Rethinking the Role of Structural Information: How It Enhances Code Representation Learning?

Qiushi Sun
Department of Mathematics
National University of Singapore
Singapore
qiushisun@u.nus.edu

Nuo Chen
School of Data Science and Engineering
East China Normal University
Shanghai, China
nuochen@stu.ecnu.edu.cn

Jianing Wang
School of Data Science and Engineering
East China Normal University
Shanghai, China
lygwjn@gmail.com

Xiaoli Li
Institute for Infocomm Research
Agency for Science, Technology and Research
Singapore
xlli@i2r.a-star.edu.sg

***Abstract*—Code pre-trained models (CodePTMs) have recently exhibited remarkable accomplishments in the realm of software engineering. However, there are still limited advancements in understanding the inner mechanism of these models, as well as their sensitivity to samples of varying quality. Codes have a more rigid and structured syntax compared to natural languages; hence, leveraging and understanding structural information becomes essential for analyzing, interpreting, and utilizing CodePTMs. While previous studies have verified models' ability to acquire knowledge from code structure through techniques such as attention analysis and probing tasks, *the specific roles it plays in downstream tasks have yet to be explored*. In this work, we propose a set of novel and practical methods for probing and exploiting the structural information within the code. In particular, dataflow perturbation experiments are first employed to explore the sensitivity of models with varying levels of structural information when confronted with input changes. Based on our findings, structure-aware exemplars selection strategies are proposed for both code generation and understanding, aiming to recover the model performance at minimal cost under perturbed conditions. Moreover, efficient fine-tuning can be achieved by utilizing exemplars instead of full fine-tuning.**

***Index Terms*—Pre-trained Language Models, Software Engineering, Code Intelligence**

I. Introduction

The remarkable achievements of pre-trained language models (1) have catalyzed the development of their counterparts for *programming languages* (PLs). Under the assumption of "Software Naturalness" hypothesis (2; 3), which posits that PLs can be processed similarly to natural languages (NLs), researchers have treated source codes as sequential data. Consequently, sequential neural architectures, such as the transformer-based models (4; 5), have been implemented to understand and generate programs (6; 7). Owing to the vast training corpora (8), Code pre-trained models (CodePTMs) demonstrate outstanding capabilities in various downstream tasks (9) of code intelligence, *e.g.*, defect detection, code translation, and code summarization (10).

Beyond learning from code tokens, it is worth noticing that code can be represented in two modals: the source code and the structure of code extracted from their parsed *abstract syntax tree* (AST). Thus, these models further leverage code structures, such as data flow (11) and AST itself (12; 13; 14; 15; 16), to learn code representation. Previous work has demonstrated that integrating the aforementioned structural information during the pre-training stage can enhance the model's performance across various tasks, leading to a better understanding of code. Subsequently, researchers have made progress in studying how models learn structural information. These studies primarily focus on three aspects: [1] Probing tasks, which involve designing specific tasks to test the code's understanding of the structure (17; 18). [2] Attention analysis, which aims to find the relationship between attention scores and structure (19). [3] Performance analysis, which seeks to establish the connection between the level of code's structural understanding and its performance in downstream tasks (20). Yet, there is a critical gap that lingers in existing research. While substantial insights have been gleaned on the model's ability to capture structural information, less is known about how structure information can be harnessed in practice.

From a psychological perspective, the order of words usually does not significantly affect the reading process (21). During reading, the brain does not process the text in a "word-by-word" manner. Instead, it scans sentences or paragraphs, comprehending them based on the context and prior knowledge (22). In the same vein, perturbations in the sequence of text elements often yield marginal impacts on NLP task performance (23). However, regarding CodePTMs, it's reasonable to surmise that they may perform "regional scans" of the underlying structure when dealing with highly structured text like code, such as identifying a subtree within the textual data that corresponds to a syntactic and semantic unit (24; 25). In this intricate landscape, the rigid syntax of code is likely to

Work done during Qiushi Sun's internship at Institute for Infocomm Research (I^2R), Agency for Science, Technology and Research (A*STAR).

exacerbate the impacts of token-level perturbations.

In this paper, we first conduct exploratory perturbation experiments on codes at the textual level, evaluating the impact of input features and dataflow information on code representation learning. Then, we discuss and analyze whether the model can further utilize the structural information of the code. Upon confirming the influence of code structure, we design a structure-aware exemplars selection method based on the complexity of code snippets, which aims at selecting the samples with "rich" structure information. These exemplars can be used to recover or approximate the original performance of fine-tuned CodePTMs and enable us to perform efficient fine-tuning. Furthermore, through the evaluation of multiple downstream tasks across multiple backbones, we unveil models' capability to utilize structural information. Our main contributions are summarized as follows:

- This is the pioneering research to validate CodePTMs' capability to harness structural information in downstream tasks, by subjecting the code to textual perturbations. Additionally, we provide interpretations from the perspective of dataflow.
- We propose a novel *structure-aware exemplars selection strategy* that leverages ASTs to choose representative code snippets that possess rich code structural information. By carefully selecting these exemplars, we successfully recover the performance of the perturbed model and enable efficient fine-tuning.
- Experimental results across prevailing backbones demonstrate the effectiveness of our proposed method on representative downstream tasks.

II. Preliminaries and Pilot Experiments

We first conduct pilot experiments to show the effect of textual perturbation across tasks and models, laying the groundwork for subsequent research.

A. Criteria for Perturbations

The exploration of textual perturbations designed for code remains in its nascent stages, with no well-established guidelines or criteria currently in place. Nevertheless, perturbation of code data is a real-world problem, occurring either intentionally or unintentionally. Following previous research (26), we consider these perturbations should be label-consistent, non-adversarial and can be generated at scale automatically. Therefore, we adopt a straightforward approach of randomly swapping code tokens, particularly identifiers in AST. More specifically, we implement two types of perturbations, applying them to 50% and 100% of the identifiers, respectively[1].

B. Exploration of Code Perturbation

We begin by conducting perturbation experiments on four representative CodePTMs, covering both code understanding and generation tasks. Concretely, the models are first fine-tuned on the whole set of perturbed training data and subsequently evaluated on testing data[2]. The results are presented in Table I, revealing that textual perturbations have disrupted the performance obtained from fine-tuning the model on golden data for all CodePTMs. Moreover, it is important to note that different models and tasks exhibit significant variations in performance losses due to these perturbations. Notably, the clone detection task, which primarily relies on semantics rather than code structure comprehension, is least affected by the perturbation. In contrast, tasks such as defect detection which necessitates an understanding of code logic, or the code translation task that involves grasping the correspondence of code tokens across two languages, are substantially influenced by the perturbation. Based on the insights gained from our pilot experiments, we aim to investigate the following three research questions (RQs):

Methods	Clone	Defect	Code Translation	
	F1	Acc	BLEU	EM
GraphCodeBERT				
Fine-Tuning	95.00	62.88	77.49	59.85
Fine-Tuning (Pert.)	94.75 -0.25	61.86 -1.02	59.04 -18.45	47.15 -12.7
PLBART				
Fine-Tuning	93.60	63.16	81.13	63.35
Fine-Tuning (Pert.)	93.99 +0.33	62.48 -0.68	74.96 -6.17	57.85 -5.50
CodeT5				
Fine-Tuning	95.00	65.78	81.63	65.85
Fine-Tuning (Pert.)	95.16 +0.41	63.03 -2.75	77.52 -4.11	61.60 -4.25
UniXcoder				
Fine-Tuning	91.36	62.34	76.59	63.45
Fine-Tuning (Pert.)	89.77 -1.59	60.94 -1.40	69.64 -6.95	57.80 -5.65

TABLE I
PILOT EXPERIMENTS ON CODE PERTURBATION. FOR CODE TRANSLATION, WE REPORT THE AVERAGE PERFORMANCE ON THE TASKS OF C# ↔ JAVA TRANSLATIONS.

- **RQ1**: What is the interplay between the model's characteristics and the results of perturbations among various code-related tasks?
- **RQ2**: The perturbation of code tokens also disrupts the structure of the code. Can we restore the performance of the perturbed model from the perspective of code structure?
- **RQ3**: Is it possible to facilitate the model's acquisition of knowledge for downstream adaptation through the use of limited samples rich in structural information?

III. An Alternative View On Perturbation

A. Dataflow Perturbation

In section II-A, we propose two levels of identifier perturbations. While these modifications are visible at the textual level, their impact extends beyond that. The models' unusual performance in clone detection indirectly supports this observation. Dataflow, a graph that represents variable dependency relationships, plays a crucial role. Its nodes represent variables,

[1]For all experiments involving perturbation, we mitigate the impact of randomness by conducting two independent trials and taking the average value.

[2]For a fair comparison, the experimental settings of pilot experiments remain consistent with those of the main experiments.

and edges indicate the source of each variable's value. Unlike the AST, dataflow remains consistent across different abstract grammars for the same code snippet. This inherent structure of the code provides essential semantic information, which is vital for understanding the code.

For natural languages, the structure and semantics of the text are more flexible and diverse, potentially maintaining their basic meaning and readability even after certain perturbations. In contrast, such perturbations can be detrimental to source codes, as the execution sequence may be altered, leading to errors such as "using a variable before defining it" or creating incorrect dependencies between functions. These impacts are reflected in the dataflow, as shown in Figure 1.

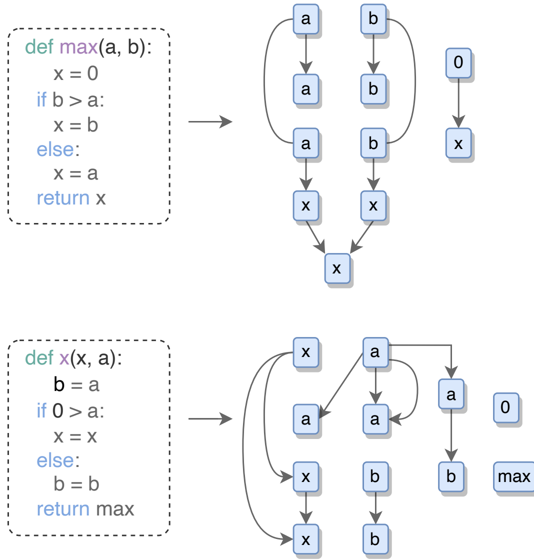

Fig. 1. A comparison of the dataflow graphs from the same code snippet before and after perturbation (swap *max* and *x* partially), where the corrupted one fails to illustrate the dependency relationships among variables.

B. Explaining Perturbation Effects in CodePTMs

Based on the above analysis, we can now elucidate the implications of perturbations across different CodePTMs and various tasks. It becomes evident that perturbations have a relatively minimal impact on the performance of the clone detection task. Even when presented with perturbed samples, the model exhibits the capability to predict code similarity at a semantic level that remains consistent. In contrast, for defect detection task, the performance degradation due to perturbation is more significant because the swapped identifiers disrupt dependency relationships among variables, making it more challenging for the model to identify defects such as logical flaws accurately. For the code translation task, we observe a notable decline in performance on both evaluation metrics. Clearly, for generation task, perturbations in the training samples significantly impede the model's ability to construct coherent code sequences. We also see similar phenomena in subsequent code summarization experiments. Contrasting the learning dynamics associated with NL representations, the impact of perturbed data extends beyond impairing syntactic coherence—it also predisposes the model to assimilate erroneous code structure.

IV. Utilizing the Structure-Aware of CodePTMs

After conducting the analysis of the perturbation results based on dataflow, we demonstrate dataflow structures play significant role in code representation learning. As in RQ3, we now utilize these findings to identify representative samples from dataflow perspective. This serves dual purposes: 1) recovering the performance of the perturbed model, and 2) facilitating efficient fine-tuning.

A. Beyond Abstract Syntax Tree

The Abstract Syntax Tree (AST) encapsulates rich structural information, but its tree structure may lead to long-range problems due to the significant distance between leaf nodes. As such, we incorporate data flow edges to trace variable transitions (27; 28), and establish connections between adjacent leaf nodes to bolster the overall connectivity of the AST, thus creating a novel version called Upgraded AST (U-AST).

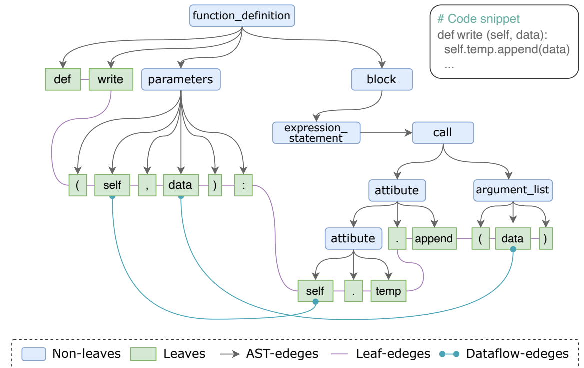

Fig. 2. A Python code snippet and its parsed AST, with connected Dataflow edges and Leaf edges.

B. Evaluate Code's Structural Information

Previous experiments have confirmed the significance of dataflow across a range of coding tasks. Next, we further explore, evaluate, and utilize the structural information inherent in the code.

In a program, dataflow connectivity is instrumental in outlining control flow, which represents the sequence and organization of operations. It also reveals data dependencies, signifying how changes in one section of code can impact other interconnected sections. Viewing U-AST as a strongly connected graph and inspired by the efficiency of communication network (29), we adopt a quantitative metric for the structural information of the code. This metric is designed to measure the communication efficiency between pairs of nodes (*e.g.*, identifier tokens), rather than merely the connectivity between adjacent nodes.

$$E(G) = \frac{1}{N(N-1)} \sum_{i \neq j} d_{ij} \tag{1}$$

The equation 1 demonstrates the underlying concept, where G represents the AST, N denotes the total number of nodes

Algorithm 1 Exemplars Selection Process

Input:
Code snippets $\mathbf{C} = [c_1, c_2, \ldots, c_n]$
Empty exemplar list $\mathbf{E}$
Desired number of exemplars k

Output:
Updated exemplar list $\mathbf{E} = [e_1, e_2, \ldots, e_k]$

```
0: procedure SELECTEXEMPLARS(C, k)
0:   for i = 1 to n do
0:     T ← ConvertToAST(c_i)
0:     G ← AugmentToUAST(T)
0:     ge ← ComputeGE(G)
0:     l_i ← 1/ge {Compute the inverse of global efficiency}
0:     if length(E) < k then
0:       AddToExemplars(E, (l_i, c_i)) {Add to exemplars if not full}
0:     else if l_i > GetTopPriority(E).l then
0:       ReplaceTopPriority(E, (l_i, c_i)) {Replace top priority if new one is higher}
0:     end if
0:   end for
0:   return E {Return the selected exemplars}
0: end procedure=0
```

in the AST, and d_{ij} signifies the pairwise distances between any two tokens. Global Efficiency (GE) quantifies the average inverse shortest path length between all pairs of nodes within the AST.

We select exemplars from the training set for each task. As demonstrated in Algorithm 1, we first convert code snippets into ASTs and augment them by incorporating leaf node connections and data flow edges to construct U-ASTs. Subsequently, we compute the global efficiency of each U-AST, and sort them in ascending order, establishing the basis for selecting exemplars. It's worth noting that our computation of $E(G)$ is based on the U-AST, as shown in Section IV-A. While calculations can be performed directly using the AST, the greater distances between leaf nodes in the AST often impede the effective representation of data flow information. To validate our perspective, we have conducted ablation studies, the comprehensive results of which are presented in Section V-D.

C. Exemplar-based Fine-Tuning

With the aid of our structure-aware exemplars selection method, we can identify representative samples that help restore the model's performance.

a) Performance Recovery: The exemplars selection is based on a code snippet, while our representative downstream tasks involve both multi-code tasks and natural language text. Therefore, it's necessary to clarify the subtle differences in exemplar selection across different tasks. For Code Understanding tasks that only include code data, the selection of exemplars is based directly on the training set. For generative tasks, such as code translation, the selection process depends on the code to be translated. For instance, in the case of Java → C# translation, the selection is based on Java code, and vice versa. As for the code summarization task, the selection process is based on the code being summarized.

b) Efficient Fine-Tuning: Employing exemplar selection to help efficient fine-tuning is straightforward, with the sample selection method being essentially the same as that in *Performance Recovery*.

V. Experiments

A. Experimental Setup

a) Backbone Models.: In our experiments, we employ a wide range of CodePTMs with different architectures and scales to substantiate the universality of our findings and the efficacy of our approach. Specifically, we employ GraphCodeBERT-base (11),PLBART-base (30), CodeT5-base (31), UnixCoder-base (15), as our backbones. It is worth noticing that these models utilize code structural information to varying degrees and through different approaches.

b) Tasks and Datasets: We conduct our experiments on four code representation learning tasks from the CodeXGLUE benchmark (9). For code generation, we first employ *code summarization* (32) that aims to generate natural language comments for the given code snippet in a different programming language. The second code generation task is *code translation* (33), which involves translating a code snippet from one programming language to another. For code understanding, *Clone detection* (34; 35) quantifies the similarity between code snippets. *Defect detection* (36) seeks to predict the presence of vulnerabilities in the source code that could pose threats to software systems.

c) Implementation Details: We utilize Tree-sitter[3] to parse source codes into ASTs. For the experiments on both performance recovery and efficient fine-tuning, we adopt the existing framework (37) for training.

Due to page limitations, we follow the configurations established by CodeXGLUE (9) across all tasks to ensure a fair comparison in terms of hyperparameter settings. For experiments involving CodeBLEU (38), we apply the same settings as stated in the literature.

B. Performance Recovery

Building upon our pilot experiments, we designate code data that has undergone perturbation as noise-infused data. We first use the selected exemplars to recover the performance of the perturbed CodePTMs, as illustrated in Table II. For the perturbation part, the models display a decline in performance when encountering perturbations in general. However, CodeT5 appears to be the most robust model, showing stable performance of code understanding and only slight performance decreases among other selected code-related tasks.

For almost all perturbed models and tasks involved, we can reach noticeable performance enhancements across various tasks using only a minor portion of exemplars (merely 10% of the total number of training data samples). This shows the power of carefully chosen exemplars in mitigating the impacts of perturbation. Nevertheless, the amount of recovery varies among the models and tasks. In particular, CodeT5 series

[3] https://github.com/tree-sitter

Tasks	Clone	Defect	Java to C#		C# to Java		Code Summarization
Metrics	F1	Accuracy	BLEU	EM	BLEU	EM	BLEU (Averaged)
Fine-Tuning with Perturbation							
GraphCodeBERT	94.75 $_{-0.25}$	61.86 $_{-1.02}$	62.75 $_{-17.83}$	46.30 $_{-13.10}$	55.33 $_{-17.31}$	48.00 $_{-18.80}$	17.59 $_{-0.54}$
PLBART	93.99 $_{+0.39}$	62.48 $_{-0.40}$	76.60 $_{-6.42}$	55.00 $_{-9.60}$	73.31 $_{-5.04}$	60.70 $_{-4.30}$	17.84 $_{-0.48}$
CodeT5	95.16 $_{+0.16}$	63.03 $_{-2.75}$	79.71 $_{-4.32}$	60.40 $_{-5.50}$	75.33 $_{-4.54}$	62.80 $_{-4.10}$	19.17 $_{-0.38}$
UniXcoder	89.77 $_{-1.59}$	60.94 $_{-1.40}$	72.17 $_{-6.78}$	57.40 $_{-5.90}$	67.11 $_{-7.11}$	58.20 $_{-5.40}$	18.71 $_{-0.52}$
Performance Recovery: Random Samples							
GraphCodeBERT	94.38 $_{-0.37}$	60.91 $_{-0.95}$	62.97 $_{+0.22}$	50.10 $_{+3.80}$	57.64 $_{+2.31}$	48.30 $_{+0.30}$	17.44 $_{-0.15}$
PLBART	93.29 $_{-0.70}$	61.16 $_{-1.32}$	78.91 $_{+2.31}$	56.40 $_{+1.40}$	76.17 $_{+2.86}$	61.70 $_{+1.00}$	17.68 $_{-0.16}$
CodeT5	95.46 $_{-0.30}$	60.11 $_{-2.92}$	80.11 $_{+0.40}$	61.00 $_{+0.60}$	74.34 $_{-0.99}$	62.10 $_{-0.70}$	19.04 $_{-0.13}$
UniXcoder	89.40 $_{-0.37}$	59.88 $_{-1.06}$	72.92 $_{+0.75}$	57.60 $_{+0.20}$	67.19 $_{+0.08}$	59.10 $_{+0.90}$	18.88 $_{+0.17}$
Performance Recovery: Exemplars							
GraphCodeBERT	94.36 $_{-0.39}$	62.31 $_{+0.45}$	70.10 $_{+7.35}$	52.50 $_{+6.20}$	59.15 $_{+3.82}$	47.30 $_{-0.70}$	17.66 $_{+0.07}$
PLBART	94.10 $_{+0.11}$	62.78 $_{+0.30}$	79.91 $_{+3.31}$	57.40 $_{+2.40}$	76.54 $_{+3.23}$	62.20 $_{+1.50}$	17.94 $_{+0.10}$
CodeT5	95.18 $_{+0.02}$	63.20 $_{+0.17}$	81.49 $_{+1.78}$	62.60 $_{+2.20}$	76.98 $_{+1.65}$	64.20 $_{+1.40}$	19.23 $_{+0.06}$
UniXcoder	90.37 $_{+0.60}$	61.15 $_{+0.21}$	73.40 $_{+1.23}$	58.80 $_{+1.40}$	67.77 $_{+0.66}$	59.10 $_{+0.90}$	19.10 $_{+0.39}$

TABLE II

THE IMPACT OF PERTURBATION ON THE MODEL'S PERFORMANCE IS FIRST PRESENTED (COMPARING WITH NORMAL FINE-TUNING), THEN IT DEMONSTRATES THE MODEL'S PERFORMANCE CAN BE SWIFTLY RECOVERED BY FINE-TUNING WITH A SMALL NUMBER OF EXEMPLARS (COMPARING WITH RANDOM SAMPLES). DUE TO SPACE LIMITATIONS, HERE WE PRESENT THE MEAN PERFORMANCE OF THE CODE SUMMARIZATION.

perform the best in this regard, often achieving improvements over their perturbed performance, and occasionally even showing significant leaps. When it comes to the two code generation tasks, the performance recovery is conspicuous, suggesting that well-chosen exemplars can effectively guide models back to generate coherent sequences.

C. Exemplars-based Efficient Fine-Tuning

Fine-tuning pre-trained language models is quite expensive in terms of computational cost and GPU memory. Therefore, we employ the exemplars for fine-tuning with the aim of achieving comparable performance to full fine-tuning. We consider two scenarios for fine-tuning, namely using exemplars equivalent to 10% and 20% of the training set size.

The performance of exemplar-based efficient fine-tuning in code understanding and code generation tasks is demonstrated in Table III and Table IV, respectively. The effectiveness of fine-tuning with exemplars has been validated in both scenarios. Among them, the performance gains provided by different tasks vary, with generation tasks generally exhibiting greater performance gains than tasks focused on code understanding. When utilizing 20% random training data, CodeT5 (trained with dataflow information) outperforms other comparable models, underscoring its innate ability to capture key information from a modest amount of data.

Moreover, the performance gain is further enhanced as the scale of our model increases, thereby validating the scalability of structural information utilization. This improvement can be partially attributed to the enhanced model capacity, which allows for effective learning of the underlying structures from more complex samples. In summary, the use of exemplars for training steadily contributes to the overall performance improvement across all methods on both tasks. Besides, it can be observed that the CodeT5 series with a shallow encoder is capable of capturing more structural information for downstream tasks.

Tasks	Clone	Defect
Metrics	F1	Accuracy
10% Training Data (Random)		
GraphCodeBERT	91.28	58.75
PLBART	91.93	60.43
CodeT5	91.62	60.58
UniXcoder	87.26	59.11
10% Training Data (Exemplars)		
GraphCodeBERT	91.96 $_{+0.68}$	59.22 $_{+0.47}$
PLBART	92.62 $_{+0.69}$	60.54 $_{+0.11}$
CodeT5	92.51 $_{+0.89}$	60.63 $_{+0.05}$
UniXcoder	87.37 $_{+0.11}$	59.40 $_{+0.29}$
20% Training Data (Random)		
GraphCodeBERT	94.17	60.32
PLBART	94.01	60.96
CodeT5	95.13	61.04
UniXcoder	88.58	59.35
20% Training Data (Exemplars)		
GraphCodeBERT	94.29 $_{+0.12}$	60.67 $_{+0.35}$
PLBART	94.15 $_{+0.14}$	61.10 $_{+0.14}$
CodeT5	91.48 $_{-0.03}$	61.12 $_{+0.08}$
UniXcoder	88.79 $_{+0.22}$	59.48 $_{+0.13}$

TABLE III

EFFECTIVENESS OF USING EXEMPLARS FOR DOWNSTREAM TASKS ADAPTATION ON CODE UNDERSTANDING TASKS ON VARIOUS BACKBONES.

It's worth noting that under our scenario of fine-tuning with 20% of exemplars, backbones like PLBART and CodeT5 can achieve performance on two code generation tasks nearly equivalent to full fine-tuning, which requires a much larger dataset. This further illustrates the efficacy of our method which is capable of extracting useful information from a smaller set of highly representative data.

Methods	Java to C#		C# to Java		Code Summarization					
	BLEU	EM	BLEU	EM	Ruby	JavaScript	Go	Python	Java	PHP
10% Training Data (Random)										
GraphCodeBERT	79.81	59.60	75.00	59.00	11.99	15.06	18.43	19.15	18.57	25.22
PLBART	82.28	59.70	78.51	64.50	13.01	15.66	18.60	19.52	18.89	23.83
CodeT5	83.79	65.20	78.35	65.30	15.22	15.12	19.06	19.20	19.32	24.95
UniXcoder	78.09	62.10	74.67	63.60	14.59	16.12	18.71	19.62	20.31	25.84
10% Training Data (Exemplars)										
GraphCodeBERT	80.19 +0.38	60.00 +0.40	75.29 +0.29	60.20 **+1.20**	11.90 -0.09	15.23 +0.17	18.53 +0.08	19.18 +0.03	18.86 +0.29	25.41 +0.19
PLBART	82.72 +0.44	60.90 **+1.20**	78.88 +0.37	64.20 -0.30	13.47 +0.46	16.57 +0.91	19.03 +0.20	19.55 +0.03	19.09 +0.20	23.90 +0.07
CodeT5	84.40 +0.61	65.70 +0.50	79.17 +0.82	65.90 +0.60	15.48 +0.26	16.22 **+1.10**	19.57 +0.51	19.96 +0.76	20.38 **+1.06**	26.09 **+1.14**
UniXcoder	78.79 +0.70	63.00 +0.90	74.78 +0.11	65.00 **+1.40**	14.81 +0.22	16.14 +0.02	18.82 +0.03	19.73 +0.11	20.49 +0.18	25.92 +0.08
20% Training Data (Random)										
GraphCodeBERT	79.97	60.10	75.14	60.30	12.07	14.82	18.45	19.04	18.73	25.11
PLBART	81.91	60.30	78.17	63.70	13.12	15.33	18.83	19.45	18.83	23.45
CodeT5	84.01	65.00	78.34	64.00	15.23	16.01	19.44	19.91	20.38	25.51
UniXcoder	78.12	62.40	77.23	64.80	14.95	15.70	18.79	19.57	19.71	24.91
20% Training Data (Exemplars)										
GraphCodeBERT	80.20 +0.23	60.50 +0.40	74.85 -0.29	59.60 -0.70	11.87 -0.20	15.56 +0.74	18.66 +0.21	19.19 +0.15	18.66 -0.07	25.35 +0.24
PLBART	82.91 **+1.00**	61.50 **+1.20**	78.80 +0.62	64.50 +0.80	13.42 +0.30	15.96 +0.63	18.87 +0.04	19.49 +0.04	19.28 +0.45	23.75 +0.30
CodeT5	84.33 +0.32	65.50 +0.50	80.10 **+1.76**	67.40 **+3.40**	15.60 +0.37	16.08 +0.07	19.45 +0.01	19.82 -0.09	20.42 +0.04	26.14 +0.63
UniXcoder	78.97 +0.85	63.20 +0.80	77.30 +0.07	65.10 +0.30	14.99 +0.04	16.05 +0.35	18.84 +0.05	19.77 +0.20	20.14 +0.43	25.80 +0.89

TABLE IV
COMPARATIVE ANALYSIS OF USING SELECTED EXEMPLARS FOR DOWNSTREAM TASKS ADAPTATION ON CODE TRANSLATION AND CODE SUMMARIZATION ACROSS FOUR CODEPTMS. WE CONDUCT COMPARISONS BETWEEN NORMAL FINE-TUNING AND EXEMPLAR-BASED FINE-TUNING USING DATA THAT MAKEUP ONLY 10% AND 20% OF THE TOTAL TRAINING SET SAMPLE SIZE, RESPECTIVELY.

D. Further Analysis

We conduct the following studies to verify the effectiveness of components in our methods.

a) Effectiveness of Exemplars: Considering that exemplars encapsulate "rich" code structural information, there is potential to achieve performance comparable to, or even surpassing, full fine-tuning with reduced computational overhead. Here we compare the results obtained utilizing varying quantities of exemplars against those achieved with the entire dataset. As presented in Table V, it is evident that fine-tuning based on exemplars requires only 20% of the entire dataset's volume to nearly match the performance achieved with full fine-tuning. Remarkably, When we utilize exemplar data equivalent to 30% of the whole dataset, the performance surpasses full fine-tuning on certain metrics. This enhancement could partially be attributed to our method's innate capacity to sift through and exclude low-quality samples from the training set, effectively mitigating the potential interference of noise in the model's learning trajectory.

b) Effectiveness of U-AST: As mentioned in Section IV-A, we employ a variant of AST to compute global effectiveness, which aids in solving the problem of poor connectivity among leaf nodes. Here we select exemplars in the same manner while using raw ASTs and then make comparisons to confirm the necessity, As shown in Table VI, it is evident that incorporating U-AST significantly improves CodeBLEU. This improvement underscores its capability to exploit the code structure, aiding the model in generating code that adheres more closely to established norms.

Manifestly, in code translation tasks (C# ↔ Java) that have a higher correlation with understanding the structure of the code, models trained on our exemplars selected based on U-AST consistently yield significantly better performance.

Tasks	Code Translation (Avg.)			Code Summarization
Metrics	BLEU	EM	CodeBLEU	BLEU (Avg.)
CodeGen				
Full Fine-Tuning	80.46	65.70	81.42	20.57
10% Exemplars	77.07	64.05	79.25	19.95
20% Exemplars	80.07	65.80	81.46	20.35
30% Exemplars	**80.89**	**67.20**	**81.78**	**20.76**
CodeT5+				
Full Fine-Tuning	**83.97**	63.91	84.52	**21.75**
10% Exemplars	81.24	65.40	83.29	20.62
20% Exemplars	82.00	**66.60**	85.19	21.08
30% Exemplars	83.46	66.45	**85.61**	21.56

TABLE V
COMPARING FULL FIN-TUNING WITH EXEMPLAR-BASED EFFICIENT TUNING, WHERE THE PERCENTAGES INDICATE THE PROPORTION OF EXEMPLAR DATA USED RELATIVE TO THE TOTAL NUMBER OF SAMPLES.

Tasks	Code Translation (Avg.)			Code Summarization
Metrics	BLEU	EM	CodeBLEU	BLEU (Avg.)
CodeT5				
Raw AST	81.47	64.70	82.39	19.06
U-AST	**81.75**	**65.20**	**85.71**	**19.18**
UniXcoder				
Raw AST	77.79	63.42	80.98	19.22
U-AST	**78.05**	**64.50**	**83.94**	**19.31**

TABLE VI
ABLATION STUDIES ON THE EFFECTIVENESS OF USING U-AST FOR EXEMPLARS SELECTION ON EFFICIENT FINE-TUNING WITH CODET5 AND UNIXCODER BACKBONES.

VI. Related Works

a) Code Pre-trained Models: Pre-trained language models have revolutionized the landscape of NLP (39; 40, *inter alia*). Recent works have started utilizing widely-recognized model architectures (1) and leveraging pretraining strategies for code pre-trained models (CodePTMs). CuBERT (41) first pre-train the BERT model on a large-scale Python corpus. Then, GraphCoddeBERT (42) utilize masked language model (MLM) and replaced token detection (RTD) tasks to train CodeBERT in six programming languages. GraphCodeBERT (11) is a variant that integrates structural information to facilitate code representation learning. Besides the models with encoder architecture, CodeT5 (31), PLBART (30) and UniXcoder (15) are pre-trained based on encoder-decoder architecture with multi-task training strategies. These models bring forth new possibilities for enhancing performance across diverse code intelligence tasks. Recently, CodeGen (43) release a series of models across different scales for program synthesis, filling a gap between larger and smaller CodePTMs.

b) Textual Perturbations: It is a prevalent understanding that the performance of language models may be compromised when dealing with noisy data in real-world scenarios. Perturbation experiments are always involved to detect unintended model biases (44). Input perturbations are employed to evaluate whether pre-trained models' performance will be hampered when small changes are introduced (45). Recently, quantitative measures (26) have been proposed to explain why certain models demonstrate less robustness to some perturbations than others. We draw inspiration from these perturbation strategies, and apply them to source code data without resorting to methods such as adversarial sample generation (46).

c) Analyzing CodePTMs: Considering the differences between code and NL, analysis of CodePTMs generally focuses on structural information. Probing techniques (17) are first involved in explaining the models' behaviours. Then, Diagnostic tasks (18) are designed about code syntactic structure. After that, qualitative analyses are conducted to evaluate how CodePTMs interpret code structure (19). CAT-probing (20) innovatively establishes the relationship among AST, attention mechanism and downstream tasks. Beyond understanding the internal mechanisms. After that, diverse properties of source code, *i.e.*, lexical, syntactic, and structural information encoded in different layers of CodePTMs are discovered (47). Moreover, evidence is presented that models trained on code can comprehend meanings, despite their training being confined to tasks such as next token prediction (48; 49).

VII. Conclusion

In this paper, we propose a collection of novel and practical methods to delve into and leverage the structural information within the code. Perturbation experiments of varying degrees are first conducted to explore the impact of structural information on code representation learning. Subsequently, we further scrutinize the phenomenon by examining the impact of dataflow information from AST. Inspired by these insights, we implement a novel metric to measure code structure information and employ it to carry out exemplar selection. By training on a minimal set of exemplars with rich structure information, we not only recover the performance of models affected by noise but also enable them to perform efficient fine-tuning, which significantly reduces the computational cost and data volume requirements for downstream tasks. Finally, we achieve performance close to or even surpassing full fine-tuning on generation tasks with minimal cost, indicating the effectiveness of our utilization of structural information.

References

[1] X. Qiu, T. Sun, Y. Xu, Y. Shao, N. Dai, and X. Huang, "Pre-trained models for natural language processing: A survey," *SCIENCE CHINA Technological Sciences*, 2020.

[2] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. T. Devanbu, "On the naturalness of software," *Commun. ACM*, 2016.

[3] L. Buratti, S. Pujar, M. Bornea, S. McCarley, Y. Zheng, G. Rossiello, A. Morari, J. Laredo, V. Thost, Y. Zhuang, and G. Domeniconi, "Exploring software naturalness through neural language models," 2020.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. of NeurIPS*, 2017.

[5] T. Lin, Y. Wang, X. Liu, and X. Qiu, "A survey of transformers," *AI Open*, 2022.

[6] Y. Xu and Y. Zhu, "A survey on pretrained language models for neural code intelligence," 2022.

[7] D. Zan, B. Chen, F. Zhang, D. Lu, B. Wu, B. Guan, Y. Wang, and J.-G. Lou, "Large language models meet nl2code: A survey," 2023.

[8] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," 2019.

[9] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement *et al.*, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," 2021.

[10] Q. Sun, N. Chen, J. Wang, X. Li, and M. Gao, "Transcoder: Towards unified transferable code representation learning inspired by human skills," 2023.

[11] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. LIU, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "GraphCodeBERT: Pre-training code representations with data flow," in *ICLR*, 2021.

[12] D. Zügner, T. Kirschstein, M. Catasta, J. Leskovec, and S. Günnemann, "Language-agnostic representation learning of source code from structure and context," in *Proc. of ICLR*, 2021.

[13] H. Peng, G. Li, W. Wang, Y. Zhao, and Z. Jin, "Integrating tree path in transformer for code representation," *Proc. of NeurIPS*, 2021.

[14] X. Wang, Y. Wang, F. Mi, P. Zhou, Y. Wan, X. Liu, L. Li, H. Wu, J. Liu, and X. Jiang, "SynCoBERT: Syntax-guided multi-modal contrastive pre-training for code representation," in *arXiv*, 2021.

[15] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "UniXcoder: Unified cross-modal pre-training for code representation," in *Proc. of ACL*, 2022.

[16] H. Peng, G. Li, Y. Zhao, and Z. Jin, "Rethinking positional encoding in tree transformer for code representation," in *Proc. of EMNLP*, 2022.

[17] A. Karmakar and R. Robbes, "What do pre-trained code models know about code?" in *ASE*, 2021.

[18] S. Troshin and N. Chirkova, "Probing pretrained models of source codes," in *Proceedings of the Fifth BlackboxNLP Workshop*, 2022.

[19] Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu, and H. Jin, "What do they capture? a structural analysis of pre-trained language models for source code," in *Proc. of ICSE*, 2022.

[20] N. Chen, Q. Sun, R. Zhu, X. Li, X. Lu, and M. Gao, "CAT-probing: A metric-based approach to interpret how pre-trained models for programming language attend code structure," in *Proc. of EMNLP Findings*, 2022.

[21] F. Xu, Q. Lin, J. Han, T. Zhao, J. Liu, and E. Cambria, "Are large language models really good logical reasoners? a comprehensive evaluation from deductive, inductive and abductive views," 2023.

[22] K. Rayner, A. Pollatsek, J. Ashby, and C. Clifton Jr, *Psychology of reading*. Psychology Press, 2012.

[23] J. Hessel and A. Schofield, "How effective is BERT without word ordering? implications for language understanding and data privacy," in *Proc. of ACL*, 2021.

[24] N. Chen, Q. Sun, J. Wang, M. Gao, X. Li, and X. Li, "Evaluating and enhancing the robustness of code pre-trained models through structure-aware adversarial samples generation," in *Proc. of EMNLP Findings*, 2023.

[25] N. Chen, Q. Sun, J. Wang, X. Li, and M. Gao, "Pass-tuning: Towards structure-aware parameter-efficient tuning for code representation learning," in *Proc. of EMNLP Findings*, 2023.

[26] Y. Zhang, L. Pan, S. Tan, and M.-Y. Kan, "Interpreting the robustness of neural NLP models to textual perturbations," in *Proc. of ACL Findings*, 2022.

[27] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *SANER*, 2020.

[28] R. Zhu, L. Yuan, X. Li, M. Gao, and W. Cai, "A neural network architecture for program understanding inspired by human behaviors," in *Proc. of ACL*, 2022.

[29] V. Latora and M. Marchiori, "Efficient behavior of small-world networks," *Physical review letters*, 2001.

[30] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proc. of AACL*, 2021.

[31] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proc. of EMNLP*, 2021.

[32] U. Alon, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *Proc. of ICLR*, 2019.

[33] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Divide-and-conquer approach for multi-phase statistical migration for source code (t)," in *ICASE*, 2015.

[34] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," 2014.

[35] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," 2016.

[36] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proc. of NeurIPS*, 2019.

[37] J. Wang, N. Chen, Q. Sun, W. Huang, C. Wang, and M. Gao, "HugNLP: A unified and comprehensive library for natural language processing," in *Proc. of CIKM*, 2023.

[38] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," 2020.

[39] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. of NAACL*, 2019.

[40] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," 2019.

[41] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *Proc. of ICML*, 2020.

[42] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. of EMNLP Findings*, 2020.

[43] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," in *Proc. of ICLR*, 2023.

[44] V. Prabhakaran, B. Hutchinson, and M. Mitchell, "Perturbation sensitivity analysis to detect unintended model biases," in *Proc. of EMNLP*, 2019.

[45] M. Moradi and M. Samwald, "Evaluating the robustness of neural language models to input perturbations," in *Proc. of EMNLP*, 2021.

[46] J. Zhang, W. Ma, X. Xie, Q. Hu, and Y. Liu, "Black-box adversarial attack guided by model behavior for programming pre-trained language models," 2023.

[47] E. Shi, Y. Wang, H. Zhang, L. Du, S. Han, D. Zhang, and H. Sun, "Towards efficient fine-tuning of pre-trained code models: An experimental study and beyond," in *ISSTA*, 2023.

[48] A. Jha and C. K. Reddy, "Codeattack: Code-based adversarial attacks for pre-trained programming language models," *Proc. of AAAI*, 2023.

[49] C. Jin and M. Rinard, "Evidence of meaning in language models trained on programs," 2023.