Natural Is the Best: Model-Agnostic Code Simplification for Pre-trained Large Language Models

YAN WANG*, Central University of Finance and Economics, China
XIAONING LI*, Central University of Finance and Economics, China
TIEN N. NGUYEN, University of Texas at Dallas, USA
SHAOHUA WANG†, Central University of Finance and Economics, China
CHAO NI, Zhejiang University, China
LING DING, Central University of Finance and Economics, China

Pre-trained Large Language Models (LLM) have achieved remarkable successes in several domains. However, code-oriented LLMs are often heavy in computational complexity, and quadratically with the length of the input code sequence. Toward simplifying the input program of an LLM, the state-of-the-art approach has the strategies to filter the input code tokens based on the attention scores given by the LLM. The decision to simplify the input program should not rely on the attention patterns of an LLM, as these patterns are influenced by both the model architecture and the pre-training dataset. Since the model and dataset are part of the solution domain, not the problem domain where the input program belongs, the outcome may differ when the model is pre-trained on a different dataset. We propose SlimCode, a model-agnostic code simplification solution for LLMs that depends on the nature of input code tokens. As an empirical study on the LLMs including CodeBERT, CodeT5, and GPT-4 for two main tasks: code search and summarization, we reported that 1) the removal ratio of code has a linear-like relation with the saving ratio on training time, 2) the impact of categorized tokens on code simplification can vary significantly, 3) the impact of categorized tokens on code simplification is task-specific but model-agnostic, and 4) the above findings hold for the paradigm–prompt engineering and interactive in-context learning. The empirical results showed that SlimCode can improve the state-of-the-art technique by 9.46% and 5.15% in terms of MRR and BLEU score on code search and summarization, respectively. More importantly, SlimCode is 133 times faster than the state-of-the-art approach. Additionally, SlimCode can reduce the cost of invoking GPT-4 by up to 24% per API query, while still producing comparable results to those with the original code. With this result, we call for a new direction on code-based, model-agnostic code simplification solutions to further empower LLMs.

CCS Concepts: • **Computing methodologies** → **Neural networks**; • **Software and its engineering**;

Additional Key Words and Phrases: AI4SE, Machine Learning, Neural Networks, Code Simplification, Pre-trained Large Language Models

*Both authors contributed equally to the paper
†Corresponding Author

Authors' addresses: Yan Wang, Central University of Finance and Economics, Beijing, China, dayanking@gmail.com; Xiaoning Li, Central University of Finance and Economics, Beijing, China, 2022212378@email.cufe.edu.cn; Tien N. Nguyen, University of Texas at Dallas, Dallas, USA, tien.n.nguyen@utdallas.edu; Shaohua Wang, Central University of Finance and Economics, Beijing, China, davidshwang@ieee.org; Chao Ni, Zhejiang University, Hang Zhou, China, chaoni@zju.edu.cn; Ling Ding, Central University of Finance and Economics, Beijing, China, 2021312380@email.cufe.edu.cn.

ACM 2994-970X/2024/7-ART27
https://doi.org/10.1145/3643753

ACM Reference Format:
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. *Proc. ACM Softw. Eng.* 1, FSE, Article 27 (July 2024), 23 pages. https://doi.org/10.1145/3643753

1 INTRODUCTION

Pre-trained Large Language Models (PLLM) have emerged as powerful tools that can significantly impact the way software code is written, reviewed, and optimized, making them invaluable resources for programmers. They offer developers the ability to leverage pre-trained knowledge and tap into vast code repositories, enabling faster development cycles and reducing the time spent on repetitive or mundane coding tasks. The abilities to summarize the code in concise textual descriptions or to search for desired code are also valuable to developers in code comprehension. However, while these models offer substantial benefits, their adoption also presents multiple challenges. First, the fine-tuning and prediction for downstream tasks (e.g., code summarization or code search) with pre-trained large language models can be time-consuming and prohibitively costly, and eventually becomes practically infeasible. For example, an article in CNBC news reported [16] that the cost to develop and maintain the software can be extremely high with PLLMs. The company in the article was estimated to be spending lots of time and nearly $200,000 a month on PLLMs in order to keep up with the millions of user queries it needed to process each day. Second, due to the need for massive computational resources to train, fine-tune and predict these models effectively, they often place the limit on the number of input words or code tokens. The limit for the number of tokens to be fed to CodeBERT when run locally is 512 tokens. The word position embedding matrix for GPT-2 is (1,024x768) and for GPT-3 is (2,048x128). Other large language models, (e.g., CodeT5 [43], CodeGen [30], GPT-4 [4]), are heavy in computational complexity and quadratically with the length of the input code sequence. The key question is *"can we reduce the cost of applying PLLM in code-related tasks by simplifying its input, while maintaining the same performance?"*

Toward overcoming those challenges, DietCode [46] simplifies the input program of CodeBERT [7] with three strategies, namely, word dropout, frequency filtering, and an attention-based strategy which retains the statements and tokens that receive the *most attention weights* during pre-training. The strategies were chosen according to a preliminary empirical analysis from the authors, reporting that CodeBERT pays more attention to certain types of tokens and statements. Experimental results on a dataset with two downstream tasks show that DietCode provides comparable results to CodeBERT with 40% less computational cost in fine-tuning and testing.

However, we conjecture that the *decision on simplifying the input program should depend on the nature of the source code*, rather than on what input tokens the model pays more or less attention. Let us elaborate this point further. The decision to simplify the input program should not rely on the attention patterns of the CodeBERT model, as these patterns are influenced by both the model architecture and the pre-training dataset. Since the model and dataset are part of the solution domain, not the problem domain where the input program belongs, the outcome may differ when the model is pre-trained on different datasets. The rationale is that the attention patterns can be influenced by various factors, including the specific model architecture and the training data. Different datasets may exhibit different linguistic patterns, programming styles, or coding conventions, which can affect the attention patterns learned by the model during pre-training. Consequently, relying solely on attention patterns to determine the importance of input tokens may not yield consistent results across different datasets or model variations.

To ensure robustness and generalizability, it is advisable to consider a broader range of factors when making decisions about program simplification. We hypothesize that those factors could include the program's lexical tokens, syntactic structures, control flows, program dependencies,

and the desired outcome of the simplification process. Those factors can be valuable in guiding the simplification process, especially when dealing with potential variations introduced by different pre-training datasets. Keeping the above hypothesis in mind, we conducted our empirical study with the following research questions:

(1) **RQ-1: What is the impact of randomly removing code tokens on the performance of PLLM?** Our objective is to explore the relationship between the proportion of code removed and the efficiency of code summarization and code search of pre-trained large language models. To achieve this, we use random tokens with various simplification ratios to assess the impact on the training time and performance of PLLMs for downstream tasks.
(2) **RQ-2: What is the impact of removing lexical tokens on the performance of PLLM?** We remove simple type tokens from the code, which are the most basic elements in the program and account for a large proportion.
(3) **RQ-3: What is the impact of removing syntactical tokens on the performance of PLLM?** In order to examine the impact of tokens in key syntax structures on code simplification, we use Abstract Syntax Tree (AST) to select tokens to be removed, including the tokens in control structures, method invocations and signatures.
(4) **RQ-4: What is the impact of removing semantically non-essential tokens on the performance of PLLM?** We aim to identify the token types deemed to be semantically non-essential for removal, i.e., those with minimal impact on performance. Given the prevalent use of Program Dependence Graph (PDG) in representing source code semantics, e.g., in semantic code clone detection, we aim to remove the tokens absent in the PDG and evaluate the results on code-related downstream tasks.

In RQ-1, we evaluate the impact of randomly removing tokens from code on the performance of PLLMs for two tasks: *code search and code summarization*. We use five different levels of removal ratio and training time, as well as two different performance metrics, to measure the relations between removal ratio and the efficiency and effectiveness of the models CodeBERT and CodeT5. In RQ-(2-4), we systematically remove tokens by categories and structures to analyze the impact of different token types on PLLMs. We categorize tokens into six categories that reflect the three different natural levels of code, and use the tools AST-maven [26] and JAST2DyPDG [11], to generate ASTs and PDGs for source code, respectively.

The key findings of our empirical study are highlighted:

- **Key Finding 1 (RQ-1)**: The code simplification ratio has a linear-like relationship with the saving ratio on training time. For example, as 50% of the code was removed, the training time was also reduced by 51.5% and 31.76% on code search and summarization with CodeBERT.
- **Key Finding 2 (RQ-(2-4))**: The impact of token categories on code simplification varies significantly. E.g., removing symbol tokens that account for 51.38% of total code tokens results only causes a 2.8% decrease in CodeBERT's performance on code search. Meanwhile, removing variables that account for 15.48% of total tokens leads to a 12.52% decrease in CodeBERT's performance.
- **Key Finding 3 (RQ-(2-4))**: The impact of categorized tokens on code simplification is task-specific but model-agnostic. In the code summarization/search for both CodeBERT and CodeT5, the impact of the method signature/variables removal is the most significant on the models' performance, while the impact of symbol tokens removal is the least significant.

The above results on the impacts of different types of code tokens motivate us to develop a code simplification method that leverages the inherent properties of source code. This will enable efficient and effective code simplification, regardless of the underlying LLMs. We firstly model the problem of code simplification as 0-1 knapsack problem and propose SlimCode, a novel method of

code simplification based on the above empirical findings. SlimCode is applied to two classical downstream tasks (classification task and generative task) and we measure the performance. The empirical results in Section 5.4 show that SlimCode can improve the state-of-the-art technique, DietCode, by 9.46% and 5.15% in terms of MRR and BLEU score on code search and summarization, respectively. More importantly, SlimCode is 133 times faster than DietCode.

Additionally, we also found that the results reported in RQ-(1-4) are valid for the paradigm of pre-training, prompt engineering, and interactive in-context learning with GPT-4 (Section 6.2). For code search, SlimCode can reduce the inference time of GPT-4 by 27% and the cost of invoking GPT-4 by 24% per API query, with a 2.3% improvement of precision compared to results with the original code. In this paper, we make the following key contributions:

- **The state-of-the-art knowledge**. Our study is the first to leverage the nature of code to simplify the input code for pre-trained large language models and GPT-like models.
- **SlimCode: a new code simplification approach**. We quantitatively analyzed the impacts of different types of code tokens and proposed SlimCode, a model-agnostic code simplification method for PLLMs. Our code and data are available at https://github.com/gksajy/slimcode.

2 TOKEN CATEGORY AND REMOVAL

Large language models based on the Transformer architecture are aimed to learn the *lexical, syntactic, and semantic* knowledge of programming languages after pre-training, and map them to token vectors. Each token carries varying amounts of information at different levels. By categorizing tokens, we identify which words carry informative "hints" for downstream tasks.

In this study, we categorize different tokens in code snippets based on their nature. We consider six categories: *symbol tokens, identifiers, tokens in control structures, tokens in method invocations, tokens in method signatures, and tokens in Program Dependence Graphs (PDG).*

The first three categories are at the lexical level, as the selection criteria considers only the inherent features of the tokens themselves. The next three categories are at the syntactic level because their selection requires considering the grammar. The final category is at the semantic level, as token selection depends on program semantics. Let us provide more details.

2.1 Lexical Level

At the lexical level, we chose *symbol tokens* and *identifiers*.

2.1.1 Symbol Tokens. Symbol tokens are defined as the ones including brackets, separators and operators, *i.e.*, {=,+, -, *, /, %, ++, −−, !, ==, ! =, >,>=, <, <=, ||, ; <<, >>, >>>, |, (,), {, }, [,], „ ;, ., ;}. Symbol tokens are extensively used in programming to form the fundamental building blocks in a program. They specify the delimiters and the detailed instructions to the computer in certain operations. However, as the most basic elements, symbol tokens usually carry less high-level, syntax information and high-level semantic, downstream task-related information. Thus, removing such symbols is expected to greatly simplify code, as they make up a significant portion of code tokens.

2.1.2 Identifiers. Identifiers convey the intention of developers in programming tasks. Readability is a defining feature of identifiers, as they allow variables to have meaningful and descriptive names. This can improve code readability and make better understandability. Improved readability results in more "hints" for downstream tasks. This study examines the effects of removing identifiers.

2.2 Syntactic Level

At the syntactic level, we use AST to select tokens to be removed. We consider three types of tokens for removal: *tokens in control structures, tokens in method invocations, and tokens in method*

signatures. The tokens at the syntactic level carry more information about the code's content. They reflect the business logic behind the code to some extent and could provide more hints for solving downstream tasks. Therefore, we aim to study the impact of removing syntactic-level tokens.

2.2.1 Tokens in Control Structures. Tokens in control structures are defined as the code tokens that are used as keywords for control-flow statements. For example, the token `if` defines a conditional structure; the token `for` defines a repetition in the code, etc. To be specific, the tokens in control structures include the following: `for`, `if`, `try`, `catch`, `switch`, `while`, `do while` and their conditions.

2.2.2 Tokens in Method Signatures. Tokens in a method signature include the ones in the signature and in the parameters of the method. This type tokens can be found in the `MethodDeclaration` and `Parameter` nodes in an AST. Method signatures are also strongly related to code content, meanwhile they can provide information about the inputs and outputs of a method. This can make it easier for models to understand how the method works and how it can be used in their own code.

2.2.3 Tokens in Method Invocations. Method invocations include calls to methods of objects as well as calls to methods within the same class. Notice that a method invocation typically consists of method name, parameters, and return value(s), and all the information can provide potential or direct hints for the downstream tasks, such as the purpose and functionality of the method.

2.3 Semantic Level

Our goal is to identify the token types considered as *semantically non-essential* for elimination, specifically those with minimal impact on performance. A PDG captures the data and control dependencies between different statements of a program, providing significant semantic information. Control dependencies reflect the execution order of statements, and data dependencies reflect how variables are modified and used throughout the program. These dependencies can provide insights into the intended behavior of the program. Therefore, at the semantic level, we retained the tokens within an PDG. To assess the impact of semantic information on downstream tasks, we remove all tokens not present in the PDG of a program and measure the performance.

3 EXPERIMENTAL METHODOLOGY

In this section, we introduce our experimental design, including downstream software engineering tasks, models under study, datasets, evaluation metrics, and implementation.

3.1 Downstream Tasks

We evaluated the code simplification approaches in two downstream tasks: code search and code summarization. These tasks are commonly used in software engineering to demonstrate the ability of LLMs in natural language and programming language understanding [2, 7, 12, 31, 41].

(1) **Code Search**: This is a typical use case for pre-trained code models such as CodeBERT [7]. The goal of this task is to find relevant code snippets from a codebase given a query.
(2) **Code Summarization**: The goal of this task is to generate a natural language summary for a given code snippet. It is also widely used to evaluate pre-trained models for source code [24]. We used this task to verify the effectiveness of program simplification in generation tasks.

3.2 The Models Under Study

We opted for widely utilized models such as CodeBERT [7] and CodeT5 [43], alongside the cutting-edge PLLM, GPT-4 [4]. We aim to study the impact of code simplification on those models for code summarization and code search. Thus, the variant models under study are listed as follows:

- *CodeBERT-based code search.* CodeBERT is a BERT-based encoder. To perform code search, we need to add a fully connected layer on top of the CodeBERT model for binary classification.
- *CodeT5-based code search.* CodeT5 is a pre-trained model based on Transformer with an encoder-decoder structure. For code search, its encoder is separately extracted and joint with a fully connected layer for the classification task.
- *CodeBERT-based code summarization.* As CodeBERT has only an encoder, it cannot perform text generation tasks. Therefore, following the work of DietCode [46], we added to it a Transformer decoder for code summarization.
- *CodeT5-based code summarization.* As CodeT5 is suitable for text and code generation tasks, we use it directly for code summarization without any modification.
- *GPT4-based code search.* As GPT-4 provides only Web-based APIs, we cannot access its model. For classification problems like code search, we can only use prompts to get the classification results and corresponding analysis generated by GPT-4 in a specified format.
- *GPT4-based code summarization.* We input the code into the specified prompt, then submit it directly to GPT-4's Web API and receive the corresponding code description.

Currently, pre-trained large language models work under two different paradigms for natural language processing:

(1) **Pre-train and fine-tune paradigm**. In this paradigm, a model with a fixed architecture is first pre-trained as a Language Model (LM). Then, the pre-trained LM is adapted to different downstream tasks by introducing additional parameters and fine-tuning them using task-specific objective functions. CodeBERT and CodeT5 are the typical models used in this paradigm and also commonly used in software engineering community [7, 43].

(2) **Pre-train, prompt, and predict paradigm**. In this paradigm, a downstream task is reformulated to resemble those solved during the original LM training with the help of a textual prompt, rather than adapting pre-trained LMs to downstream tasks via fine-tuning. GPT-4 model is the state-of-the-art model [32] that can be used for code-related downstream tasks in this paradigm.

3.3 Datasets

Table 1. Statistics of the code snippets and code description in code search and summarization datasets (Total: the total number of code snippets, Avg/Max/Min: the average/maximum/minimal number of tokens in a code snippet or code description).

Dataset	Code Snippet				Code description		
	Total	Avg	Max	Min	Avg	Max	Min
Code_Search_train	904,817	112.67	68,278	20	19.2	3439	1
Code_Search_validate	30,437	95.57	3,092	21	19.05	521	1
Code_Search_test	26,780	113.42	5,542	20	20.22	709	1
Code_Summarization_train	164,813	100.99	512	17	13.25	175	3
Code_Summarization_validate	5,183	90.79	501	18	13.39	147	3
Code_Summarization_test	10,948	100.06	512	20	12.71	111	3

3.3.1 Data Collection. To compare results from different models in pre-train and fine-tune paradigm, we directly use the two datasets from CodeBERT [7]: code search and code summarization datasets. These datasets are the extensions of CodeSearchNet [10], which is a collection of datasets and benchmarks for code retrieval using natural language texts. It consists of approximately 2 millions pairs of (comment, code) that were extracted from the Github open-source repositories, covering six languages (Python, PHP, Go, Java, JavaScript, and Ruby). Our experiments were carried

Table 2. The statistics of the code snippets and description of 400 random samples from code search and summarization (Avg/Max/Min: the average/maximum/minimal number of tokens in code snippet or description.)

Data	Avg	Max	Min
Search_code	122.01	1680	24
Summarization_code	101.27	505	22
Search_description	22.45	478	1
Summarization_description	13.34	117	3

out only on the Java language since DietCode [46] reports similar results for different languages. The statistics of the two datasets are shown in Tables 1. Each dataset was split into three portions for training, validation, and testing.

3.3.2 Data Sampling. We have about 0.1M data points in our testing dataset. We need representative samples, as the inference of GPT-4 is expensive. Therefore, we set the confidence level of 95%, the margin of error E=5%, population proportion p=0.5 (as our classification task's category proportion is close to 50%). The formula for computing the sample size is as follows: $n = \frac{Z^2 \cdot p \cdot (1-p)}{E^2}$ where the Z value is chosen as 1.96 because the confidence level is 95%. That formula gives us a sample size n of 384. Thus, we chose 400 as our sample size. Our sample ensures sufficient sample size as well as the time and effectiveness. The statistics of the randomly selected samples are shown in Table 2. As seen, the statistics of our sampled dataset are approximately consistent with that of the original datasets. For the tasks of code search and summarization, the average/max/min number of tokens in code snippets are 122.01/1680/24 and 101.27/505/22, and the average/max/min number of tokens in code description are 22.45/478/1 and 13.34/117/3.

3.4 Metrics

The purpose of our work is to prune the input code to reduce training time while measuring the performance accordingly, given different pre-trained models and downstream tasks. Here we use the ratio of simplification to measure the degree of simplification of a code snippet *Code*. Given a code snippet *Code* and its corresponding simplified one *Scode*, the simplification ratio is defined as:

$$SimplifiedRatio = \frac{|Code| - |Scode|}{|Code|} \times 100 \quad (1)$$

where $|Code|$ and $|Scode|$ are the lengths of *Code* and *Scode*, i.e., the number of tokens.

The length of each code snippet and its corresponding simplified code are different. However, to have a fair comparison between different models, we need to set a fixed input size for all models. Thus, to follow the setting in DietCode [46], we set the input lengths of all original code snippets for code search and summarization to 200 and 256, respectively. Then, the input length of simplified code is $(1 - SimplifiedRatio) \times 200$ or 256.

The efficiency of simplified code is measured by the ***time cost*** it takes for fine-tuning in the pre-train and fine-tune paradigm, as well as for prediction in the prompting-prediction paradigm.

For evaluation, code search performance is measured using ***MRR*** (mean reciprocal rank), which is the average of the multiplicative inverse of the index for the first correct answer for the query. Code summarization performance is measured using the ***BLEU-4*** score, which calculates the average of n-gram precision on a couple of sequences. For code search in the second paradigm (i.e., GPT-4-like models), due to the high computational requirements of MRR, we use ***Precision*** which highly correlated with MRR on the models' effectiveness. Specifically, we randomly replace the code description in 400 samples and check if the replacement content matches the code part. Precision is

Table 3. Results of random-token removal on CodeBERT and CodeT5 for code search. (R-Time: Reduced Training Time in %, R-MMR: Reduced MMR in %.) (RQ1)

Ratio	CodeBERT				CodeT5			
	Time	R-Time	MRR	R-MRR	Time	R-Time	MRR	R-MRR
0%	433m	0.00%–	0.743	0.00%–	434m	0.00%–	0.749	0.00%–
10%	388m	10.39%↓	0.706	4.98%↓	391m	9.08%↓	0.737	1.60% ↓
20%	354m	18.24%↓	0.694	6.59%↓	355m	18.20%↓	0.726	3.07% ↓
30%	309m	28.64%↓	0.668	10.09%↓	310m	28.57%↓	0.696	7.08% ↓
40%	252m	41.80%↓	0.648	12.79%↓	253m	41.71%↓	0.679	9.35% ↓
50%	210m	51.50%↓	0.611	17.77%↓	212m	51.52%↓	0.641	14.42% ↓

Table 4. Results of random-token removal on CodeBERT and CodeT5 for summarization. (R-Time: Reduced Training Time in %, R-BLEU: Reduced BLEU-4 in %.) (RQ1)

Ratio	CodeBERT				CodeT5			
	Time	R-Time	BLEU-4	R-BLEU	Time	R-Time	BLUE-4	R-BLUE
0%	910m	0.00%–	18.58	0.00%–	916m	0.00%–	20.49	0.00%–
10%	840m	7.69%↓	17.35	6.62%↓	845m	7.75%↓	19.56	4.49%↓
20%	802m	11.87%↓	17.18	7.53%↓	807m	11.90%↓	19.47	4.93%↓
30%	734m	19.34%↓	16.75	9.85%↓	740m	19.21%↓	19.07	6.88%↓
40%	684m	24.84%↓	16.36	11.95%↓	689m	24.78%↓	18.36	10.35%↓
50%	621m	31.76%↓	15.06	18.95%↓	627m	31.55%↓	17.30	15.53%↓

the ratio of the number of correctly detected examples to the total number of all examples. The ratio of matching samples to non-matching samples is 50%.

3.5 Implementation

We set up CodeBERT [7] and CodeT5 [43] with default hyper-parameters. For optimization, they used the Adam optimizer with learning rates of 1×10^{-5} and 5×10^{-5} for two downstream tasks. We used a server with 2 CPUs of Intel(R) Xeon(R) Golden 2.40GHz and 2 GPUs of Nvidia A100.

4 EXPERIMENTAL RESULTS

4.1 Impact of Randomly Removing Code Tokens on the Performance of PLLMs (RQ-1)

4.1.1 Analysis Procedure. To answer this research question, we randomly removed code tokens from each given input code snippet, and then the simplified code is fed into CodeBERT and CodeT5 for evaluating the performance metrics, i.e., MRR and BLEU-4, and the corresponding training time. Particularly, for each code snippet in training, validation and testing datasets, we first calculated the length of its tokens. Then, we randomly removed 10%-50% of the total tokens and fed the simplified code input into the CodeBERT and CodeT5 models.

4.1.2 Empirical Results. Tables 3 and 4 show the empirical results on the positive correlations between the simplified ratio and training time, and those between the change of MRR and BLUE-4 of both CodeBERT and CodeT5 models. For instance, *when 50% of the code is removed, the training time for code search using the CodeBERT model decreases by 51.50%.* Similarly, for code summarization using CodeT5, *the training time is reduced by 31.55% when the simplification ratio is set to 50%.*

4.1.3 The Impact of Random-Token Removal on CodeBert and CodeT5. For CodeBERT, as the simplified ratio increases, *the training time follows a linear-like descent* on both of downstream

tasks. When 10%-50% of the code tokens is removed, the training time of CodeBERT on code search can be reduced by 10.39%-51.5% with a sacrifice of MRR in dropping 4.98%-17.77%. The percentage of reduced time is always greater than the performance sacrifice for CodeBERT on code search. *When the code simplification ratio increases (i.e., more code tokens are removed), the percentage of the training time saved also increases, but the speed of performance declines slower than the increase speed of the code simplification ratio.* We observe the similar trend for CodeBERT on code summarization. This could serve as a guidance for practitioners in the code simplification process.

4.1.4 CodeBERT versus CodeT5. CodeT5 achieves similar time-saving results as CodeBERT, but with better MRR and BLUE-4 performance in the same conditions. We observe the differences in the results from both models on two tasks. CodeBERT has only one encoder, while CodeT5 has an encoder-decoder structure. The pre-training of CodeT5 is much more complex than CodeBERT, and it could learn more correspondences between tokens of code snippets and the descriptions. Therefore, when removing random tokens, the CodeT5 model can also find more tokens with "hints" in the remaining code to help improve the effectiveness of downstream tasks.

For code search, the encoders of CodeBERT and CodeT5, and one fully-connected layer, are used for matching checks. The time complexity of the core module is $4LC^2 + 2L^2C$, where L is the length of the input code and C is the size of the dimension of each token. In our case, $C = 768$, and the average L is less than 120 (Table 1). Thus, the first factor of time complexity is much larger than the second one. When L decreases, the change in time follows a pattern similar to a linear curve.

For code summarization, CodeBERT has added text generation capability with a Transformer decoder as in DietCode [46]. As the decoder also requires a lot of training time, this makes the total training time for code summarization longer. Moreover, the proportion of time saved by simplifying the code can be lower in the total training time. This is consistent with the result in Table 3.

Moreover, from the columns *Ratio* and *R-Time* in Table 3: (10%,10.39%), (20%,18.24%), (30%,28.64%), etc., the ratio is 1:1 between time saved and code reduction ratio for code search. Similarly, the corresponding ratio is 1:0.6 for code summarization as observing the same columns in Table 4.

RQ1 Takeaway: *Minimizing the volume of code entered can significantly cut down on training time. In the context of code search, the correlation between time saved and code reduction is nearly one-to-one (1:1). However, in the realm of code summarization, this correlation is approximately 0.6 (0.6:1), indicating a slightly lower reduction in time compared to code reduction. With the same reduction in the input, the performance of CodeT5 is better than that of CodeBERT.*

4.2 Impact of Removing Lexical Tokens on the Performance of PLLMs (RQ-2)

4.2.1 Analysis Procedure. We removed all of the identifiers and symbol tokens from each code snippet in the training, validation, and testing datasets for CodeBERT and CodeT5. We then checked their training time and performance on both downstream tasks. Particularly, we used JavaParser [26] to build AST, then identified the locations of the identifiers in the code based on the `NameExpr` node in the AST. For simple symbols, we deleted them from the code snippets using string matching.

4.2.2 Empirical Results. Tables 5 and 6 display the results on the impact of removing lexical tokens including *identifiers and symbols* on code search and code summarization. For instance, removing all identifiers (accounting for 15.48% of the code tokens) leads to a 12.52% decrease in MRR for code search using the CodeBERT model. On the contrary, for symbol tokens, increasing the simplified ratio to 52.65% in the code summarization task led to only 0.59% decrease in BLEU-4.

4.2.3 Impact of Identifier Removal. The number of identifiers accounts for 15.48% of total tokens. Removing all identifiers can save training time about 15% and 9% for both models on code search

Table 5. Impact of categorized tokens on code search for CodeBERT and CodeT5, measured by training time and MRR. (R-Time: Reduced Training Time in %, R-MMR: Reduced MMR in %.) (RQ2)

Methods	Ratio	CodeBERT				CodeT5			
		Time	R-Time	MRR	R-MRR	Time	R-Time	MRR	R-MRR
Base	0.00%	433m	0.00%–	0.743	0.00%–	434m	0.00%–	0.749	0.00%–
Identifiers	15.48%	367m	15.24%↓	0.650	12.52%↓	375m	13.59%↓	0.683	8.81%↓
Symbol tokens	51.38%	215m	50.35%↓	0.722	2.83%↓	193m	55.53%↓	0.729	2.67%↓
Control structures	11.90%	381m	12.01%↓	0.715	3.77%↓	386m	11.06%↓	0.730	2.54%↓
Method invocations	37.47%	266m	38.57%↓	0.682	8.21%↓	268m	38.25%↓	0.698	6.81%↓
Method signature	15.96%	354m	18.24%↓	0.649	12.65%↓	354m	18.43%↓	0.680	9.21%↓
PDG	24.84%	332m	23.33%↓	0.713	4.04%↓	328m	24.42%↓	0.729	2.67%↓

Table 6. Impact of categorized tokens on code summarization for CodeBERT and CodeT5, measured by training time and BLEU-4. (R-Time: Reduced Training Time in %, R-BLEU: Reduced BLEU-4 in %) (RQ2)

Methods	Ratio	CodeBERT				CodeT5			
		Time	R-Time	BLEU-4	R-BLEU	Time	R-Time	BLUE-4	R-BLEU
Base	0.00%	910m	0.00%–	18.58	0.00%–	916m	0.00%–	20.49	0.00%–
Identifiers	15.69%	829m	8.90%↓	17.87	3.83% ↓	828m	9.61% ↓	19.49	4.88% ↓
Symbol tokens	52.31%	606m	33.41%↓	18.47	0.59% ↓	628m	31.44% ↓	20.34	0.73% ↓
Control structures	16.48%	820m	9.89%↓	18.57	0.05% ↓	843m	7.97% ↓	20.38	0.54% ↓
Method invocations	35.48%	692m	23.96%↓	18.17	2.21% ↓	703m	23.25% ↓	20.31	0.88% ↓
Method signature	11.36%	813m	10.66%↓	15.86	14.64% ↓	817m	10.81% ↓	16.61	18.94% ↓
PDG	23.62%	750m	17.58%↓	18.46	0.65% ↓	766m	16.38% ↓	20.39	0.49% ↓

and summarization, respectively, but with a sacrifice of performance. The performance sacrifice of a model on two tasks are different: CodeBERT drops 12.52% on code search and 3.83% on code summarization; CodeT5 drops 8.81% on code search and 4.88% on code summarization, indicating that *identifiers can affect code search more than code summarization.*

4.2.4 Impact of Symbol Token Removal. The number of symbols accounts for +50% of the total tokens. Removing all the symbols can save the training time by 51% and 31% in both models (based on the RQ1 results) on code search and summarization, respectively, but with a performance decrease. The performance sacrifice on two tasks are consistent: CodeBERT only drops 2.83% on code search and 0.59% on summarization; CodeT5 only drops 2.67% on code search and 0.73% on summarization, indicating that *symbols have less impact on code search and code summarization.*

4.2.5 Symbols versus Identifiers. The notable difference in the impacts of symbols and identifiers could be attributed to the following factors: 1) Identifiers exhibit a closer resemblance to natural language texts compared to symbols. This proximity to the source code's description imparts greater cues for both classification and generation tasks. 2) Typically, identifiers are closely linked to a method's name, which encapsulates its purpose or functionality [27]. Consequently, identifiers display a clear correlation with certain tokens in the code description. This correlation leads to numerous many-to-many correspondences following the fine-tuning process, thereby enabling identifiers to furnish a substantial array of indications for subsequent tasks.

RQ2 Takeaway: *Making up a substantial portion of the content yet have minimal impact on performance, symbols are well-suited for code removal. Conversely, while identifiers constitute a smaller proportion, they wield a more significant influence on the model's performance. Therefore, a decision to retain or eliminate identifiers should be made at a later stage in the code simplification.*

4.3 Impact of Removing Syntactic Tokens on the Performance of PLLMs (RQ3)

4.3.1 Analysis Procedure. We removed the tokens from the following syntactical structures: *control structures, method invocations*, and *method signatures*. For control structures, we used the JavaParser tool to convert each code snippet into an AST. Then, we identified the `for`, `if`, `try`, `catch`, `switch`, `while`, and `do-while` statements in the code segment by extracting the `ForStmt`, `IfStmt`, `TryStmt`, `CatchStmt`, `SwitchStmt`, `WhileStmt`, and `DoStmt` nodes from the AST. We then recognized their condition expressions based on parentheses, and finally removed all of them. The handling of method invocations and signatures is similar to that of control structures.

4.3.2 Empirical Results. Tables 5 and 6 show the impact of removing syntactic tokens on code search and code summarization. For instance, with a simplified ratio of 18.05% for CodeT5 on code summarization, there is an 11.36% drop in BLUE-4 score for token removal in method signatures. Similarly, for CodeBERT and the removal of tokens in method invocations and control structures, with a simplified ratio of 37.47%/11.90%, there is an 8.21%/3.77% drop in MRR in code search.

4.3.3 Impact of the Removal of the Tokens in Method Signatures. For both models, tokens in method signatures have the greatest impact on code search and summarization. For example, the removal of the signature tokens that account for 11.36% of all tokens in the code summarization dataset, reduces the BLEU score by 14.64% and 18.94% for CodeBERT and CodeT5, respectively. Additionally, their impact on code search is very similar to that of identifiers, while their impact on code summarization is significantly stronger compared to all other tokens, regardless of the model.

4.3.4 Impact of the Removal of the Tokens in Method Invocations and Control Structures. The impacts of tokens in method invocations and control structure are similar. For example, the tokens in method invocations and control structures account for 37.47% and 11.90% of all tokens in the code search dataset, and the removal of them reduces MRR by 8.21% and 3.77% for CodeBERT, respectively. Thus, the tokens in control structures have slightly more impact than tokens in method invocations. Removing tokens in control structures and method invocations has a higher impact on code search compared to that of code summarization.

4.3.5 Difference of Impacts among Tokens in Control Structures, Method Signatures and Invocations. The most profound impact is attributed to the tokens within method signatures. For example, the removal of the signature tokens that account for 11.35% of the whole code can cause a performance drop of 18.94% in BLEU score for CodeT5 on summarization. In typical instances, the names and formal parameters of regular methods are meticulously crafted, encompassing information that surpasses mere syntax, encapsulating the method's intention and operations. This fixed co-occurrence relationship between method signatures and their corresponding descriptive tokens facilitates the assimilation of their correlation by models. Consequently, tokens present in method signatures offer richer cues for downstream tasks, notably in steering the process of description generation. Tokens within method invocations and control structures hold significance due to their strategic positions within the AST, rendering the identification of corresponding tokens within the description a more straightforward endeavor. Notably, crucial branch conditions tend to be mirrored by pertinent tokens within the description.

RQ3 Takeaway: *The tokens within method signatures wield the most substantial influence over downstream tasks, particularly those pertaining to generation tasks. Tokens in control structures and method invocations possess a moderate impact on downstream tasks, surpassing symbol tokens by a significant margin and even outweighing the influence of identifiers. Tokens in method invocations and control structures have a higher impact on code search than on code summarization.*

4.4 Impact of Removing Semantically Non-Essential Tokens (RQ4)

4.4.1 Analysis Procedure. An PDG captures the interconnections of data and control among various segments of source code, offering substantial semantic insights. These inter-dependencies can unveil the envisaged program behavior. The PDG serves as a vehicle for representing source code semantics, with lesser significance placed on statements lying outside its scope in the code's representation. We use JAST2DyPDG [11] to convert each code snippet into a program dependence graph. Unlike the experiments in RQ-(2-3), we do not remove the tokens from a PDG. We remove the nodes (statements) that are not involved in either of two relations: data and control dependencies. We evaluate the training time and the performance of CodeBERT and CodeT5 on the simplified code.

4.4.2 Empirical Results. Tables 5 and 6 show the impact of removing tokens not in PDG on code search and summarization. For example, removing all tokens not in PDG (account for 24.84% of total tokens) leads to a 4.04% decrease in MRR for code search using CodeBERT. Meanwhile, the simplified ratio of 23.62% in the code summarization only leads to a 0.49% decrease in BLEU-4 for CodeT5, which indicates that the tokens outside the PDG is not important. We further analyzed the tokens outside the PDG and observed that most of them are in the code structures such as `try`, `catch`, and `finally`.

RQ4 Takeaway: *Tokens that are not present in PDGs have little impact on downstream tasks due to their little semantic information. The impact of tokens outside the PDG is less significant than that of tokens in control structures and invocations, but more significant than that of symbol tokens for both the CodeBERT and CodeT5 models on code search and code summarization.*

5 SLIMCODE: MODEL-AGNOSTIC CODE SIMPLIFICATION FOR PLLMS

5.1 Important Levels of Tokens

The crux of what we have learned from the previous experiments for RQ-(2-4) is the relative order of the importance of each type of tokens on the performance of the models in the downstream tasks. We have the order of importance as follows: **method signature > identifiers > control structures ≈ method invocations > symbol tokens**. This relative order is model-agnostic and even task-independent. Note that 1) although the tokens in control structures are slightly better than the ones in method invocation, we still consider them almost equal; 2) tokens not in PDG are not included in this ranking since building PDGs is time-consuming and might not always possible for incomplete code.

Table 7. The overlap of the categories of tokens (check/cross/line mark means that there is/isn't an overlap/meaningless in the two corresponding categories)

	Signature	Identifiers	Invocations	Structures	Symbols
Signature	–	–	–	–	–
Identifiers	✓	–	–	–	–
Invocations	×	✓	–	–	–
Structures	×	✓	✓	–	–
Symbols	✓	×	✓	✓	–

The principle of our novel model-agnostic code simplification technique is that *the tokens with the lower levels of importance should be removed before the ones with the higher levels*. The ranking score for the important level of a token is based on its category. An out-of-category token is the least important one. A token can belongs to multiple categories (Table 7). In some cases, a token may

Algorithm 1 SlimCode: Code Simplification Algorithm.

INPUT: $D = \{d_1, ..., d_m\}$, ranking scores V, $SimplifiedRatio$ and the original input length L
OUTPUT: A simplified code dataset D'
PROCEDURE:

1: Initialize D', a copy of D
2: **for** j from 1 to m **do**
3: Initialize an empty dictionary $removedTokens$ with positions and their tokens
4: **if** $n_j > (1 - SimplifiedRatio) \times L$ **then**
5: $\mathcal{W} \leftarrow n_j$–(1-SimplifiedRatio)×$L$
6: $currentWeight \leftarrow 0$
7: **while** $currentWeight < \mathcal{W}$ **do**
8: Add {index: token with highest v } ($\in d'_j$,$\notin removedTokens$) into $removedTokens$
9: $currentWeight \leftarrow sizeof(removedTokens)$
10: $d_j = d_j / selectedTokens[1 : \mathcal{W}]$
11: **return** D'

belong to more than one categories, which actually are more important than the ones belonging to one category. For example, an identifier can appear in a method invocation or a control structure.

Based on the above principle, we assign different rank scores to various tokens. Tokens with a lower rank score have higher importance. For example, the rank score of "signature" is the highest at 1, followed by identifiers in a control structure or method invocation with scores of 2 and 3. Identifiers in signature receive a ranking score of 1, as they are integral parts of the signature. Symbol tokens within the signature receive a ranking score of 7, emphasizing their uniform importance across all instances. Tokens in control structures and method invocations are not shown in Table 7. Their ranking scores are assigned with 6 and 5, respectively since control structures slightly outweigh method invocations. Other tokens are assigned with the score of 8.

5.2 Problem Formulation for Code Simplification

A dataset contains m snippets (documents) and each code snippet d_j ($1 \leq j \leq m$) can be considered as a sequence of tokens, denoted as $d_j = \{t_1, ..., t_{n_j}\}$. The index i of t_i records the position of the corresponding token in the code snippet. The same token can appear multiple times, i.e., $\exists i, j\ (i \neq j)$ and $t_i = t_j$. Each token t_i ($1 \leq i \leq n_j$) represents an item. v_i represents the ranking score of each token t_i, and the weights w_is of all tokens in d_j are equal to 1. With the given $SimplifiedRatio$, the total number of tokens to be removed for each snippet is $\mathcal{W} = n_j - (1 - SimplifiedRatio) \times L$ and L is the original input length.

We formulate code simplification as an 0-1 knapsack optimization problem. Given a set of n items numbered from 1 up to n, each with a weight w_i and a value v_i, along with a maximum weight $\mathcal{W}$, the knapsack problem is defined in Equation 2 as follows:

$$maximize \sum_{i=1}^{n} v_i x_i,\ such\ that \sum_{i=1}^{n} w_i x_i \leq \mathcal{W}\ and\ x_i \in \{0, 1\}. \tag{2}$$

5.3 SlimCode Algorithm

Algorithm 1 shows our greedy method (SlimCode) for code simplification. In Algorithm 1, at lines 1, SlimCode initializes a copy of the original code dataset D' as the returned simplified code dataset. At lines 2-10, it removes the tokens of each snippet in D' one-by-one based on their

Table 8. Results of SlimCode for CodeBERT and CodeT5 on code search and summarization.(10%-50%:removing 10%-50% tokens for each snippet. Time: time to running removal algorithms. Times:the multiple by which SlimCode is faster than DietCode.)

	Code Search						Code Summarization					
Ratio	**CodeBERT**		**CodeT5**		**Pruning**		**CodeBERT**		**CodeT5**		**Pruning**	
	MRR	R-M	MRR	R-M	Time	Times	BLUE	R-B	BLUE	R-B	Time	Times
Base	0.743	0.00% –	0.754	0.00%–	N/A	——	18.58	0.00%–	20.49	0.00%–	N/A	——
10%	0.740	0.82% ↑	0.749	0.66%↓	17m	32.2	18.56	0.11%↓	20.44	0.24%↓	45s	133.1
20%	0.721	1.77% ↓	0.748	0.80%↓	17m	28.9	18.29	1.56%↓	20.38	0.54%↓	53s	101.4
30%	0.734	0.00% –	0.751	0.40%↓	20m	21.9	18.62	0.22%↑	20.49	0.00%↓	59s	80.3
40%	0.733	0.14% ↓	0.757	0.40%↑	21m	18.3	18.41	0.91%↓	20.52	0.15%↑	66s	64.3
50%	0.719	2.04% ↓	0.745	1.19%↓	21m	18.3	18.63	0.27%↑	20.23	1.27%↓	69s	53.2

Table 9. Results of DietCode for CodeBERT and CodeT5 on code search and summarization.(10%-50%:removing 10%-50% tokens for each snippet, R-M: Reduce MRR, BLEU:BLEU-4 values, R-B:Reduced BLEU-4 values, Time: the time for removing tokens in code snippet dose not include weight retrieval.)

	Code Search					Code Summarization				
Ratio	**CodeBERT**		**CodeT5**		**Pruning**	**CodeBERT**		**CodeT5**		**Pruning**
	MRR	R-M	MRR	R-M	Time	BLUE	R-B	BLUE	R-B	Time
Base	0.743	0%–	0.754	0.00%–	N/A	18.58	0%–	20.49	0.00%–	N/A
10%	0.702	4.36%↓	0.730	3.18%↓	9h24m	17.68	4.84%↓	19.68	3.90%↓	1h40m
20%	0.686	6.54%↓	0.718	4.77%↓	8h28m	17.94	3.44%↓	19.77	3.51%↓	1h30m
30%	0.693	5.59%↓	0.714	5.31%↓	7h37m	17.73	4.57%↓	19.68	3.95%↓	1h19m
40%	0.679	7.49%↓	0.707	6.21%↓	6h45m	17.53	5.65%↓	19.42	5.22%↓	1h11m
50%	0.651	11.31%↓	0.676	10.34%↓	5h59m	17.67	4.90%↓	19.22	6.20%↓	1h02m

ranking scores stored in the dictionary V. In line 3, `removedToken` records the pair of the index and the corresponding tokens with the highest rank score ({index:token}) in d' at each turn. In line 4, $(1 - SimplifiedRatio) \times L$ is the number of tokens to be retained. Line 5 defines the number of removed tokens. In line 6, `currentWeight` represents the number of currently selected tokens to be removed. In lines 7-9, at each turn, the algorithm repeatedly selects the remaining token (not in `removedToken`) with the highest ranking score from left to right until the number of removed tokens is reached. Finally, each code removes the first $\mathcal{W}$ selected tokens (line 10).

5.4 Empirical Evaluation Results on SlimCode

5.4.1 Accuracy Comparison. Table 8 and Table 9 show that SlimCode outperforms the state-of-the-art DietCode [46] in all experiments for code search and summarizaton tasks. Specifically, SlimCode can improve DietCode by 6.48% and 4.27% in terms of MRR (i.e., code search) and BLEU score (i.e., code summarization) for CodeBert, and by 5.47% and 4.19% in terms of MRR and BLEU score for CodeT5, respectively.

Note that there are two interesting observations: 1) Intuitively, the performance of the removed code for downstream tasks might not be better than that for the original code, however, SlimCode can achieve better results. 2) Usually, the results of the code with a higher $SimplificationRatio$ should be worse. But, in some cases, the opposite is true for both methods. For instance, for CodeBERT in code search dataset, the performance of $SimplificationRatio = 30\%$ is better than that of $SimplificationRatio = 20\%$.

The reason is that if the code contains more useful information, the downstream tasks will have better results. In fact, not all code snippets can be fully inserted into models, as the maximum

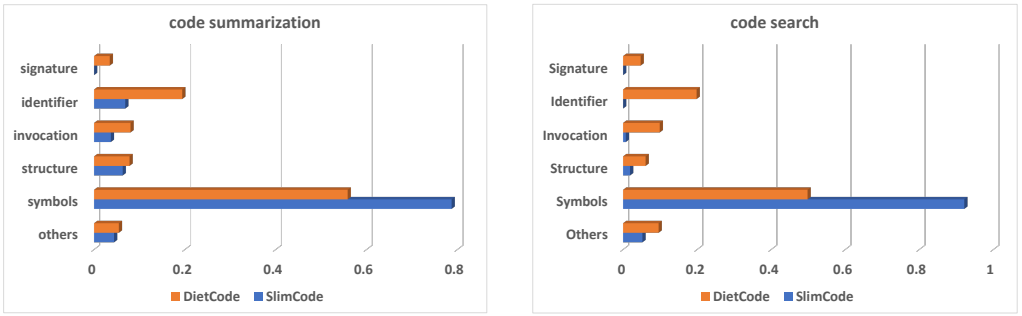

Fig. 1. The percentage distribution of removed tokens based on token categories.

lengths for original codes are limited to 200 and 256 characters for code search and summarization. Therefore, even if some tokens are removed (e.g., 50%) and the maximum lengths are reduced (e.g., $maxLength$=100), 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 two interesting observations mentioned above may occur.

Actually, SlimCode often removes low-quality tokens (e.g., symbols) while allowing high-quality tokens to enter the model input. For instance, as reducing by 20% tokens from a total of 200 tokens, the remaining top 160 tokens might still contain many symbol tokens. Consequently, more informative tokens sometimes do not make it into the top 160. However, when reducing by 30% tokens, the additional 10% for removal could contain more symbols. Thus, the remaining top 140 tokens could contain less symbol tokens and more important tokens could enter the top 140. Consequently, the result of $SimplificationRatio$=30% could be better than that of $SimplificationRatio$=20%.

5.4.2 Percentage Distribution of Removed Tokens based on Token Categories. To further compare SlimCode and DietCode, we present the percentage distribution of removed tokens by token categories (Fig. 1). The figure illustrates that SlimCode selects tokens to be removed based on their ranking scores in both code search and code summarization (Algorithm 1). Notably, removed tokens from the categories of symbol and other tokens can account for over 95% of the total tokens removed. Meanwhile, SlimCode removes much fewer tokens from method signatures and identifiers compared to DietCode. In contrast, while *DietCode also removes many symbol tokens that account for 50% of the total tokens removed, it removes many high-ranking tokens (or partially removed after tokenization), especially identifiers.* As a result, much important semantic information is lost, leading to a decrease in downstream task performance.

5.4.3 Time Efficiency. SlimCode can be up to 133 times faster than DietCode, which needs to run the model on the training dataset to obtain the weights of all tokens, and then remove tokens based on their weights. Table 8 and Table 9 show the running times of both token-removal algorithms on the training datasets, as well as the time efficiency improvement of SlimCode over Dietcode. From the tables, we can see that DietCode needs 1 hour to 10 hours to remove tokens with different $SimplificationRatio$s, and higher $SimplificationRatio$ means less pruning time. Notice that the pruning time of DietCode does not include the time for weight retrieval from a model.

Compared to SlimCode, DietCode requires an additional step of selecting code statements, which is converted into a knapsack problem. This step is implemented by dynamic programming, and has a time complexity of $O(|D_t| \times n_s \times (1 - SimplifiedRatio) \times (L + ml))$, where $|D_t|$ is the size of the training set, n_s and $(1 - SimplifiedRatio) \times (L + ml)$ are the average number of total statements and the number of candidate tokens to retain in code snippets, and ml is the average length of the

longest sentence in each code snippet. The complexity of SlimCode is $O(|D_t| \times n_t)$ according to Algorithm 1. Thus, even without considering other steps of DietCode, SlimCode is at least n_s times faster than DietCode since usually $(1 - SimplifiedRatio) \times (L + ml) > n_t$. For example, in code search with $|D| = 0.9M$, $n_s = 17$, $SimplifiedRatio = 10\%$, $L = 200$, $n_t = 112$, and $ml = 20$, the ratio of $\frac{|D_t| \times n_s \times (1 - SimplifiedRatio) \times (L + ml)}{|D_t| \times n_t}$ is 30. This is close to the experimental result of 32 times.

Moreover, the performance of SlimCode on code summarization can improve by up to 133 times, as the average length of code snippets in the summarization dataset has been reduced to 100. This means that more snippets do not need to be pruned with the given target length, and lines 6-10 of Algorithm 1 do not need to be executed. However, DietCode does not benefit from this change.

We also observe that DietCode using CodeBERT and CodeT5 generate two distinct lists of top 1,000 tokens using the learned weights from two models, with a low Spearman Correlation Coefficient value, i.e, 0.43. This finding suggests that the token weights from CodeBERT and CodeT5 are quite different. The reason could be that the pre-training loss functions, pre-training data, network structures, or tokenizers of CodeBERT and CodeT5 are not the same. Any of these differences can lead to changes in the weight of each token. Thus, *DietCode is model dependent, as its token weights learned from CodeBERT cannot be directly applied to CodeT5, and vise versa.*

We replicated the training processes to obtain the weights for all tokens in both models (they needs at least another 21 hours). We found that the Spearman Correlation Coefficient between the two lists of the top 1,000 tokens with the highest weights from the two models is low. This finding suggests that the token weights can vary between different models due to varying attention patterns and training datasets.

RQ5 Takeaway: *Overall, SlimCode has outperformed DietCode in code search and summarization by up to 9.46% and 5.15%, respectively. Importantly, SlimCode is much more efficient, being up to 133 times faster than DietCode. Importantly, SlimCode is model-agnostic, while DietCode is not as its attention scores for tokens learned from one model cannot be directly applied to another.*

6 USAGE OF SLIMCODE FOR GPT-4 WITH ZERO-SHOT-LEARNING

6.1 Analysis Procedure

In this study, our goal is to evaluate whether the previous results and conclusions are applicable to the pre-train, prompt, and prediction paradigm. We replicate the analysis procedures in the above RQ1-4 with GPT-4. As GPT-4 only provides a Web API, we used the selenium package to simulate a browser and communicate with GPT-4. We conducted code search and code summarization tasks using two prompts: 1) for code search, *"Please check whether the 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 code snippet. Answer in the following format."* Finally, we analyzed the results of GPT-4 to calculate the number of total tokens, precision, and BLEU-4.

6.2 Empirical Results of Replicating Previous Experiments on GPT-4

Tables 10 and 11 show the results of various code simplification methods on code search and code summarization respectively. Note that, here we use the total number of tokens (the input tokens and output tokens) to replace the prediction time, mainly because 1) the fee charged for the usage of computing resource of GPT-4 is based on the total number of tokens [33]; 2) GPT-4 only provides a web interface, and the actual prediction time is affected by multiple factors, which is usually difficult to measure accurately. Moreover, the input tokens not only include the code snippet, but also the prompts. From the Tables, we have the following empirical findings:

Table 10. Results of removal methods on GPT-4 for code search. (IT:Input Tokens, R-IT:Reduced Input Tokens in %, OT: Output Token, R-OT:Reduced Output Tokens in %; TT:Total Tokens, R-P:Reduced Precision in %)

Removal method	IT	R-IT	OT	R-OT	TT	R-TT	Precision	R-P
Base	69820	0.00%	44584	0.00%	114404	0.00%–	0.85	0.00%
Random(10%)	65144	6.70%↓	42352	5.01%↓	109728	4.09%↓	0.74	12.94%↓
Random(20%)	60275	13.69%↓	38844	12.87%↓	99119	13.37%↓	0.68	20.00%↓
Random(30%)	55411	20.64%↓	38204	14.31%↓	93615	18.17%↓	0.68	20.00%↓
Random(40%)	50548	27.60%↓	36852	17.34%↓	87400	23.60%↓	0.64	24.71%↓
Random(50%)	45649	34.62%↓	33472	24.92%↓	79121	30.84%↓	0.63	25.88%↓
Identifier	60976	12.67%↓	39637	11.10%↓	10063	12.10%↓	0.66	22.35%↓
Symbol tokens	43276	38.02%↓	39049	12.42%↓	82325	28.04%↓	0.78	8.24%↓
Control structures	60032	14.02%↓	41257	7.46%↓	101289	11.46%↓	0.65	23.53%↓
Method invocations	49824	28.63%↓	41348	7.26%↓	132520	15.83%↓	0.70	17.65%↓
Method signature	64925	7.01%↓	39324	11.80%↓	104249	8.88%↓	0.70	17.65%↓
PDG	52360	25.01%↓	40459	9.25%↑	92819	18.87%↑	0.71	16.47%↓
DietCode(10%)	64776	7.22%↓	40328	9.55%↓	105104	13.94%↓	0.65	23.53%↓
DietCode(20%)	59941	14.15%↓	36753	17.56%↓	96694	19.93%↓	0.64	24.71%↓
DietCode(30%)	55036	21.17%↓	39300	11.85%↓	94336	21.26%↓	0.62	27.06%↓
DietCode(40%)	50224	28.06%↓	37061	16.87%↓	87285	25.62%↓	0.57	32.94%↓
DietCode(50%)	45388	34.99%↓	35364	20.68%↓	80752	30.23%↓	0.56	34.11%↓
SlimCode(10%)	64776	7.22%↓	40988	8.06%↓	105769	7.55%↓	0.78	8.24%↓
SlimCode(20%)	59941	14.15%↓	39500	11.40%↓	99441	13.08%↓	0.71	16.47%↓
SlimCode(30%)	55036	21.17%↓	38224	14.27%↓	93260	18.48%↓	0.74	12.94%↓
SlimCode(40%)	50224	28.06%↓	38616	13.39%↓	88840	22.34%↓	0.74	12.94%↓
SlimCode(50%)	45388	34.99%↓	37879	15.05%↓	83267	27.22%↓	0.87	2.35%↑

6.2.1 Finding 1. As the simplified ratio increases, the total number of tokens and metrics decrease for both tasks in GPT-4. This empirical finding substantiates that code simplification leads to a reduction in computational resources required for Prompt-based Pattern Learning Model (PPLM). This can be attributed to the fact that GPT-4 requires the generation of a significant amount of analytical content by following a sequence of statements through reasoning, as opposed to solely producing binary responses. It demonstrates that in the absence of generating analytical text, the classification performance of GPT-4 is notably deficient, a characteristic linked to its pre-training methodology [32]. As the code becomes more incomplete, the analytical content of GPT-4 on the remaining code also decreases. Consequently, the total number of generated tokens and metrics decreases as well. Furthermore, GPT-4 lacks the capability to produce code descriptions closely resembling the ground truth, as it cannot undergo fine-tuning. In other words, it lacks exposure to the specific code description corresponding to each code snippet. The overlap of words between the generated description and the ground truth is considerably lower when compared to CodeBERT and CodeT5. Thus, the BLEU-4 scores exhibit a marked reduction relative to their preceding values and demonstrate a proximity to one another.

6.2.2 Finding 2. For code search, simplified code based on different token categories still shows different performance based on the GPT-4 model. Compared to CodeBERT and CodeT5, their impacts remain almost unchanged except that tokens in control structures and invocations have greater impacts. In the case of code summarization, the results across different categories tend to have a similar appearance, which differs from previous cases.

We find that in code search, tokens within the method structures and invocations exhibit greater significance compared to tokens within the method signatures. This disparity may be attributed to

Table 11. Results of various removal methods on GPT-4 for code summarization.(IT:Input Tokens, R-IT:Reduced Input Tokens in %, OT: Output Token, R-OT:Reduced Input Tokens in %; TT:Total Tokens, R-B:Reduce BLEU-4 in %)

Removal method	IT	R-IT	OT	R-OT	TT	R-TT	BLEU-4	R-B
Base	48037	0.00%-	15348	0.00%-	63385	0.00%-	5.50	0.00%-
Random(10%)	44361	7.65%↓	16249	5.86%↑	60610	4.38%↓	5.36	2.55%↓
Random(20%)	40509	15.67%↓	15532	1.20%↑	56041	11.59%↓	5.49	5.37%↓
Random(30%)	36680	23.64%↓	15401	0.34%↑	52081	17.83%↓	5.43	1.27%↓
Random(40%)	32813	31.69%↓	15424	0.50%↑	48237	23.90%↓	5.25	4.55%↓
Random(50%)	28912	39.81%↓	14961	2.53%↓	43873	30.78%↓	4.96	8.15%↓
Identifier	41665	13.27%↓	15392	0.29%↑	57057	9.98%↓	5.48	0.36%↓
Symbol tokens	27680	42.38%↓	16688	8.73%↑	44368	30.00%↓	5.22	5.09%↓
Control structures	42301	11.94%↓	16024	4.40%↑	58325	7.98%↓	5.44	1.09%↓
Method invocations	33168	30.95%↓	15501	0.99%↑	48669	23.22%↓	4.80	12.72%↓
Method signature	43664	9.09%↓	16313	6.28%↑	59981	5.37%↓	5.21	5.27%↓
PDG	42232	12.08%↓	15104	1.59%↓	57336	9.54%↓	5.78	-5.09%↑
DietCode(10%)	43876	8.66%↓	16080	4.77%↑	59956	5.41%↓	4.96	9.82%↓
DietCode(20%)	40105	16.51%↓	14845	3.28%↓	54950	13.31%↓	5.27	4.18%↓
DietCode(30%)	36277	24.48%↓	15076	1.77%↓	51353	18.98%↓	5.00	9.09%↓
DietCode(40%)	32484	32.38%↓	15100	1.62%↓	47584	24.93%↓	4.66	15.27%↓
DietCode(50%)	28724	40.20%↓	14521	5.39%↓	43245	31.77%↓	4.64	15.63%↓
SlimCode(10%)	44005	8.39%↓	15868	3.62%↑	59873	5.48%↓	5.42	1.45%↓
SlimCode(20%)	40188	16.34%↓	16544	7.79%↑	56732	10.49%↓	5.40	1.82%↓
SlimCode(30%)	36321	24.39%↓	16092	4.85%↑	52413	17.31%↓	5.50	0.00%-
SlimCode(40%)	32492	32.36%↓	16593	8.11%↑	49085	22.56%↓	5.20	5.45%↓
SlimCode(50%)	28724	40.20%↓	15848	3.26%↑	44572	29.68%↓	5.51	2.04%↑

their heightened influence on GPT-4's sequential reasoning process. Tokens within the method structures and invocations directly encapsulate the code's execution process, providing richer details about the code. Without the details, there cannot be a proper sequential reasoning process, and the quality of classification and text generation of GPT-4 will be severely compromised. Conversely, tokens within the method signature offer a broader overview of the code's functionality. It is evident that tokens involved in method invocations exert the most pronounced impact, while those associated with PDG have the smallest effect. This is likely because disrupting the sequence of method invocations significantly influences the generation of code descriptions in accordance with the code's order. Additionally, tokens in method signatures, identifiers, and control structures exhibit relatively substantial effects on BLEU-4 scores. Despite symbol tokens having a BLEU-4 score of only 5.22, their simplified ratio stands at 42%. Consequently, eliminating symbol tokens remains the most effective strategy to enhance code summarization.

6.2.3 SlimCode and DietCode applied on GPT-4. On average, *SlimCode outperforms DietCode with a 26% improvement in precision for code search and a 10% increase in BLEU-4 for code summarization.* In fact, the simplified code produced by SlimCode can even surpass the original code in performance. It is evident that the performance enhancement achieved by SlimCode surpasses that of DietCode in comparison to the improvements by CodeBERT and CodeT5. This discrepancy arises primarily from DietCode's approach of simplifying code through statement deletion, which poses a substantial challenge for GPT-4. Given GPT-4's reliance on sequential code analysis followed by downstream

tasks, this method may lead to GPT-4 failing to accurately infer the content of the removed statements. Conversely, *SlimCode's strategy of removing symbol tokens proves more advantageous for GPT-4 in "recovering" the deleted tokens based on its pre-trained knowledge.* This assertion is supported by results obtained through a random removal method.

Notably, an intriguing phenomenon emerges wherein *SlimCode with a SimplifiedRatio of 50% outperforms the original code on both tasks.* With a reduction ratio of 50%, all symbol tokens, along with certain tokens within control structures and method invocations, are removed. At this, GPT-4 encounters difficulty in reconstructing specific code details. Consequently, it leans more heavily on high-level semantic information gleaned from method signatures and identifiers, in conjunction with its pre-training knowledge, to execute downstream tasks. This adjustment may in fact enhance the efficacy of these downstream tasks. One conceivable explanation lies in the fact that when the original code contains an abundance of specific code details, GPT-4 tends to rely heavily on these details to perform downstream tasks. However, if these details are "inaccurate" (e.g., due to illogical variable or method names), GPT-4 is more prone to identify them as "matching contradictions" or "generating partially unreasonable expressions," resulting in a decrease in effectiveness.

If we remove 50% tokens by using SlimCode (the last row of Table 10), the precision of simplified code can be even better than that of the original code, with the improvement of precision about 2.35%. Additionally, the total number of tokens is reduced by 27.22%, which leads to cost saving because according to OpenAI's pricing policy [33], the cost is proportional to the number of tokens. Removing symbol tokens saves the cost of invoking GPT-4 by 24% here since the total cost of 400 samples just is 9.54 dollars. As the use of GPT-4 and other PLLMs becomes more prevalent, companies are likely to use them to solve thousands of other tasks. Our study can yield better results, save much time, and reduce more expenses. The same trend is observed for code summarization.

RQ6 Takeaway*: Experimental validation of the GPT-4 model confirmed that all findings from the previous research questions hold for the pre-training, prompt, and prediction paradigm. Importantly, SlimCode can yield better results, save inference time, and reduce expenses for smaller input sizes.*

7 DISCUSSIONS AND POTENTIAL USES OF SLIMCODE

7.1 Computational Efficiency Improvement

The computational complexity of code-oriented LLMs has been a concern, particularly as the length of input code sequences increases. Our work addresses this challenge by simplifying input code before feeding it to the LLMs. The linear-like relationship between code reduction and training time savings is a significant finding. This implies that by applying SlimCode, developers can substantially reduce the time required for model training and inference, making LLMs more practical and efficient.

7.2 Generalizability Across Models and Tasks

We showed that the impact of categorized tokens on code simplification is both model-agnostic and task-specific. This finding is crucial since it highlights that SlimCode can be applied across different LLM architectures such as CodeBERT, CodeT5, and GPT-4. This generalizability needs more experiments to ensures that the benefits of SlimCode can be realized across LLMs, making it a versatile solution for code simplification. More experiments are also needed with other tasks.

7.3 Reduced Dependency on Attention Patterns

SlimCode, which is not dependent on attention patterns, is a more robust and stable method for code simplification. This is important when the LLM is trained on different datasets, as the outcomes might vary due to varying attention patterns.

7.4 Cost Savings in API Usages to LLMs

Our study shows that implementing SlimCode can result in a significant reduction in the cost of invoking LLMs through API queries, specifically GPT-4. This cost-saving potential has practical implications for organizations and developers that heavily rely on LLMs for code-related tasks. By reducing the number of API queries needed, organizations can effectively manage their resources while still benefiting from the power of LLMs.

8 RELATED WORK

Pre-trained Models in Software Engineering and its Understanding. Pre-trained technologies have achieved a remarkable success in Natural Language Processing (NLP) [6, 24, 44] and pre-trained models (PTM) associated with programming languages [7–9, 13, 14, 17, 19, 20, 23, 39] also make a great process of code intelligence, including code repair [18, 22], code generation [5, 42], defect detection [21, 28, 29, 40], code summarization [2, 7, 12, 24], code search [31, 41], etc. Due to pre-trained models' powerful ability, many works [1, 3, 15, 25, 34, 36] investigated the mechanisms of PLM for code. Zhang et al. [46] focused on the specific knowledge (e.g., critical tokens/statements) learned by PLMs. Wan et al. [38] performed a structural analysis of PLMs for source code and found that the Transformer attention can capture high-level structural information. Ahmed and Devanbu [2] studied the impact of identifiers and that of non-identifiers.

Program Simplification. Several state-of-the-art approaches are proposed, including SIVAND [35] and P2IM [37]. These approaches are usually based on the delta debugging prototype [45], which treats a code snippet and an auxiliary deep learning model (e.g., code2vec) as the input. The model splits the code snippet into fragments and each fragment is subsequently treated as the input of the neural network model for performing testing tasks (e.g., method name prediction or misused variables detection). These methods are computationally inefficient (e.g., hundreds of hours for SIVAND to process 10K functions) since they need to run a deep learning model and evaluate the performance at each iteration. To solve that, Zhang et al. [46] proposed DietCode to improve further running efficiency (e.g., two minutes for DietCode to process 10K functions).

9 CONCLUSION

In summary, we propose SlimCode, a model-agnostic solution to code simplification that has far-reaching implications in terms of computational efficiency, generalizability, cost savings, and broader applicability within the field of software engineering (SE). The departure point of SlimCode is the exploration of the nature of code tokens and their impacts on the performance of LLMs. It not only addresses current challenges but also opens up new avenues for improving various SE tasks that rely on LLMs. The empirical results show that SlimCode can improve over DietCode by 9.46% and 5.15% in terms of MRR and BLEU score on code search and summarization, respectively. More importantly, SlimCode is 133 times faster than DietCode. Additionally, SlimCode can reduce the cost of invoking GPT-4 by up to 24% per API query, while still producing comparable results to those with the original code. We also call for a new direction on code-based, model-agnostic code simplification solutions to further empower LLMs in many other software engineering tasks.

10 DATA AVAILABILITY

Our code and experimental code are publicly available at https://github.com/gksajy/slimcode.

ACKNOWLEDGMENTS

The author Tien N. Nguyen is 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

[1] 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* (2021).

[2] Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1443–1455. https://doi.org/10.1145/3510003.3510049

[3] 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. https://openreview.net/forum?id=BJOFETxR-

[4] ChatGPT [n. d.]. OpenAI. https://openai.com/.

[5] 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* (2020).

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[7] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. 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 (Findings of ACL, Vol. EMNLP 2020)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[8] 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* (2022).

[9] 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. https://openreview.net/forum?id=jLoC4ez43PZ

[10] 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* (2019).

[11] JAST2DyPDG [n. d.]. JAST2DyPDG. https://github.com/hpnog/javaDependenceGraph.

[12] 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*. PMLR, 54–63.

[13] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2019. Pre-trained contextual embedding of source code. *arXiv preprint arXiv:2001.00059* (2019).

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

[15] 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)*. IEEE, 1332–1336.

[16] Latitude [n. d.]. Latitude. https://www.cnbc.com/2023/03/13/chatgpt-and-generative-ai-are-booming-but-at-a-very-expensive-price.html.

[17] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461* (2019).

[18] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 602–614. https://doi.org/10.1145/3377811.3380345

[19] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. A Context-based Automated Approach for Method Name Consistency Checking and Suggestion. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 574–586. https://doi.org/10.1109/ICSE43902.2021.00060

[20] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Fault Localization with Code Coverage Representation Learning. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 661–673. https://doi.org/10.1109/ICSE43902.2021.00067

[21] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 292–303. https://doi.org/10.1145/3468264.3468597

[22] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. DEAR: a novel deep learning-based approach for automated program repair. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 511–523. https://doi.org/10.1145/3510003.3510177

[23] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. 2022. CodeReviewer: Pre-training for automating code review activities. *arXiv e-prints* (2022), arXiv–2203.

[24] 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* (2019).

[25] 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)*. 336–347. https://doi.org/10.1109/ICSE43902.2021.00041

[26] Maven [n. d.]. AST Maven. https://mvnrepository.com/artifact.

[27] Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. 2020. Suggesting natural method names to check name consistencies. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1372–1384. https://doi.org/10.1145/3377811.3380926

[28] Chao Ni, Wei Wang, Kaiwen Yang, Xin Xia, Kui Liu, and David Lo. 2022. The best of both worlds: integrating semantic features with expert features for defect prediction and localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (, Singapore, Singapore,) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 672–683. https://doi.org/10.1145/3540250.3549165

[29] Chao Ni, Kaiwen Yang, Xin Xia, David Lo, Xiang Chen, and Xiaohu Yang. 2022. Defect Identification, Categorization, and Repair: Better Together. *arXiv preprint arXiv:2204.04856* (2022).

[30] 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* (2022).

[31] 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*, Luc De Raedt (Ed.). ijcai.org, 5546–5555. https://doi.org/10.24963/ijcai.2022/775

[32] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). https://doi.org/10.48550/arXiv.2303.08774 arXiv:2303.08774

[33] OPENAI-Pricing [n. d.]. OPENAI-Pricing. https://openai.com/pricing.

[34] Matteo Paltenghi and Michael Pradel. 2021. Thinking Like a Developer? Comparing the Attention of Humans with Neural Models of Code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 867–879. https://doi.org/10.1109/ASE51524.2021.9678712

[35] 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* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 441–452. https://doi.org/10.1145/3468264.3468539

[36] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. 2021. A primer in BERTology: What we know about how BERT works. *Transactions of the Association for Computational Linguistics* 8 (2021), 842–866.

[37] 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* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 945–955. https://doi.org/10.1145/3468264.3468545

[38] 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* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2377–2388. https://doi.org/10.1145/3510003.3510050

[39] Shaohua Wang, NhatHai Phan, Yan Wang, and Yong Zhao. 2019. Extracting API tips from developer question and answer websites. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, 321–332. https://doi.org/10.1109/MSR.2019.00058

[40] Wenbo Wang, Tien N. Nguyen, Shaohua Wang, Yi Li, Jiyuan Zhang, and Aashish Yadavally. 2023. DeepVD: Toward Class-Separation Features for Neural Network Vulnerability Detection. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2249–2261. https://doi.org/10.1109/

ICSE48619.2023.00189

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

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

[43] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. 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*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[44] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems* 32 (2019).

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

[46] 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* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1073–1084. https://doi.org/10.1145/3540250.3549094

Received 2023-09-20; accepted 2024-01-23