmetic	3.83190	0.00269	0.23538	1.15031	0.05800	0.00915
Variable Declaration	10.49360	0.00353	0.24737	1.18534	0.10431	0.10301
Function Invocation	5.25585	0.00349	0.25464	1.23739	0.10638	0.09292
Return	6.59062	0.00353	0.27754	1.33069	0.20165	0.17654
Switch	4.36353	0.00352	0.23530	1.02716	0.07194	0.00936
Break	0.83018	0.00536	0.23083	1.02695	0.04734	0.00253
Setter	2.85442	0.00196	0.24565	1.16452	0.07033	0.04076
Synchronized	1.02363	0.00526	0.23753	1.00888	0.08507	0.01309
Try	4.33814	0.00371	0.24224	1.05390	0.08925	0.02697
Catch	2.07879	0.00385	0.24606	0.90369	0.05224	0.00641
Method Signature	29.17083	0.00353	0.33731	1.56490	1.74525	6.69616
Finally	0.53785	0.00439	0.22154	1.41311	0.09047	0.00939
Getter	7.59034	0.00373	0.24690	1.18191	0.06407	0.03392
Throw	5.20869	0.00346	0.23824	1.12604	0.08951	0.02551
Case	2.70762	0.00338	0.23824	1.15401	0.03953	0.00673
While	2.57405	0.00311	0.22953	1.06577	0.05870	0.01045
Continue	0.44534	0.00662	0.25256	1.35214	0.04609	0.00160
If Condition	4.03009	0.00262	0.24831	1.21111	0.08341	0.04783
For	22.29719	0.00348	0.24910	1.11164	0.07938	0.02098
Logging	4.90106	0.00278	0.24053	1.15288	0.14864	0.11938

Table 8: (RQ-2) Statistics of encoder-decoder attention scores based on 0.16M training dataset. (Max/Min: the maximum/minimum of encoder-decoder attention scores in each category; Global/Global_variance: the average/variance of the global attention scores of tokens for each category; Category-local/Local_variance: the averages/variance of category-local attention scores.)

Category	Max	Min	**Global**	**Global_variance**	**Category-local**	**Local_variance**
Annotation	7.94088	0.31593	2.61080	13.75766	1.54646	0.08766
Arithmetic	37.44108	0.06535	2.68657	15.41490	2.30496	2.52065
Variable Declaration	65.53831	0.08960	2.86623	15.63358	2.69239	7.96826
Function Invocation	63.96667	0.00918	2.86456	15.94000	2.80177	8.54368
Return	55.23188	0.09667	3.08082	17.61045	4.75692	16.01764
Switch	30.02598	0.07416	2.70932	16.36071	2.40672	2.63213
Break	28.02130	0.04057	2.64491	16.43219	2.66594	1.21296
Setter	69.05594	0.02932	2.84634	17.25399	2.32736	5.09966
Synchronized	78.08508	0.04113	2.84346	17.08405	3.11007	3.03446
Try	78.26600	0.02602	2.82429	17.31111	2.45927	2.69108
Catch	34.98584	0.07141	3.01365	19.80343	2.44455	4.18487
Method Signature	91.68832	0.13817	3.29116	18.21317	5.91461	30.92008
Finally	10.49109	0.74180	2.38186	7.77507	2.98907	1.73634
Getter	68.48614	0.02599	2.87695	16.58745	2.57884	6.41530
Throw	87.67154	0.05743	2.79540	16.04224	3.09842	8.12932
Case	23.25289	0.03370	2.74530	16.10519	1.79642	1.54551
While	67.68087	0.04015	2.69720	15.52064	2.40500	3.14009
Continue	9.85333	0.26809	2.48510	12.64186	1.72747	0.36762
If Condition	57.87812	0.05160	2.83536	15.84334	2.49570	5.97373
For	60.61899	0.03448	2.90877	17.21421	2.99445	6.88533
Logging	65.63157	0.03885	2.77210	15.53356	2.89339	8.41999

two columns, our proposed averages of attention scores/variances for context-aware, category-local attentions are much reduced from 0.1 to 156 times compared to the global averages within each category. However, the change from the global attention average to the category-local average is insignificant. This could be due to 1) our selection of the largest attention score as the representative, and 2) the difference in the Encoder-Decoder attention scores is not substantial across categories.

C.3 (RQ-3) Do the averages of self-attention scores reflect the CLS attentions and the Encoder-Decoder attentions?

Our answer is 'No'. *The accumulated attention scores from the self-attention (as used in* DIETCODE*) is for pre-training and cannot reflect and substitute for those from the CLS and Encoder-Decoder attentions. i.e., the self-attention is used for pre-trained tasks and vectored general representations, not directly for downstream tasks*. For elaboration, these attention schemes are for different tasks. The self attention is for pre-training tasks, while CLS attention is for fine-tuning downstream discriminative tasks, and the Encoder-Decoder attention is for downstream sequence-to-sequence generation tasks. In fact, the encoders of CodeBERT and CodeT5 have been trained in multiple pre-trained tasks. CodeBERT is pre-trained with two objectives: Masked Language Model (MLM, bimodal data) and Replaced Token Detection (RTD, unimodal data). In MLM, the model is trained to predict the identity of tokens that have been randomly masked in the input sequence. In RTD, the model is given an input sequence where some tokens have been replaced with incorrect ones. The model task is to predict which tokens are the original ones and which have been replaced. These objectives enable the model to capture the long dependencies between tokens in bimodal sequence (MLM) and unimodel (RTD) to obtain general representations via the self attention. Thus, the averages of self-attention scores cannot replace the CLS and Encoder-Decoder attentions. The latter attentions are directly applied to downstream tasks (Fig. 3).

To illustrate our above answer, Figures 8a) and 8b) show the heatmaps of the accumulated self-attention (in DIETCODE) and the Encoder-Decoder attention for the bubblesort example. Each row is one statement without indentation (except that lines 5-6 contain one statement since it is too long

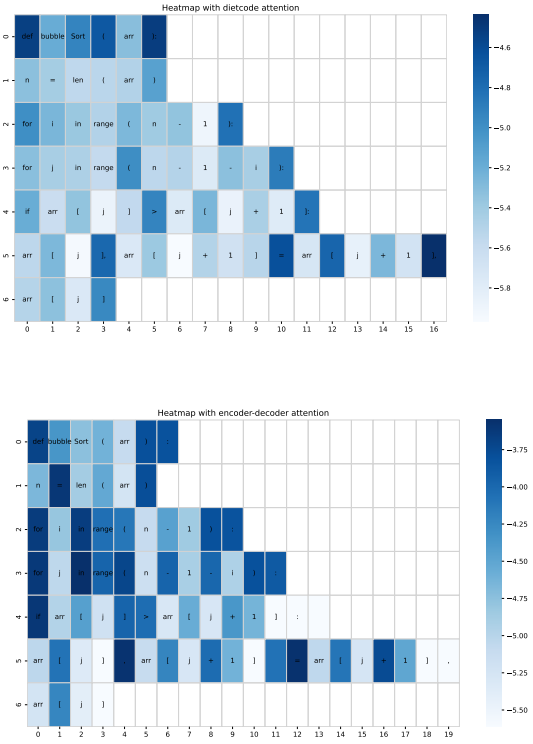


Figure 8: Bubble-sort: Heatmaps of (a) accumulated self-attention scores, (b) Encoder-Decoder attention scores

to show). The darker color means higher attention scores, i.e., the tokens are more important.

For comparison, the colors of the cells in Fig. 8a) (accumulated self-attention used by DIETCODE) have the shades in-between the ones of Figure 7a) for 'CLS' attention and Fig. 8b) for Encoder-Decoder attention. Specifically, for both method signature and method body, both contain tokens of higher importance as in Fig. 8b). The importance of tokens in the method signature is slightly higher than that of tokens in the method body in Fig. 7a). The above situation mainly stems from MLM and RTD tasks needing to do token-level prediction in code, thus requiring to pay attention on tokens in the method body. Keywords and separators can have high attention weights as they play important roles in predicting tokens (Zhang et al., 2022). In addition, since tokens in the method signature could contain more important 'guidance' for prediction, their attention scores could be larger.

In Fig. 8b), at each text token generation iteration, every code token has one attention score, and here the largest attention score in all iterations is used since the largest score represents the most

Table 9: Dataset Statistics (Total: # code snippets, Avg/-Max/Min: the average/max/min # of tokens in a code snippet or description, tr: training, val: validation, test: test dataset, Search: code search, Sum: summarization).

Dataset	Code Snippet				Code Description		
	Total	Avg	Max	Min	Avg	Max	Min
Search_tr	908,886	112.67	68,278	20	19.2	3439	1
Search_val	30,655	95.57	3,092	21	19.05	521	1
Search_test	26,909	113.42	5,542	20	20.22	709	1
Sum_tr	164,923	100.99	512	17	13.25	175	3
Sum_val	5,183	90.79	501	18	13.39	147	3
Sum_test	10,955	100.06	512	20	12.71	111	3

Table 10: Results of **LEANCODE** with DIETCODE's removal algorithm (10%-50% removal for each snippet, BLEU: BLEU-4 values, R-B:Reduced BLEU-4, R-M: Reduced MRR)

Ratio	Code Search				Code Summarization			
	CodeBERT		CodeT5		CodeBERT		CodeT5	
	MRR	R-M	MRR	R-M	BLUE	R-B	BLUE	R-B
Base	0.726	—	0.747	—	18.25	—	20.55	—
10%	0.701	3.44%↓	0.723	3.21%↓	17.40	4.66%↓	18.53	9.83%↓
20%	0.703	3.17%↓	0.717	4.01%↓	17.34	4.97%↓	18.18	11.53%↓
30%	0.702	3.31%↓	0.712	4.69%↓	17.31	5.15%↓	18.08	12.02%↓
40%	0.696	4.13%↓	0.713	4.55%↓	17.09	6.35%↓	18.03	12.26%↓
50%	0.682	6.06%↓	0.695	6.96%↓	16.73	8.32%↓	17.63	14.20%↓

important degree that the token has played in the generation process. As seen, the method body and signature are almost equally important, and import tokens are more evenly distributed for this code snippet. Namely, different generated tokens pay attention to different code tokens. This is largely different from the CLS attention distribution in code search, as shown in Fig. 7a), which emphasizes on the tokens mainly in the method signature.

D Dataset Statistics

Table 9 shows the detailed dataset statistics.

E Replacement Study

LEANCODE mainly consists of two aspects: token weights and a token removal algorithm (Algorithm 1). Here, we aim to determine how each of them contributes to LEANCODE's performance. Our procedure is to replace LEANCODE's removal algorithm in LEANCODE with DIETCODE's. Then, comparing the replacement results with the DIETCODE's results, the performance drop will reflect the contribute of token weights for LEANCODE' performance due to the same removal algorithm. Similarly, the difference of results of LEANCODE and the replacement method can show the contribute of LEANCODE's removal algorithm since both share the same token weights.

Table 10 shows the replacement results. Compared to DIETCODE'results in Tables 2 and 3, we can see that the performance drops of DIETCODE reach up to 37.1% for code search and 19.05% for code summarization. The performance drop conversely demonstrates that more accurate token weights significantly enhanced LEANCODE's performance. On the other side, compared to LEANCODE' results, we find that the maximal performance drops can be 3.7% for code search and 9.9% for code summarization. It demonstrates that the token-level removal algorithm (Algorithm 1) is much better than that of DIETCODE and significantly contributes to LEANCODE's performance.