A Critical Study of What Code-LLMs (Do Not) Learn

Abhinav Anand[1,a], Shweta Verma[1,a], Krishna Narasimhan[*2,b], and Mira Mezini[1,3,4,c]

[1]Technische Universität Darmstadt
[2]AI Quality & Testing Hub
[3]Hessian Center for Artificial Intelligence (hessian.AI)
[4]National Research Center for Applied Cybersecurity ATHENE
[a]{abhinav.anand, shweta.verma}@tu-darmstadt.de
[b]k.narasimhan@aiqualityhub.com
[c]mezini@cs.tu-darmstadt.de

Abstract

Large Language Models trained on code corpora (code-LLMs) have demonstrated impressive performance in various coding assistance tasks. However, despite their increased size and training dataset, code-LLMs still have limitations such as suggesting codes with syntactic errors, variable misuse etc. Some studies argue that code-LLMs perform well on coding tasks because they use self-attention and hidden representations to encode relations among input tokens. However, previous works have not studied what code properties are not encoded by code-LLMs. In this paper, we conduct a fine-grained analysis of attention maps and hidden representations of code-LLMs. Our study indicates that code-LLMs only encode relations among specific subsets of input tokens. Specifically, by categorizing input tokens into syntactic tokens and identifiers, we found that models encode relations among syntactic tokens and among identifiers, but they fail to encode relations between syntactic tokens and identifiers. We also found that fine-tuned models encode these relations poorly compared to their pre-trained counterparts. Additionally, larger models with billions of parameters encode significantly less information about code than models with only a few hundred million parameters.

1 Introduction

Code-LLMs (cLLMs) are Transformer models (Vaswani et al., 2017) trained on a large corpus of code and natural language - programming language (NL-PL) pairs. These models are used, either in a zero-shot manner or after fine-tuning, for coding assistance tasks, including code summarization, code retrieval, code completion, code generation, and program repair (Xu and Zhu, 2022).

While the performance of models on benchmarks has significantly improved in the past few years, there are still issues with performance in real-world settings. Code generated by models has compilation errors due to syntactical mistakes (Le et al., 2022), semantic errors like random identifiers (Guo et al., 2021), and can invoke undefined or out-of-scope functions, variables and attributes (Chen et al., 2021). Some studies suggest that models do not generalize well (Hajipour et al., 2022; Hellendoorn et al., 2019), learn shortcuts (Sontakke et al., 2022; Rabin et al., 2021), and memorize training inputs (Rabin et al., 2023a; Yang et al., 2023b). To understand the cause of these issues, it is imperative to understand which code properties are used by cLLMs for prediction and generation and which are not encoded by cLLM. But the black-box nature of neural networks makes this understanding a challenging task.

Prior studies have used attention analysis (Wan et al., 2022) and probing on hidden representation (Belinkov, 2022) to study what cLLMs encode. Some of these studies show that models can learn the syntactic and semantic structure of code (Wan et al., 2022; Troshin and Chirkova, 2022; López et al., 2022) and understand code logic (Baltaji and Thakkar, 2023). However, they rely on non-systematically validated assumptions. For example, studies on attention analysis set an arbitrary attention threshold of 0.3. The studies which probe hidden representation of code models assume a linear encoding of information. The effect of these assumptions has hitherto remained unstudied. Further, these studies do not evaluate which code properties are not encoded by cLLMs. In this paper, we make two important contributions to advance the state of the art in the interpretability of cLLMs.

First, we perform a systematic analysis of assumptions in previous work and show that they can lead to misleading conclusions. Specifically, we examine the influence of the attention threshold and evaluation metric on attention analysis, and

[*]Work conducted while afiliated with Technische Universität Darmstadt.

Findings of the Association for Computational Linguistics ACL 2024, pages 15869–15889

for probing on hidden representation, we explore whether the code relations among tokens are encoded linearly or non-linearly. To avoid several limitations of classifier and structural probing methods (Maudslay et al., 2020; Hewitt and Liang, 2019; Belinkov, 2022), we perform probing of hidden representation without any additional classifier layers or parameters. Based on our observations, we make some new suggestions for experimental setup of analysis of attention maps and hidden representation.

Second, armed with our insights from the first analysis, we set up and perform a fine-grained analysis of attention and hidden representation of cLLMs at the code token level to critically examine what they learn and do not learn. Previous studies examining the code comprehension ability of cLLMs have analyzed all input tokens together, without distinguishing between different categories of code tokens such as identifiers (e.g., function names, variables) and syntactic tokens (e.g., keywords, operators, parentheses). To investigate whether there are specific relations that cLLMs fail to encode, we separately analyze the syntactic-syntactic, identifier-identifier, and syntactic-identifier relations that are encoded in the self-attention values and hidden representations.

There are different types of relations between code tokens, including relations in an abstract syntax tree (AST), as well as, data flow or control flow relations between code blocks. Similar to Wan et al. (2022), we focus primarily on syntactic relations in the AST and create a syntax graph with edges between code tokens within a motif structure (Figure 6b). But such a syntax graph does not encompass all the relations among identifiers, in particular how values flow from one variable to another. Thus, we extend the study to data-flow relations and create a data flow graph (DFG) with edges among related variables following Guo et al. (2021).

We perform attention analysis to study whether a token pays attention to related tokens and analysis of hidden representation to study the information encoded by the model in the vector representation of a token. To study information encoded in hidden representations, we take hidden representations of pairs of tokens and evaluate if the information encoded by the model is sufficient to predict the relation between these two tokens. Specifically, we evaluate with respect to predicting edges in a DFG and sibling and distance prediction in an AST.

We study models with 110M to 3.7B parameters with different architectures, pre-training objectives, and training datasets [1]. In summary,

- We provide evidence that prior work often made incorrect assumptions in their experimental settings, which led to misleading conclusions. In particular, previous works on attention analysis assume an attention threshold of 0.3 and study heads with best precision (shown in Figure 1). Also, the studies on hidden representation assume linear encoding of information in hidden representation.

- The attention maps of cLLMs fall short in encoding syntactic-identifier relations, while they do encode syntactic-syntactic and identifier-identifier relations. Also, the hidden representations of cLLMs do not encode sufficient information to discriminate between different identifier types and to understand subtle syntactical differences.

- We show that the issues of cLLMs with encoding code syntax persists for big models with significantly increased number of parameters or for models that are fine-tuned on specific tasks. In fact, we observe a reduction in encoding code syntax and even data-flow relations with large size and fine-tuning.

2 Related Work

Several studies have provided some possible explaination of the working of cLLMs. Cito et al. (2022) and Rabin et al. (2023b) used input perturbation, while, Liu et al. (2023) used backpropagation to find the most relevant input tokens. Zhang et al. (2022a) created an aggregated attention graph and studied its application to the `VarMiuse` task. Wan et al. (2022) performed attention analysis and probing with structural probes (Hewitt and Manning, 2019). López et al. (2022) used structural probe to create binarized AST from hidden representations.

Probing classifiers have been used to test syntax and semantic understanding (Karmakar and Robbes, 2021; Troshin and Chirkova, 2022; Ahmed et al., 2023), the effect of positional embeddings (Yang et al., 2023a), relation between self-attention and distance in AST (Chen et al., 2022) and logic understanding (Baltaji and Thakkar, 2023).

[1]The code is available at `https://github.com/stg-tud/code-LLM-critical-evaluation`.

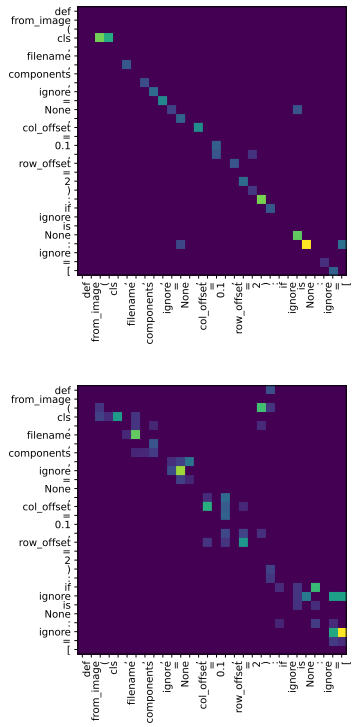


Figure 1: Attention map for head with best precision (head 1) **(top)** and head with best f-score (head 2) **(bottom)** of layer 9 of CodeBERT for first 30 tokens of a python code (see Figure 6a for code). The head with best precision mostly encodes next-token attention, while head with best f-score encodes more complex relation.

Other studies established correlations between input tokens, model output, and self-attention. Bui et al. (2019) created an attention-based discriminative score to rank input tokens and studied the impact of high-ranked tokens on output. Attention-based token selection was utilized by Zhang et al. (2022b) to simplify the input program of CodeBERT (Feng et al., 2020). Rabin et al. (2021) and Rabin et al. (2022) simplified the input program while preserving the output and showed that the percentage of common tokens between attention and reduced input programs is typically high.

Our Work studies the limitations of code models in encoding code structure which has hitherto remained unexplored. Our study spanning multiple transformer architectures, sizes and training objectives demonstrate a significant gap in encoding some code properties. This gap could be a possible explanation for poor performance of cLLMs on real-world tasks (Hellendoorn et al., 2019).

3 Experiments

In this section, we elaborate on the experiments that we performed to analyze self-attention and the hidden representation of cLLMs. For attention analysis, we compare the self-attention of models with the motif structure in a program's AST and DFG. For hidden representations, we perform probing without classifiers using DirectProbe (Zhou and Srikumar, 2021). We provide details on AST, DFG, DirectProbe, and motif structure in Appendix B.

3.1 Models and Dataset

We analyze a wide range of pre-trained and fine-tuned models. The parameters range from 110M to 3.7B. The investigated models also have different architectures, training datasets, and objectives.

Among the subjects there are the encoder-only models such as CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2021)), encoder-decoder models such as CodeT5 (Wang et al., 2021), PLBART (Ahmad et al., 2021) and CodeT5+ (Wang et al., 2023), and decoder-only models. CodeGen (Nijkamp et al., 2023) is a decoder-only model trained with fill-in-the-middle objective (Bavarian et al., 2022) for bi-directional context while UnixCoder with encode-decoder architecture (Guo et al., 2022) has a UniLM-style (Dong et al., 2019) training.

We also investigate models with different objectives. CodeT5-musu (Wang et al., 2021) is fine-tuned for summarization tasks, CodeT5+220Mbi (Wang et al., 2023) can be used in a zero-shot manner for summarization and retrieval tasks, and CodeRL (Le et al., 2022) is a larger CodeT5 model (CodeT5_lntp) trained for code generation in an actor-critic setup using test cases for reward.

For our experiments, we randomly sampled 3000 Python codes from the test set of CodeSearchNet dataset (Husain et al., 2019) after removing docstrings and comments. More details about the models and data preparation are presented in Appendix C and Appendix D respectively.

3.2 Attention Analysis

3.2.1 Setup

Model graph. The attention map of a head is a $n * n$ matrix (n is the number of input tokens). The elements of the matrix represent the significance each token attributes to other tokens. We consider the matrix as the adjacency matrix of a graph with input tokens corresponding to nodes and attention

values inducing an edge. Similar to previous works on attention analysis (Wan et al., 2022; Zhang et al., 2022a), we merge the sub-tokens of input code tokens by averaging their attention values.

We considered the edges of the model graphs as predictions and that of code graphs (defined later) as the ground truth in the computation of precision and recall.

Prior studies have typically set an arbitrary threshold of 0.3 for attention analysis and have excluded heads with very few attention values, usually less than 100, from the analysis (Wan et al., 2022; Vig et al., 2021). This approach excludes more than 99.5% of self-attention values (see Appendix E), thereby skewing the conclusions drawn. For instance, Wan et al. (2022) reported high precision values, indicating that the majority of attention values correspond to relations in the AST. However, we observe a significantly reduced recall, as shown in Figure 2. The low recall shows that only a small proportion of syntactic relations are encoded in attention values greater than 0.3. Further, a code token is always syntactically related to the next token, unless there is a line break in between. Consequently, encoding next token attention results in high precision. As shown in Figure 1, the heads with best precision often only encode next-token attention. On the other hand, heads with best f-score encode more relations such as attention paid to tokens other than the next-token.

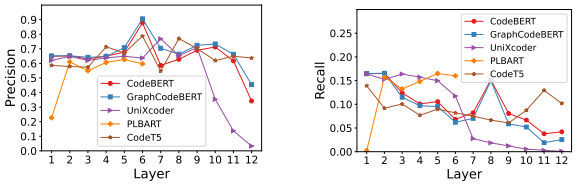
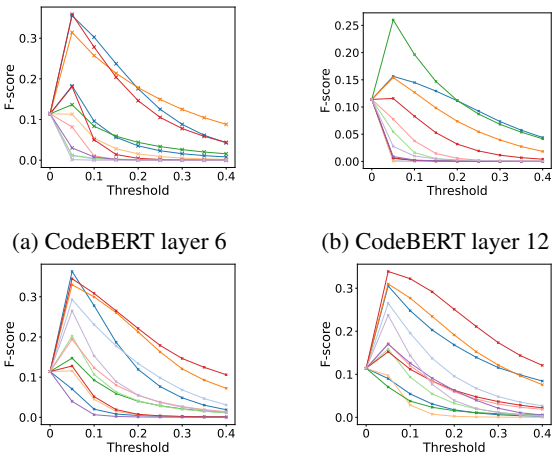


Figure 2: On comparing model graph with syntax graph with an attention threshold of 0.3, the precision (left) is high but the recall is very low (right).

So, to balance between precision and recall, we use F-score. We evaluate F-scores for all heads across various models and layers at different threshold values. As shown in Figure 3, the highest F-score is achieved when using a threshold of 0.05. We use this threshold for all experiments. Similar to previous works (Wan et al., 2022), we set all values below the threshold to 0 and those above to 1. That is, we don't weight the calculations with actual self-attention values. Such a weighting will lower the precision and recall and increase the graph edit distance per node (Section 3.2.2). Setting values to 1 refers to the best-case scenario. Thus, the limitations documented in this work exist even in the best-case scenario. Weighing with original values will only make these limitations more stark without changing the conclusion.

Code graphs. We compare the *model graph* with two *code graphs*: the *syntax graph*, representing relations in an AST, and the *DFG graph*. The syntax graph comprises syntactic relations among all tokens, while the DFG comprises data flow relations among identifiers. Following Wan et al. (2022), we assume two tokens to have a syntactic relation if they exist in the same motif structure (see Appendix B). Since we want to study the encoding of syntactic-syntactic, identifier-identifier, and syntactic-identifier relations separately, we create a *non-identifier graph* with the same nodes as the syntax graph but only encompassing AST relations between syntactic tokens.



(a) CodeBERT layer 6 (b) CodeBERT layer 12

(c) CodeT5 layer 6 (d) CodeT5 layer 12

Figure 3: The plot illustrates F-score between model graph and syntax graph at different thresholds for all heads. Each curve in a plot represents one head. The plots for layer 6 and layer 12 of CodeBERT and CodeT5 are shown out of various models and layers evaluated at different thresholds. For most heads, F-score is highest at a threshold of 0.05 for all models.

3.2.2 Analysis

For each model, we compare the model graph of a head with the code graphs in two ways.

First, we compute the precision and recall between the set of edges in the model graph and the code graphs. We consider the edges of the code graphs as ground truth and those of the model graphs as predictions. For comparison across lay-

ers of a model, we select the heads with the highest F-score for each layer.

Second, we calculate the graph edit distance (GED) (Sanfeliu and Fu, 1983) per node to quantify the similarity between code and model graphs. GED between two graphs G_1 and G_2 computes the cost of inserting, deleting, or substituting nodes and edges to transform G_1 into an isomorphic graph of G_2. Code graphs and model graphs share the same set of nodes and have only one edge type. So, we assign a cost of 1 for both edge deletion and insertion operations and 0 otherwise. In all calculations, we apply the operations to model graphs. We also calculate the GED between the model graph and the non-identifier graph. For GED calculations, we use the NetworkX package (Hagberg et al., 2008).

3.3 Analysis of Hidden Representations

3.3.1 Qualitative Analysis with t-SNE

The hidden representation, $\boldsymbol{h}_i^l$ of i^{th} word at the output of layer l, is a d-dimensional vector. We use t-SNE (van der Maaten and Hinton, 2008) – a widely used technique to project high-dimensional data into a two-dimensional space while preserving the distance distribution between points - to qualitatively analyze the hidden representations in two settings.

First, we study the distribution of hidden representations of different token types; to this end, we collect the hidden representations of code tokens of specific types from 100 programs, each having a minimum of 100 tokens.

Second, we compare the distance distribution between tokens in an AST and between their hidden representations. In the AST, siblings have similar distance distribution. So, in t-SNE visualization of AST tree distances, siblings cluster together. If the distance between hidden representations corresponds to the distance in the AST, hidden representations should also have a similar distance distribution. To this end, we construct distance matrices of both for randomly selected code samples.

3.3.2 Probing on Hidden Representations

We use DirectProbe (Zhou and Srikumar, 2021) to quantitatively evaluate the syntactic and data flow information encoded in hidden representations of each token for a given layer. We create datasets for each layer of the models we examined. Each data point is represented as $(\boldsymbol{h}_i^l * \boldsymbol{h}_j^l) : label_t$. $* \in \{concatenation, difference\}$ is an operation between hidden representations of tokens i and j of layer l. $t \in \{siblings, treedistance, dataflow\}$ is a task to evaluate whether hidden representations encode information about the specific property. Each dataset is split in a $80:20$ ratio into training and test sets. The training set is used to create clusters for each label and the test set is used to evaluate the quality of clustering.

Using $dataflow$, we study whether data flow relations are encoded. Here, both i and j are identifiers, $label \in \{NoEdge, ComesFrom, ComputedFrom\}$ and $* = concatenation$. Using $siblings$ and $treedistance$, we study the encoding of relations in an AST. For both tasks, token i is one of a subset of Python keywords (listed in Appendix H). In one set of experiments, (`Keyword-All`), token j can be any other token. In another set, (`Keyword-Identifier`), token j is an identifier. For the siblings task, $label \in \{sibling, notsibling\}$, where two tokens in the same motif structure are considered to be siblings, and $* = concatenation$. The minimum distance between two code tokens in an AST is 2 while tokens far apart in an AST don't have any discriminative syntactic relations. So, for tree distance, we only consider $label \in \{2, 3, 4, 5, 6\}$. Moreover, Reif et al. (2019) showed that square of distance between two vectors, $(\boldsymbol{h}_i^l - \boldsymbol{h}_j^l)^T(\boldsymbol{h}_i^l - \boldsymbol{h}_j^l)$, corresponds to distance in a tree. Hence, we set $* = difference$ for the distance prediction task.

The tree distance between a keyword and an identifier denotes different identifier types and syntax structures. For instance, consider the statements of the form (a) `if var1:` and (b) `if var1 == var2:`. The tree distance between `if` and `var1` is 2 in (a) and 3 in (b). In a function declaration, the identifier types function name, parameters, and default parameters are, respectively, at a distance of 2, 3 and 4 from `def`. Hence, if the hidden representations encode information about different identifier types and syntax, it follows that hidden representations of (`Keyword-Identifier`) pairs at a certain distance in AST must form separable clusters.

4 Results

4.1 Attention Analysis

In Figure 4 we present the recall between model graphs and code graphs. We observe that different models encode code relations to varying degrees. Surprisingly, fine-tuned and larger models do not encode a higher proportion of code relations com-

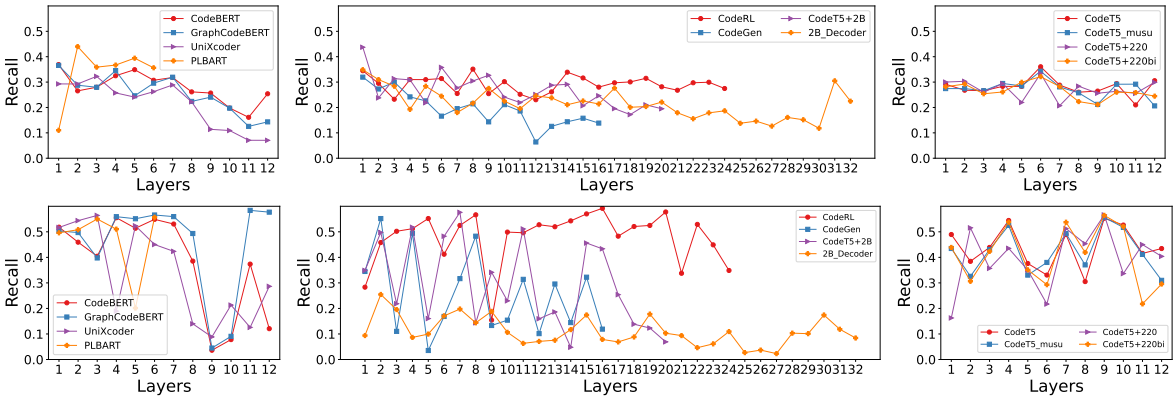


Figure 4: Recall of model graphs with syntax graphs (top) and data flow graphs (bottom). The plots show irrespective of training-objectives, fine-tuning or larger sizes, the models do not encode more than 40% of syntactic relations and around 55% of data flow relations. Enc-Dec models encode syntactic relations much better in deeper layers.

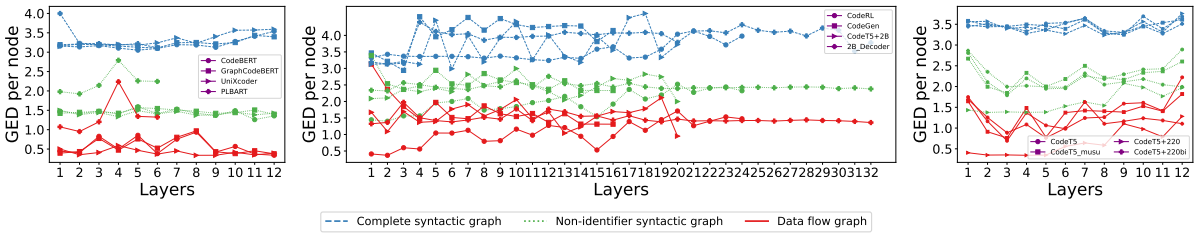


Figure 5: Graph edit distance (GED) per node (lower value show higher similarity) of model graph from DFG, non-identifier syntax graph and complete syntax graph for various models. The gap between non-identifier and complete syntax graph shows that on introducing syntax-identifier edges the similarity reduces drastically and thus, these edges are not present in the model graphs. For very large models (center), even DFG edges are encoded poorly.

pared to smaller pre-trained models, even if they perform better on benchmarks. Similarly, the actor-critic training of CodeRL does not improve encoding of code relation compared to CodeT5_lntp, even if it performs significantly better on code generation(Le et al., 2022). Further, the decoder-only CodeGen model with 3.7B parameters barely encodes code-relations in deeper layers.

We also find that the proportion of encoded relations degenerate in deeper layers of encoder-only models but not in encoder-decoder models. The degeneration in deeper layers of encoder-only models contradicts Wan et al. (2022), who concluded that the last two layers encode the syntactic relations better. Wan et al. (2022) uses a higher threshold (0.3) than in our work (0.05) and compares the heads with the best precision instead of those with the best F-score (our work). Our findings are consistent with the observations of Grishina et al. (2023), who utilize early layers of CodeBERT for improved and efficient classification.

Overall, in Figure 4 we find that the models we studied encode 30-40% of syntactic relations and around 50% of data flow relations. This means that the majority of the code relations are still not encoded within the self-attention values. This raises the question - what relations are not encoded and how important are they for code understanding? To study the limitations quantitatively, we measure the similarity between model graphs and code graphs. The results are presented in Figure 5.

For all models, we find that the model graph has the highest similarity with DFG. However, smaller encoder-decoder models and deeper layers of larger encoder-decoder models have lower DFG similarity compared to encoder-only models. Thus, encoder-only models encode data flow relations better than encoder-decoder models and very large models encode data flow relations very poorly.

When we study the syntax graphs in Figure 5, we observe that model graphs of all models across each layer have much higher similarity with non-identifier graphs than with complete syntactic graphs. This means that the syntactic-identifier token relations are not encoded in the model graph. The reasoning is as follows. The edges in complete syntax graph comprises of all edges in non-identifier graph and additional syntactic-identifier

edges. If these additional edges were present in the complete graph, the deletion cost and, hence, the overall cost for the complete syntax graph would have decreased. However, we observe a significant increase in cost per node, by a factor of 1.5-2. Thus, these additional edges relating syntactic and identifier tokens are not encoded in self-attention values, irrespective of model size and architecture. In fact, larger models encode syntactic relations poorly compared to smaller models.

4.2 Analysis of Hidden Representation

In our study of hidden representations using t-SNE, we find that the clustering of hidden representations does not follow syntactic relations in AST. In both the settings (hidden representation of tokens and distance matrix described in Section 3.3.1) we find that the hidden representations create clusters based on token types rather than on syntactic relations. Due to space constraints, we show the t-SNE projections in Appendix G.

In hidden representations (Figure 10), the clusters of syntactically related tokens such as, `def`, `(`, `)` and `:`, are not close to each other. But in distance matrix, certain syntactically related tokens do exist together. For the code in Figure 6a, we find that `def` is close to `(`, `)`, and `:` while `if` is close to `is` and `none` in the projection of fifth layer of CodeBERT (Figure 11). Similarly, `not` and `in` occur together. However, identifiers are far from syntactic tokens including the token `=`, which usually establishes relations among variables. We found similar patterns for deeper layers of all models, while all tokens cluster together in the first few layers.

These observations contradict previous studies that use classifier and structural probing (Troshin and Chirkova, 2022; Karmakar and Robbes, 2021; Ahmed et al., 2023; Wan et al., 2022). The previous works assume a linear encoding of information and hence, use a simple probe (Belinkov, 2022). The studies conclude that hidden representations can encode syntactic relations among tokens.

Using DirectProbe (see Appendix B), we study both, what information is encoded in hidden representation and how - linearly or non-linearly. We report the number of clusters and clustering accuracy for the last layer in Tables 1, 2 and 3 (See Appendix I for more layers and models). The number of clusters created by DirectProbe indicates whether the hidden representations encode a property linearly or non-linearly. Linear encoding results in the same number of clusters as the number of labels. For all three tasks, we observe a significantly higher number of clusters than labels across all models, usually twice as many. This means that hidden representations encode syntactic and data flow relations non-linearly. Thus, a simple probe is not sufficient to study hidden representation of cLLMs (Belinkov, 2022)

In case of pre-trained models, we find that DirectProbe forms clusters with high accuracy on siblings and data flow tasks (Tables 2 and 3). But, on the tree distance tasks shown in Table 1, the cluster accuracy is poor for $distacne > 2$

Tokens	Model	No. of clusters	Label Accuracy				
			2	3	4	5	6
{Keyword-All}	GraphCodeBERT	9	0.84	0.78	0.67	0.67	0.57
	CodeT5	10	0.83	0.79	0.70	0.64	0.60
	CodeT5+220M	11	0.78	0.67	0.58	0.65	0.58
	CodeT5+220Mbi	10	0.64	0.60	0.52	0.46	0.44
	CodeT5+770M	9	0.76	0.70	0.58	0.61	0.58
	CodeRL	13	0.67	0.67	0.62	0.67	0.55
	Codegen	11	0.61	0.65	0.56	0.54	0.48
	CodeT5+2B	9	0.63	0.66	0.47	0.55	0.47
{Keyword-Identifier}	GraphCodeBERT	7	0.79	0.68	0.52	0.57	0.49
	CodeT5	6	0.78	0.66	0.59	0.55	0.48
	CodeT5+220M	7	0.82	0.73	0.65	0.61	0.52
	CodeT5+220Mbi	7	0.65	0.55	0.51	0.43	0.41
	CodeT5+770M	5	0.75	0.69	0.61	0.59	0.53
	CodeRL	5	0.67	0.63	0.55	0.53	0.46
	Codegen	5	0.68	0.68	0.54	0.55	0.60
	CodeT5+2B	5	0.64	0.63	0.55	0.42	0.51

Table 1: The number of clusters formed by DirectProbe and label accuracy on hidden representation of last layer on distance prediction with 5 labels.

Tokens	Model	No. of clusters	Label Accuracy	
			Not Siblings	Siblings
{Keyword-All}	GraphCodeBERT	4	0.76	0.87
	CodeT5	7	0.82	0.91
	CodeT5+220M	3	0.78	0.94
	CodeT5+220Mbi	6	0.72	0.78
	CodeT5+770M	6	0.81	0.88
	CodeRL	6	0.79	0.85
	Codegen	4	0.76	0.85
	CodeT5+2B	5	0.48	0.85
{Keyword-Identifier}	GraphCodeBERT	3	0.75	0.86
	CodeT5	4	0.80	0.86
	CodeT5+220M	3	0.80	0.87
	CodeT5+220Mbi	4	0.58	0.74
	CodeT5+770M	4	0.75	0.87
	CodeRL	4	0.67	0.78
	Codegen	3	0.77	0.83
	CodeT5+2B	3	0.65	0.76

Table 2: The number of clusters formed by DirectProbe and label accuracy on hidden representation of last layer on siblings prediction with 2 labels.

Tokens	Model	No. of clusters	Label Accuracy		
			No Edge	Comes From	Computed From
{Identifier-Identifier}	GraphCodeBERT	7	0.71	0.94	0.93
	CodeT5	4	0.57	0.86	0.90
	CodeT5+220M	4	0.69	0.90	0.88
	CodeT5+220Mbi	3	0.64	0.84	0.84
	CodeT5+770M	4	0.63	0.89	0.92
	CodeRL	6	0.65	0.85	0.84
	Codegen	5	0.63	0.86	0.92
	CodeT5+2B	4	0.63	0.89	0.92

Table 3: The number of clusters formed by DirectProbe and label accuracy on hidden representation of last layer data flow edge prediction with 3 labels.

for `Keyword-All` token pairs and even poorer for `Keyword-Identifier` pairs. However for fine-tuned (CodeRL, CodeT5_musu) and zero-shot (CodeT5+220Mbi, CodeT5+2B, CodeGen) models, the accuracy is poor on data flow and siblings task with `Keyword-Identifier` token pairs and dismal on distance prediction task.

The observations imply that the hidden representations do not encode sufficient information for the distance prediction task. As described in Section 3.3.2, this in turn implies that hidden representations of code models do not encode information about different identifier types and syntax structures. Surprisingly, the fine-tuned and zero-shot models additionally also do not properly understand which syntactic and identifier tokens are siblings and which tokens have data flow relations.

5 Discussion

5.1 Limitations of cLLMs

Our analysis of attention maps reveals that they do not encode self-attention between syntactic and related identifier tokens. For example, in the best F-score case in Figure 1, we observe that the keyword `if` pays attention to the related syntactic token `is`, but not to the related identifier `ignore`. The analysis of hidden representations reveals that they do not encode sufficient information to differentiate between common syntactic structures.

We argue that these issues limit the ability of cLLMs to understand the program flow and what the code does. Program flow depends on the value of the expression associated with the conditional (`if, elif`) or loop (`for, while`). However, the syntactic tokens do not pay attention to the associated expression. Further, the hidden representations do not encode sufficient information to differentiate between the forms of expression. Thus, the model does not understand how to evaluate an expression - whether to use the value of the variable, evaluate a comparison or logical operator, or call a function. Due to the failure of models to understand the evaluation of the expression, they cannot reason about the execution path that will be taken. Given that a program can perform different operations depending on the execution path, the model cannot quite understand what the program does.

The evaluation of the expression, and thus the flow, may also depend on the input to the program. The input is usually not provided during training. However, even CodeRL, trained with feedback based on test cases, does not encode the information to understand the program flow. Further, these limitations exist irrespective of transformer architecture, size, or training objective. Thus, it could be a fundamental limitation of the transformer architecture on coding tasks.

5.2 Code Property v/s Model Performance

Models fine-tuned on a specific task perform better than pre-trained models on that task. However, the DirectProbe analysis reveals that pre-trained models encode syntactic information better than the fine-tuned models. Our findings are consistent with those of Troshin and Chirkova (2022), whose classifier-based probing revealed that fine-tuned models encode syntactic information worse than pre-trained models. Our analysis additionally reveals that even pre-trained models do not encode syntactic-identifier relations necessary for understanding program flow. Further, Sontakke et al. (2022) showed that models fine-tuned on summarization depend on shortcut cues such as function names and variables and not on code logic for correct summary.

Models with billions of parameters perform very well on code generation and in-filling tasks in a zero-shot manner. But our analysis reveals that they encode syntactic information very poorly. The repetitive nature of code corpora compared to natural language corpora (Hindle et al., 2016; Casalnuovo et al., 2019) results in memorization in cLLMs. However, multiple works have shown that larger models are more prone to memorizing training data compared to smaller models (Rabin et al., 2023a; Yang et al., 2023b; Barone et al., 2023). Memorization, coupled with data contamination, results in good benchmark performance (Magar and Schwartz, 2022) but the benchmark performance do not translate to real-world performance (Hellendoorn et al., 2019; Aye et al., 2021).

6 Conclusion

In this paper, we critically examined arbitrary assumptions made in previous works on interpretability of cLLMs and demonstrated that these assumptions can lead to misleading conclusions.

Further, with improved experimental setting, we conducted an in-depth analysis of self-attention and hidden representations of cLLMs. The analysis revealed that cLLMs struggle to encode code relations between syntactic and identifier tokens. This

restricts their ability to understand program flow and logic. We also observed that fine-tuned models and larger models with billions of parameters encode these relations poorly compared to smaller pre-trained models. It seems that fine-tuned and larger models rely on shortcut learning and memorized code instead of code understanding.

Our work contributes to designing more robust experiments to study interpretability of cLLMs. It also suggests that it is important to explore novel training techniques and/or architectures to enhance models' capability to encode code properties, instead of using larger models with memorization. In our future work, we aim to investigate more recent instruction-tuned models by extending this study to NL-PL alignment.

Limitations

Broadly, our work has following limitations.

First, the models we analyzed use sub-word tokenizers but we performed analysis on code words. For the code word level analysis, we merged the sub-words and the attention values / hidden representations of the corresponding sub-words by taking the mean of the values. While this is a standard practice in the analysis of attention maps and hidden representation, it can also introduce minor discrepancies in the results.

Second, we only study the cases where codes are input. Thus the tasks involving text-to-code are not analyzed in our work. It is also not trivial to extend our work to text-to-code setting. Code models and LLMs in general are highly sensitive to minor changes in input. Due to this sensitivity, semantically similar texts can lead to significantly different output. We aim to extend this work to text-to-code settings by creating a statistical method to analyze NL-PL alignment in future work. Despite this limitation, our work has relevance for code-to-code and code-to-text applications.

Third, our work focuses on Python code, despite some of the models being trained on other programming languages (PLs) along with Python. Our work focuses on Python, because (1) the performance of cLLMs is much better on Python compared to other PLs and (2) Python has become the primary focus of many recent works and most recently released models have checkpoints specifically fine-tuned for Python code. However, limiting the analysis to Python also prevents us from studying certain programming constructs, such as type systems and cLLMs' understanding of types.

Acknowledgements

This research work was supported by the National Research Center for Applied Cybersecurity ATHENE.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 2655–2668. Association for Computational Linguistics.

Toufique Ahmed, Dian Yu, Chengxuan Huang, Cathy Wang, Prem Devanbu, and Kenji Sagae. 2023. Towards understanding what code language models learned. *CoRR*, abs/2306.11943.

Gareth Ari Aye, Seohyun Kim, and Hongyu Li. 2021. Learning autocompletion from real-world datasets. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*, pages 131–139. IEEE.

Razan Baltaji and Parth Thakkar. 2023. Probing numeracy and logic of language models of code. In *IEEE/ACM International Workshop on Interpretability and Robustness in Neural Software Engineering, InteNSE@ICSE 2023, Melbourne, Australia, May 14, 2023*, pages 8–13. IEEE.

Antonio Valerio Miceli Barone, Fazl Barez, Shay B. Cohen, and Ioannis Konstas. 2023. The larger they are, the harder they fail: Language models do not recognize identifier swaps in python. In *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 272–292. Association for Computational Linguistics.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *CoRR*, abs/2207.14255.

Yonatan Belinkov. 2022. Probing classifiers: Promises, shortcomings, and advances. *Comput. Linguistics*, 48(1):207–219.

Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2019. Autofocus: Interpreting attention-based neural networks by code perturbation. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 38–41. IEEE.

Casey Casalnuovo, Kenji Sagae, and Prem Devanbu. 2019. Studying the difference between natural and programming language corpora. *Empir. Softw. Eng.*, 24(4):1823–1868.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Nuo Chen, Qiushi Sun, Renyu Zhu, Xiang Li, Xuesong Lu, and Ming Gao. 2022. Cat-probing: A metric-based approach to interpret how pre-trained models for programming language attend code structure. In *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 4000–4008. Association for Computational Linguistics.

Jürgen Cito, Isil Dillig, Vijayaraghavan Murali, and Satish Chandra. 2022. Counterfactual explanations for models of code. In *44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22-24, 2022*, pages 125–134. IEEE.

Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. 2019. Unified language model pre-training for natural language understanding and generation. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 13042–13054.

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*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

Anastasiia Grishina, Max Hort, and Leon Moonen. 2023. The earlybird catches the bug: On exploiting early layers of encoder models for more efficient code classification. *CoRR*, abs/2305.04940.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 7212–7225. Association for Computational Linguistics.

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.

Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring network structure, dynamics, and function using networkx. In *Proceedings of the 7th Python in Science Conference*.

Hossein Hajipour, Ning Yu, Cristian-Alexandru Staicu, and Mario Fritz. 2022. Simscood: Systematic analysis of out-of-distribution behavior of source code models. *CoRR*, abs/2210.04802.

Vincent J. Hellendoorn, Sebastian Proksch, Harald C. Gall, and Alberto Bacchelli. 2019. When code completion fails: a case study on real-world completions. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 960–970. IEEE / ACM.

John Hewitt and Percy Liang. 2019. Designing and interpreting probes with control tasks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pages 2733–2743. Association for Computational Linguistics.

John Hewitt and Christopher D. Manning. 2019. A structural probe for finding syntax in word representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4129–4138. Association for Computational Linguistics.

Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar T. Devanbu. 2016. On the naturalness of software. *Commun. ACM*, 59(5):122–131.

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

Anjan Karmakar and Romain Robbes. 2021. What do pre-trained code models know about code? In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 1332–1336. IEEE.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *NeurIPS*.

Yue Liu, Chakkrit Tantithamthavorn, Yonghui Liu, and Li Li. 2023. On the reliability and explainability of automated code generation approaches. *CoRR*, abs/2302.09587.

José Antonio Hernández López, Martin Weyssow, Jesús Sánchez Cuadrado, and Houari A. Sahraoui. 2022. Ast-probe: Recovering abstract syntax trees from hidden representations of pre-trained language models. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 11:1–11:11. ACM.

Inbal Magar and Roy Schwartz. 2022. Data contamination: From memorization to exploitation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 157–165. Association for Computational Linguistics.

Rowan Hall Maudslay, Josef Valvoda, Tiago Pimentel, Adina Williams, and Ryan Cotterell. 2020. A tale of a probe and a parser. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 7389–7395. Association for Computational Linguistics.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*.

Md. Rafiqul Islam Rabin, Vincent J. Hellendoorn, and Mohammad Amin Alipour. 2021. Understanding neural code intelligence through program simplification. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 441–452. ACM.

Md. Rafiqul Islam Rabin, Aftab Hussain, and Mohammad Amin Alipour. 2022. Syntax-guided program reduction for understanding neural code intelligence models. In *MAPS@PLDI 2022: 6th ACM SIGPLAN International Symposium on Machine Programming, San Diego, CA, USA, 13 June 2022*, pages 70–79. ACM.

Md. Rafiqul Islam Rabin, Aftab Hussain, Mohammad Amin Alipour, and Vincent J. Hellendoorn. 2023a. Memorization and generalization in neural code intelligence models. *Inf. Softw. Technol.*, 153:107066.

Md. Rafiqul Islam Rabin, Aftab Hussain, Sahil Suneja, and Mohammad Amin Alipour. 2023b. Study of distractors in neural models of code. In *IEEE/ACM International Workshop on Interpretability and Robustness in Neural Software Engineering, InteNSE@ICSE 2023, Melbourne, Australia, May 14, 2023*, pages 1–7. IEEE.

Emily Reif, Ann Yuan, Martin Wattenberg, Fernanda B. Viégas, Andy Coenen, Adam Pearce, and Been Kim. 2019. Visualizing and measuring the geometry of BERT. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8592–8600.

Alberto Sanfeliu and King-Sun Fu. 1983. A distance measure between attributed relational graphs for pattern recognition. *IEEE Trans. Syst. Man Cybern.*, 13(3):353–362.

Ankita Nandkishor Sontakke, Manasi Patwardhan, Lovekesh Vig, Raveendra Kumar Medicherla, Ravindra Naik, and Gautam Shroff. 2022. Code summarization: Do transformers really understand code? In *Deep Learning for Code Workshop*.

Sergey Troshin and Nadezhda Chirkova. 2022. Probing pretrained models of source codes. In *Proceedings of the Fifth BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP, BlackboxNLP@EMNLP 2022, Abu Dhabi, United Arab Emirates (Hybrid), December 8, 2022*, pages 371–383. Association for Computational Linguistics.

Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(86):2579–2605.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008.

Jesse Vig, Ali Madani, Lav R. Varshney, Caiming Xiong, Richard Socher, and Nazneen Fatema Rajani. 2021. Bertology meets biology: Interpreting attention in protein language models. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

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 *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE*

2022, Pittsburgh, PA, USA, May 25-27, 2022, pages 2377–2388. ACM.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *CoRR*, abs/2305.07922.

Yue Wang, Weishi Wang, Shafiq R. 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, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics.

Jennifer C. White, Tiago Pimentel, Naomi Saphra, and Ryan Cotterell. 2021. A non-linear structural probe. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 132–138. Association for Computational Linguistics.

Yichen Xu and Yanqiao Zhu. 2022. A survey on pre-trained language models for neural code intelligence. *CoRR*, abs/2212.10079.

Kang Yang, Xinjun Mao, Shangwen Wang, Yihao Qin, Tanghaoran Zhang, Yao Lu, and Kamal Al-Sabahi. 2023a. An extensive study of the structure features in transformer-based code semantic summarization. In *31st IEEE/ACM International Conference on Program Comprehension, ICPC 2023, Melbourne, Australia, May 15-16, 2023*, pages 89–100. IEEE.

Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, DongGyun Han, and David Lo. 2023b. What do code models memorize? an empirical study on large language models of code. *CoRR*, abs/2308.09932.

Kechi Zhang, Ge Li, and Zhi Jin. 2022a. What does transformer learn about source code? *CoRR*, abs/2207.08466.

Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022b. Diet code is healthy: simplifying programs for pre-trained models of code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 1073–1084. ACM.

Yichu Zhou and Vivek Srikumar. 2021. Directprobe: Studying representations without classifiers. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 5070–5083. Association for Computational Linguistics.

A Hardware Details

We first perform a forward pass through the models on an Nvidia A6000 48GB GPU and store the attention and hidden representation for experiments. All experiments are then run on an AMD Ryzen Threadripper 5975WX with 32 cores.

B Background

Attention Analysis. In NLP, attention analysis investigates whether self-attention corresponds to linguistic relations among input tokens. For cLLMs, attention analysis quantifies how well self-attention encodes relations among code tokens, such as relations in an AST.

Probing on Hidden Representation is a technique to study the properties encoded in the hidden representations (Belinkov, 2022). Due to the many limitations of classifier or structural probe based probing techniques (Hewitt and Liang, 2019; White et al., 2021; Maudslay et al., 2020), we use DirectProbe (Zhou and Srikumar, 2021), a non-classifier-based probing technique. DirectProbe clusters the hidden representations of a specific layer based on labels for the property we want to study. Then, the convex hull of these clusters (Figure 6d) can be used to study how well hidden representations encode information about that property. The basic idea is that a good-quality representation will have well-separated clusters, while linear encoding of a property will result in each label having one cluster. The quality of clustering can be evaluated by predicting clusters for a hold-out test set.

Abstract Syntax Trees (ASTs) are data structures that represent the syntactic structure of a code. The leaf nodes of the tree represent code tokens, and internal nodes represent different constructs of the code such as `if-else` block, `identifiers`, or `parameters`. A partial AST[2] for a Python code snippet is shown in Figure 6b.

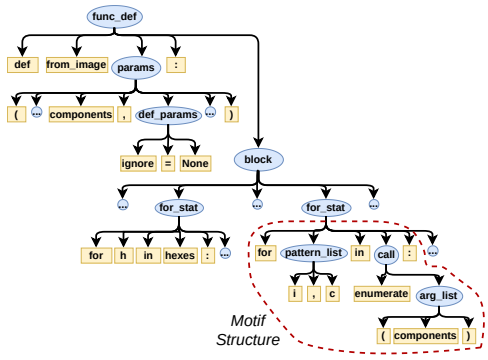
Data Flow Graphs (DFGs) have nodes representing variables and edges depicting how the values flow from one variable to another. We adopt the approach by Guo et al. (2021) to obtain the data flow relations, with two types of data flow relations, viz. `ComesFrom` and `ComputedFrom`.

Motif Structure Wan et al. (2022) defines motif structure as a non-leaf node in the AST with all it's children and assume there is a syntactical relation

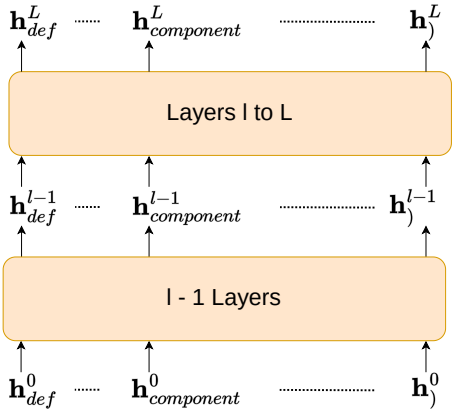
[2]We use tree-sitter (https://tree-sitter.github.io/tree-sitter/)to obtain AST of a code.

```
def from_image(cls, filename, components,
                    ignore=None,
                    col_offset=0.1,
                    row_offset=2):
    if ignore is None:
        ignore = []

    rgb = utils.loglike_from_image(filename,
                    offset=col_offset)
    loglike = np.array([utils.rgb_to_hex(t) for t in rgb])

    _, hexes = utils.tops_from_loglike(loglike,
                     offset=row_offset)

    hexes_reduced = []
    for h in hexes:
        if h not in hexes_reduced:
            if h not in ignore:
                hexes_reduced.append(h)

    list_of_Decors = []
    for i, c in enumerate(components):
        d = Decor({'colour': hexes_reduced[i], 'component':
                   c})
        list_of_Decors.append(d)

    return cls(list_of_Decors)
```

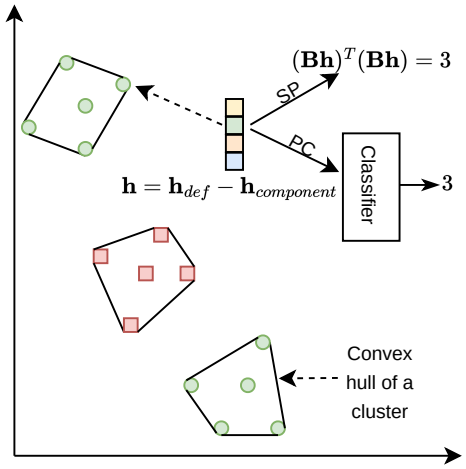
(a)



(b)



(c)



(d)

Figure 6: A python code snippet **(a)** and it's (partial) AST **(b) showing an example of motif structure**; Illustration of hidden representation in a transformer model **(c)**; An illustration of structural probe (SP), probing classifier (PC) and convex hull created by DirectProbe **(d)**.

between all leaf nodes (i.e. code tokens) of a motif structure. We show motif structure in Figure 6b.

Transformer and Self-attention. A Transformer model consists of L stacked transformer blocks. The core mechanism of a transformer block is self-attention. Given a code $c = \{c_1, c_2, ..., c_n\}$ of length n, the self-attention mechanism assigns an input token c_i attention values over all input tokens. The code c is first transformed into a list of d-dimensional vectors $\boldsymbol{H}^0 = [\boldsymbol{h}_1^0, \boldsymbol{h}_2^0, ..., \boldsymbol{h}_n^0]$. The transformer model transforms $\boldsymbol{H}^0$ into a new list of vectors $\boldsymbol{H}^L$. A layer l takes the output of the previous layer $\boldsymbol{H}^{l-1}$ as input and computes $\boldsymbol{H}^l = [\boldsymbol{h}_1^l, \boldsymbol{h}_2^l, ..., \boldsymbol{h}_n^l]$. $\boldsymbol{h}_i^l$ is the hidden representation of i^{th} word at layer l, as shown in Figure 6c. Attention values for layer l are computed as

$$Attention(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = softmax(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d}})\boldsymbol{V} \quad (1)$$

where $\boldsymbol{Q} = \boldsymbol{H}^{l-1}\boldsymbol{W}_Q^l$, $\boldsymbol{K} = \boldsymbol{H}^{l-1}\boldsymbol{W}_K^l$ and $\boldsymbol{V} = \boldsymbol{H}^{l-1}\boldsymbol{W}_V^l$. In practice, a layer l contains multiple heads, each with its own $\boldsymbol{W}_Q^l$, $\boldsymbol{W}_K^l$, $\boldsymbol{W}_V^l$ matrices. Each head thus has a set of attention values among each pair of input tokens, which constitute the attention map for that head (Figure 1).

C Model Details

We ran our experiments on multiple openly-available models chosen to represent different model architectures, sizes, training objectives and trained on different dataset. The models have parameters ranging from 110M to 3.7B parameters. We perform the experiments with pre-trained and fine-tuned models as well as models which show good benchmark performance in zero-shot setting. Among the pre-trained models, we consider CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), UniXcoder (Guo et al., 2022), CodeT5 (Wang et al., 2021) and PLBART (Ahmad et al., 2021), CodeT5+220M (Wang et al., 2023) and CodeGen (Nijkamp et al., 2023).

CodeBERT is an encoder-only bi-directional transformer with 220M parameters comprising of 12 layers, each layer having 12 heads. It has been trained on CodeSearchNet (CSN) (Husain et al., 2019) dataset with two pre-trained objectives. Masked Language Modeling (MLM) objective is used with bimodal (NL-PL pair) data, the model is trained with and Replaced Token Detection (RTD) with unimodal (only PL) data.

GraphCodeBERT uses the same architecture as CodeBERT but also takes nodes of the data flow graph (DFG) of the code as inputs with special position embeddings to indicate which tokens are nodes of DFG. It is also trained on CSN dataset. The model is first trained with MLM objective, followed by edge prediction in data flow graph and node alignment between code tokens and DFG nodes.

UniXcoder is an encoder-decoder model with 220M parameters. However, the model can be used in encoder-only, decoder-only or encoder-decoder mode using a special input token, `[MODE]`. It is also trained on CSN dataset and taked flattened ASTs of code as part of it's input during training. The model is trained with masked spans prediction, masked language modeling, multi-modal contrastive learning, whereby positive pairs are created using dropout, and cross-modal generation.

CodeT5 is an encoder-decoder model with 220M parameters trained on CSN dataset with identifier-aware and bimodal-dual generation objective. Identifier-aware pretraining uses masked span prediction, identifier tagging and masked identifier prediction alternatively to make the model attend to identifiers while bimodal-dual generation consists of NL to PL generation and PL to NL generation. Along with pre-trained CodeT5, we also experiment with CodeT5 fine-tuned for summarization task. Further, we include a larger CodeT5 model trained with next token prediction task and then trained on Python code and CodeRL (Le et al., 2022) which is fined-tuned for code generation in an actor-critic setup with feddback from test cases .

PLBART PLBART is an encoder-decoder model with 110M parameters comprising of 6 encoder layers, each with 12 heads. The model is trained with 3 denoising objectives - token masking, token deletion and token infilling - on NL and PL data from Google BigQuery [3].

CodeT5+ is a family of models trained with span denoising, causal LM, contrastive loss and matching loss. We experiment with the 220M, 770M and 2B variants of CodeT5+ model. The 220M and 770M have the same architecture as CodeT5, while the 2B variant follows the architecture of CodeGen-mono 350M for encoder and CodeGen-mono 2B for decoder. CodeT5+ can be used in encoder-only,

[3] https://console.cloud.google.com/marketplace/github-repos

Range	CodeBERT	GraphCodeBERT	UniXcoder	CodeT5	PLBART
0.0	59.13	70.3	67.28	51.92	74.63
0.0 - 0.05	39.25	28.58	31.88	46.23	74.27
0.05 - 0.3	1.48	1.00	0.76	1.64	0.97
above 0.3	0.14	0.12	0.08	0.22	0.13

Table 4: Percentage of attention values in differenr range.

encoder-decoder and decoder-only setup. So we also study the decoder of the 2B variant. Further, we also study the 220M-bimodal variant which can be used for code summarization and retrieval in zero-shot manner.

CodeGen CodeGen is a decoder-only model trained with fill-in-the-middle objective (Bavarian et al., 2022) to provide bi-directional context during training. We experiment with the 3.7B variant of the model with 16 layers and 16 heads in each layer. The model can be used for code generation in zero-shot setting.

D Datset Details

CodeSearchNet (Husain et al., 2019) dataset consists of 2 million comment-code pairs from 6 programming languages and is a commonly used dataset to pre-train models. The programming languages are Go, Java, JavaScript, PHP, Python and Ruby. The codes in the dataset are scrapped from GitHub and filtered to only contain codes with permissible licenses. Different codes have different licenses and the details of those licenses is available in the dataset. We experiment with the Python codes from test split of CSN (Husain et al., 2019).

We chose CSN for our experiments because most of the models we considered have been pre-trained on CSN or CSN augmented with additional data. Due to this, the effect of data distribution shift is minimized.

Before performing analysis we pre-process the dataset by removing any docString and code comments from the dataset. CodeBERT, GraphCodeBERT and UniXcoder has a maximum input token length of 512 tokens. So, we create a subset consisting of codes with less than 500 tokens post tokenization. CSN consists a list of code tokens for each token. For merging attention and hidden representation of sub-tokens, we use this list to keep track of where a token has been split by tokenizer. However, the list splits `*args` into `*` and `args` and `**kwargs` into `*`, `*` and `kwargs`. In Python, `*` is used for iterator unpacking and `**` for dictionary unpacking. So, to differentiate the two, we merger the `*`s of `kwargs`. From he pre-processed dataset, we randomly sample 3000 python code and run our experiments on these codes.

E Attention Distribution

In Table 4, we present the percentage of attention values which are 0, between 0 - 0.05, between 0.05 - 0.3 and more than 0.3. Note that we assume any value below 0.001 to be 0.

F Additional Attention Analysis Results

We present some additional results for attention analysis such as precision of model graphs (Figure 7) with syntax graphs and data flow graphs and graph edit distance (Figure 9)for some more models.

G t-SNE

We select 100 codes with at least 100 code tokens and get the hidden representation for each token. We then select hidden representation of the token types shown in Figure 10. We ran t-SNE on the selected hidden representation with different perplexity value (van der Maaten and Hinton, 2008) from 5 to 50 for all layers of all models. Increasing the perplexity value only made the clusters tighter but the overall distribution of points remained similar. So, the conclusion is not affected by perplexity value. We set the number of iterations to 50K, ensuring t-SNE always converges (no change in error for at least 300 iterations). We found that for all layers, tokens of same type were closer, though the clustering of same token types became tighter for deeper layers. We show the visualization for fifth layer of CodeBERT with perplexity of 50 in Figure 10.

We create a distance matrix for both the tree distance in AST and distance between hidden representation of tokens for a few code. We run t-SNE till convergence with perplexity values 5 and 10 and found the distribution to be similar. We again

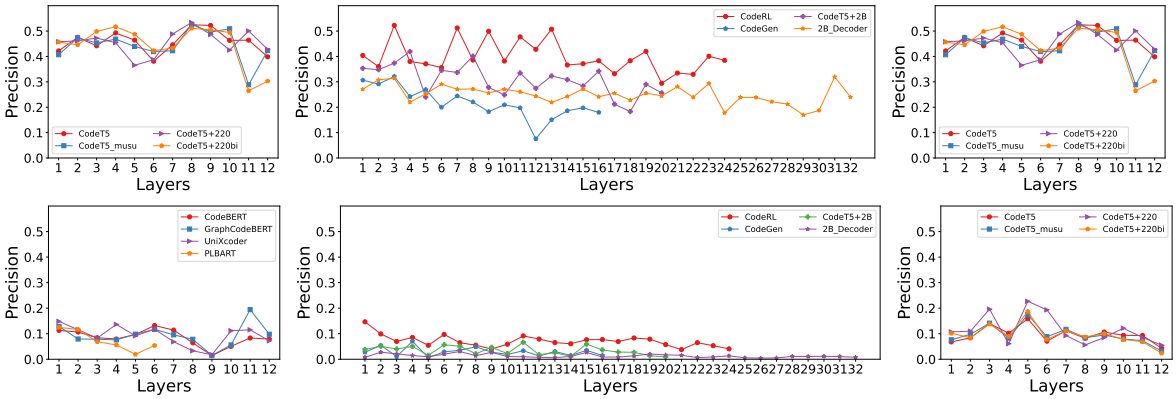


Figure 7: Precision of model graphs with syntax graphs (top) and data flow graphs (bottom).

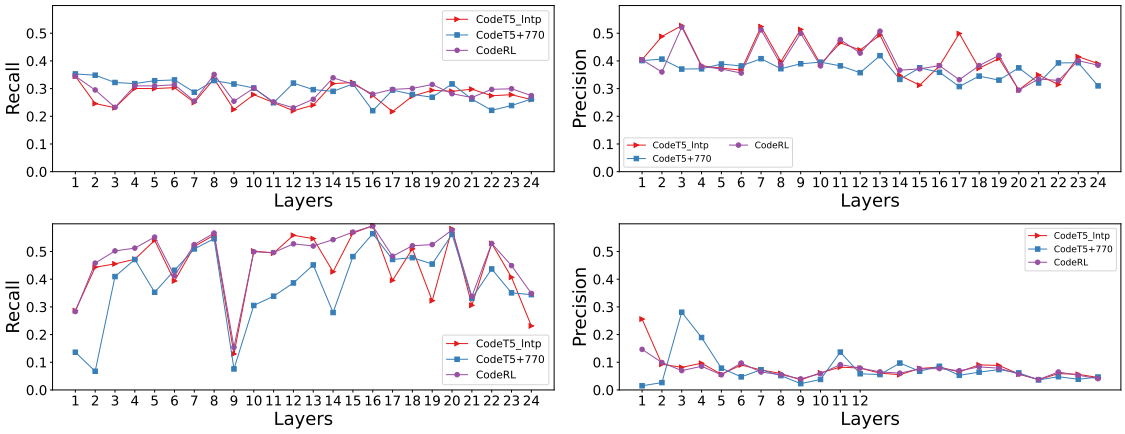


Figure 8: Precision and Recall of model graphs with syntax graphs (top) and data flow graphs (bottom).

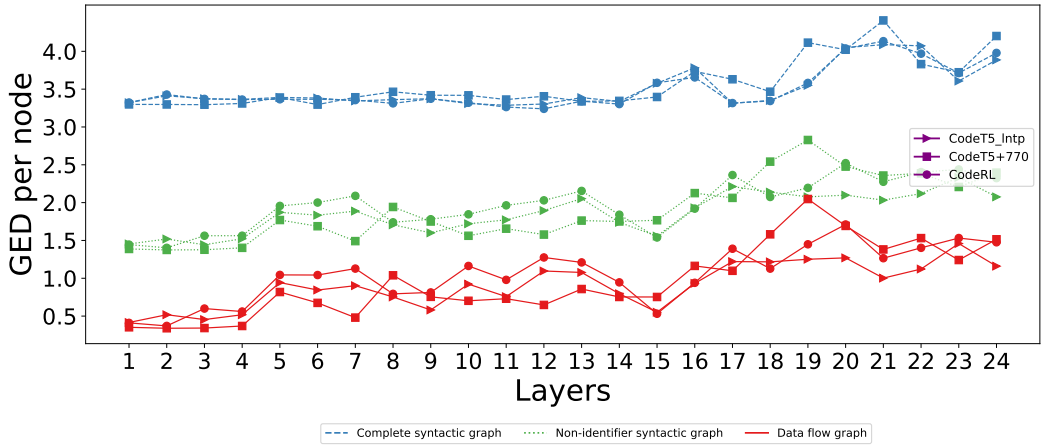


Figure 9: Graph edit distance (GED) per node (lower value show higher similarity) of model graph from DFG, non-identifier syntax graph and complete syntax graph for various models.

observed clusters of tokens of same types for hidden representation, unlike clusters of AST distance matrix. The clusters are closer for earlier layers and farther for deeper layers. We show the visualization for fifth layer of CodeBERT for code in Figure 6a in Figure 11.

We use the t-SNE implementation provided by the sci-kit learn library[4].

H DirectProbe Experiment Details

For siblings and tree distance prediction tasks, the first token is of one of the following token types: `def for if none else false true or and return not elif with try raise except break while assert print continue class`.

For distance prediction task, we randomly sample 160 codes. We select the code pairs at a maximum distance of 6, ensuring first token is of one of the selected tokens types. The second token can be of any type. We then select 1300 code pairs for each layer resulting in a dataset of 6500 data points. We split it into train and test set in the ration of 80:20. We follow the same steps for `Keyword-Identifier` too, with the difference that we use 450 codes and the second token is of type `identifier`.

For distance prediction task, we randomly sample 100 codes. We first select all tokens which are one of the selected token types. We then select equal number of siblings and non-siblings for each of these selected tokens. From this, we randomly sample 1500 siblings and 1500 non-siblings resulting in 3000 data points. We split it into train and test set in the ration of 80:20. We follow the same steps for `Keyword-Identifier` too, with the difference that we use 300 codes and the second token is of type `identifier`.

For data flow edge prediction task, we randomly sample 130 codes. We first select an identifier and then the tokens which has a data flow edge with the first token. We then select n tokens which do not have data flow edge with the first token, where,

$$n = \frac{\max(num(ComesFrom), num(ComputedFrom))}{2}. \quad (2)$$

From the selected pairs, we randomly sample 1500 pairs for each label resulting in 4500 data points. We split it into train and test set in the ration of 80:20.

In all tasks, we ensure that the same data points are used for all models and layers.

[4] https://scikit-learn.org/generated/sklearn.TSNE.html

I DirectProbe Results and Cluster Statistics

In this section, we provide the statistics of size and label of cluster created by DirectProbe for last layer of some of the models and the results of experiments with DirectProbe for middle and layers of some models and last layers of models not reported in the main text. Analysis with DirectProbe is presented in Tables 5, 6 and 7. The cluster statistics are presented in Tables 8, 9 and 10.

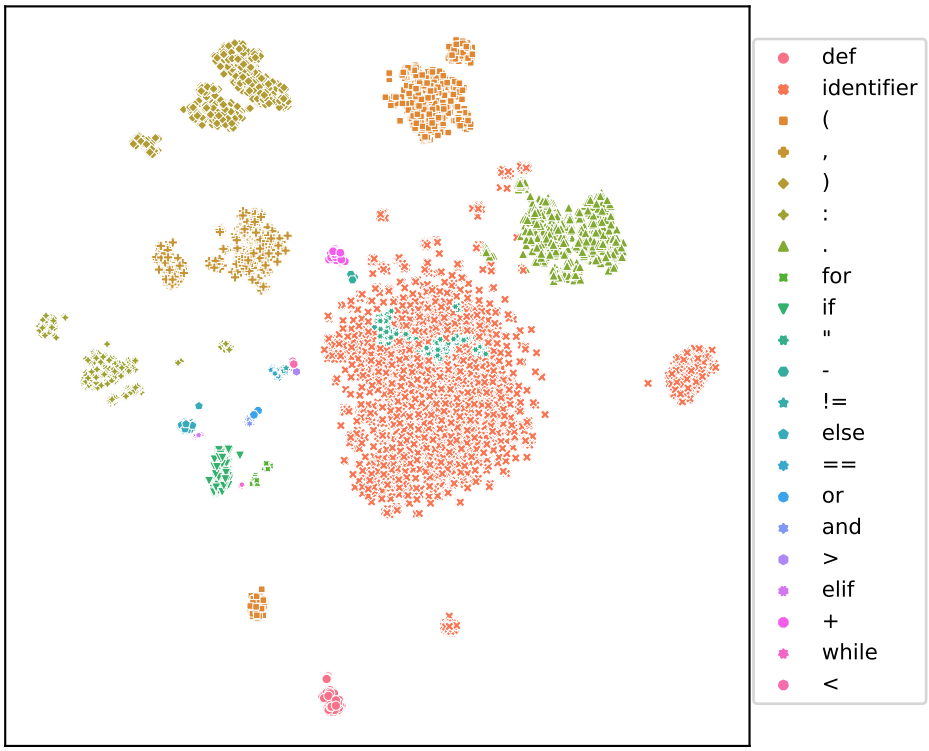


Figure 10: t-SNE visualization of hidden representation of layer 5 of CodeBERT for selected token types.

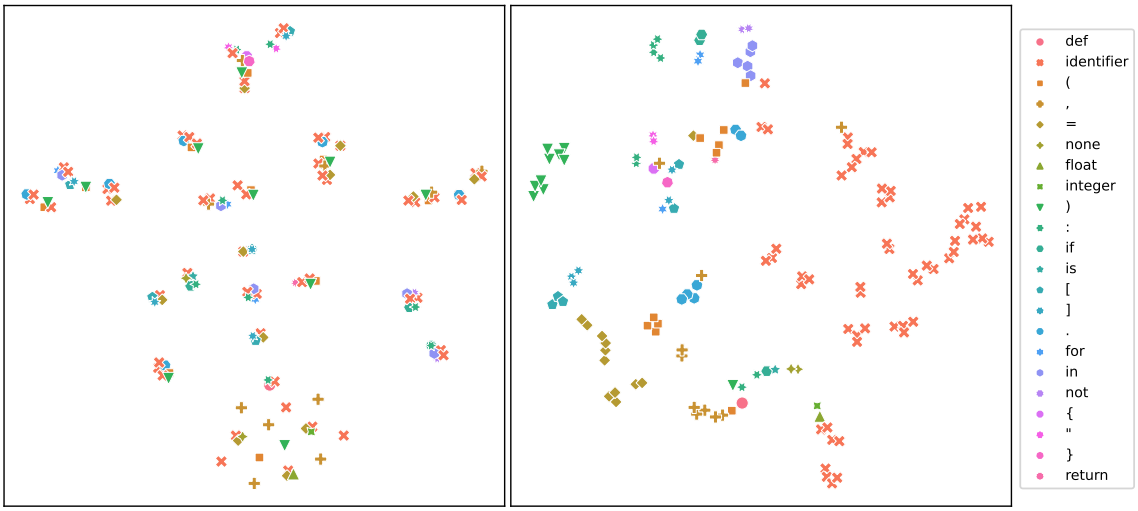


Figure 11: t-SNE visualization of distance matrix for AST(left) and hidden representation (right) of layer 5 of CodeBERT for code in Figure 6a.

Tokens	Model (Layer)	No. of clusters	Distance		Label Accuracy				
			Min	Avg	2	3	4	5	6
{Keyword-All}	CodeBERT (5)	9	0.0	1.09	0.87	0.85	0.74	0.72	0.62
	CodeBERT (9)	9	0.0	1.36	0.89	0.81	0.72	0.72	0.61
	CodeBERT (12)	10	0.0	1.27	0.85	0.75	0.73	0.68	0.55
	GraphCodeBERT (5)	11	0.0	3.99	0.88	0.84	0.75	0.70	0.63
	GraphCodeBERT (9)	9	0.0	1.74	0.83	0.81	0.69	0.68	0.62
	UniXcoder (5)	10	0.0	1.87	0.86	0.82	0.72	0.71	0.66
	UniXcoder (9)	9	0.0	0.70	0.77	0.77	0.69	0.63	0.63
	UniXcoder (12)	13	0.0	2.59	0.41	0.55	0.42	0.48	0.51
	CodeT5 (5)	9	0.0	1.65	0.79	0.80	0.70	0.67	0.65
	CodeT5 (9)	13	0.0	8.50	0.85	0.83	0.64	0.70	0.67
	PLBART (3)	13	0.0	2.60	0.79	0.77	0.62	0.70	0.57
	PLBART (6)	9	0.0	1.88	0.83	0.83	0.77	0.70	0.60
	CodeT5+220M (5)	13	0.0	0.49	0.80	0.74	0.61	0.65	0.58
	CodeT5220Mbi (5)	15	0.0	1.70	0.81	0.70	0.54	0.55	0.61
	CodeT5770M (12)	11	0.0	1.06	0.76	0.76	0.68	0.62	0.59
	CodeRL (12)	13	0.0	1.59	0.78	0.72	0.61	0.64	0.55
	CodeT5_musu (5)	13	0.0	3.38	0.76	0.72	0.57	0.66	0.59
	CodeT5_musu (12)	11	0.0	1.51	0.75	0.70	0.53	0.56	0.57
	CodeT5_lntp (12)	14	0.0	3.12	0.79	0.72	0.60	0.65	0.55
	CodeT5_lntp (24)	10	0.0	0.85	0.76	0.72	0.52	0.64	0.57
	Codegen (8)	12	0.0	87.01	0.73	0.73	0.59	0.68	0.48
	CodeT5+2B (10)	10	0.0	8.26	0.73	0.74	0.63	0.65	0.56
	CodeT5+2B_dec (16)	9	0.0	5.00	0.58	0.62	0.45	0.48	0.40
	CodeT5+2B_dec (32)	12	0.0	12.90	0.5	0.56	0.45	0.44	0.40
{Keyword-Identifier}	CodeBERT (5)	5	0.0	0.06	0.86	0.74	0.64	0.68	0.59
	CodeBERT (9)	7	0.0	3.41	0.89	0.77	0.63	0.65	0.57
	CodeBERT (12)	7	0.0	0.53	0.82	0.66	0.56	0.53	0.51
	GraphCodeBERT (5)	5	0.0	0.05	0.83	0.70	0.63	0.64	0.56
	GraphCodeBERT (9)	7	0.0	2.79	0.83	0.69	0.60	0.62	0.56
	UniXcoder (5)	7	0.0	2.33	0.82	0.66	0.61	0.61	0.49
	UniXcoder (9)	7	0.0	5.07	0.69	0.61	0.53	0.55	0.44
	UniXcoder (12)	9	0.0	5.37	0.37	0.49	0.36	0.32	0.34
	CodeT5 (5)	7	0.0	2.42	0.68	0.59	0.53	0.54	0.45
	CodeT5 (9)	5	0.0	0.23	0.78	0.66	0.60	0.61	0.51
	PLBART (3)	9	0.0	7.48	0.66	0.59	0.49	0.49	0.46
	PLBART (6)	5	0.0	0.10	0.84	0.73	0.62	0.66	0.52
	CodeT5+220M (5)	7	0.0	0.17	0.74	0.66	0.62	0.57	0.47
	CodeT5+220Mbi (5)	8	0.0	1.67	0.64	0.58	0.51	0.44	0.44
	CodeT5+770M (12)	5	0.0	0.05	0.76	0.69	0.63	0.59	0.51
	CodeRL (12)	5	0.0	0.13	0.68	0.62	0.55	0.56	0.44
	CodeT5_musu (5)	7	0.0	2.17	0.62	0.55	0.51	0.48	0.42
	CodeT5_musu (12)	7	0.0	0.50	0.62	0.61	0.52	0.48	0.42
	CodeT5_lntp (12)	5	0.0	0.13	0.66	0.60	0.55	0.55	0.43
	CodeT5_lntp (24)	5	0.0	0.13	0.69	0.64	0.59	0.55	0.46
	Codegen (8)	5	0.0	0.61	0.70	0.65	0.54	0.48	0.59
	CodeT5+2B (10)	5	0.0	0.21	0.70	0.70	0.59	0.51	0.56
	CodeT5+2B_dec (16)	5	0.0	0.33	0.55	0.57	0.48	0.49	0.48
	CodeT5+2B_dec (32)	5	0.0	0.54	0.55	0.57	0.48	0.49	0.48

Table 5: Results of analysis by DirectProbe for tree distance prediction with 5 labels.

Tokens	Model (Layer)	No. of clusters	Distance		Label Accuracy	
			Min	Avg	Not Siblings	Siblings
{Keyword-All}	CodeBERT (5)	4	0.19	8.75	0.87	0.94
	CodeBERT (9)	4	0.23	8.55	0.87	0.93
	CodeBERT (12)	4	0.18	4.63	0.87	0.88
	GraphCodeBERT (5)	5	0.24	8.38	0.87	0.91
	GraphCodeBERT (9)	4	0.24	3.30	0.84	0.92
	UniXcoder (5)	4	0.20	9.62	0.86	0.91
	UniXcoder (9)	4	0.14	6.73	0.80	0.88
	UniXcoder (12)	3	0.0	3.13	0.61	0.64
	CodeT5 (5)	5	0.17	17.09	0.84	0.85
	CodeT5 (9)	5	0.70	16.84	0.86	0.89
	PLBART (3)	4	0.19	14.17	0.83	0.86
	PLBART (6)	5	0.58	4.89	0.88	0.88
	CodeT5+220M (5)	4	0.04	1.51	0.91	0.89
	CodeT5+220Mbi (5)	5	0.24	4.56	0.89	0.82
	CodeT5+770M (12)	4	0.08	1.55	0.91	0.91
	CodeRL (12)	4	0.21	5.59	0.89	0.88
	CodeT5_musu (5)	5	0.03	5.56	0.87	0.83
	CodeT5_musu (12)	6	0.0	0.85	0.80	0.87
	CodeT5_lntp (12)	4	0.19	7.93	0.89	0.87
	CodeT5_lntp (24)	6	0.0	3.36	0.83	0.87
	Codegen (8)	3	1.76	4.62	0.79	0.89
	CodeT5+2B (10)	4	0.64	22.52	0.84	0.90
	CodeT5+2B_dec (16)	3	1.24	3.83	0.72	0.86
	CodeT5+2B_dec (32)	5	1.46	15.88	0.66	0.74
{Keyword-Identifier}	CodeBERT (5)	7	0.0	6.68	0.87	0.91
	CodeBERT (9)	4	0.31	3.67	0.88	0.91
	CodeBERT (12)	3	0.45	8.55	0.79	0.87
	GraphCodeBERT (5)	4	0.18	0.81	0.87	0.92
	GraphCodeBERT (9)	4	0.20	4.33	0.79	0.91
	UniXcoder (5)	4	0.13	6.43	0.82	0.86
	UniXcoder (9)	3	0.11	0.72	0.76	0.83
	UniXcoder (12)	4	0.14	28.73	0.47	0.56
	CodeT5 (5)	4	0.16	7.38	0.76	0.81
	CodeT5 (9)	4	0.52	19.72	0.81	0.85
	PLBART (3)	4	0.13	11.77	0.78	0.78
	PLBART (6)	4	0.28	5.17	0.80	0.87
	CodeT5+220M (5)	3	0.01	1.63	0.82	0.82
	CodeT5+220Mbi (5)	6	0.0	5.02	0.61	0.76
	CodeT5+770M (12)	3	0.05	2.60	0.83	0.88
	CodeRL (12)	3	0.13	5.55	0.75	0.80
	CodeT5_musu (5)	3	0.0	8.00	0.69	0.72
	CodeT5_musu (12)	3	0.08	2.94	0.66	0.75
	CodeT5_lntp (12)	3	0.13	5.06	0.74	0.78
	CodeT5_lntp (24)	4	0.0	0.68	0.72	0.79
	Codegen (8)	2	0.0	0.0	0.77	0.85
	CodeT5+2B (10)	3	0.59	3.68	0.75	0.84
	CodeT5+2B_dec (16)	4	1.44	159.56	0.78	0.83
	CodeT5+2B_dec (32)	4	2.56	16.33	0.67	0.72

Table 6: Results of analysis by DirectProbe for siblings prediction with 2 labels.

Tokens	Model (Layer)	No. of clusters	Distance		Label Accuracy		
			Min	Avg	No Edge	ComesFrom	ComputedFrom
{Identifier-Identifier}	CodeBERT (5)	5	0.36	7.59	0.70	0.95	0.94
	CodeBERT (9)	5	0.42	7.54	0.70	0.95	0.94
	CodeBERT (12)	4	0.24	3.68	0.69	0.91	0.90
	GraphCodeBERT (5)	4	0.41	2.32	0.68	0.94	0.94
	GraphCodeBERT (9)	4	0.51	2.90	0.73	0.95	0.95
	UniXcoder (5)	4	0.41	4.89	0.66	0.93	0.91
	UniXcoder (9)	4	0.34	4.20	0.64	0.90	0.88
	UniXcoder (12)	4	0.92	12.71	0.54	0.72	0.79
	CodeT5 (5)	6	0.0	3.40	0.69	0.92	0.81
	CodeT5 (9)	4	1.57	15.00	0.63	0.90	0.91
	PLBART (3)	6	0.0	4.76	0.68	0.90	0.83
	PLBART (6)	4	0.72	8.99	0.62	0.91	0.94
	CodeT5+220M (5)	4	0.06	1.47	0.75	0.89	0.86
	CodeT5+220Mbi (5)	3	0.18	0.61	0.70	0.86	0.79
	CodeT5+770M (12)	4	0.11	1.81	0.74	0.89	0.89
	CodeRL (12)	5	0.30	7.19	0.70	0.85	0.81
	CodeT5_musu (5)	5	0.0	6.91	0.71	0.82	0.79
	CodeT5_musu (12)	4	0.15	2.29	0.57	0.81	0.81
	CodeT5_lntp (12)	4	0.27	4.98	0.71	0.85	0.81
	CodeT5_lntp (24)	4	0.33	3.65	0.70	0.87	0.88
	Codegen (8)	4	2.57	26.09	0.52	0.82	0.90
	CodeT5+2B (10)	4	1.38	21.53	0.63	0.88	0.90
	CodeT5+2B_dec (16)	4	1.27	13.04	0.45	0.78	0.93
	CodeT5+2B_dec (32)	5	0.0	7.76	0.48	0.80	0.87

Table 7: Results of analysis by DirectProbe for data flow edge prediction with 3 labels.

CodeBERT	Cluster	0	1	2	3	4	5	6	7	8	9			
	Label	3	2	3	5	2	3	6	6	4	5			
	Size	178	806	453	225	241	400	683	357	1042	815			
GraphCodeBERT	Cluster	0	1	2	3	4	5	6	7	8				
	Label	2	3	5	3	2	6	5	6	4				
	Size	48	386	94	645	999	921	946	119	1042				
UniXCoder	Cluster	0	1	2	3	4	5	6	7	8	9	10	11	12
	Label	3	4	6	4	6	3	2	2	5	4	3	5	6
	Size	334	377	225	337	83	168	662	385	646	328	529	394	732
CodeT5	Cluster	0	1	2	3	4	5	6	7	8	9			
	Label	5	2	3	2	3	6	5	4	5	6			
	Size	26	653	354	394	677	156	61	1042	953	884			
PLBART	Cluster	0	1	2	3	4	5	6	7	8				
	Label	2	2	3	3	6	5	6	4	5				
	Size	105	942	614	417	227	183	813	1042	857				
CodeT5+220M	Cluster	0	1	2	3	4	5	6	7	8	9	10		
	Label	3	2	3	3	4	5	4	5	4	6	6		
	Size	548	1045	329	156	51	34	759	1015	223	965	75		
Codegen	Cluster	0	1	2	3	4	5	6	7	8	9	10		
	Label	3	3	2	6	3	2	5	5	4	4	6		
	Size	272	131	219	204	629	840	166	865	41	997	836		

Table 8: Cluster size and label for last layer of models for tree distance prediction task

CodeBERT	Cluster	0	1	2										
	Label	Sibling	Sibling	Non-sibling										
	Size	411	779	1210										
GraphCodeBERT	Cluster	0	1	2	3									
	Label	Sibling	Non-sibling	Non-sibling	Sibling									
	Size	1	53	1157	1189									
UniXcoder	Cluster	0	1	2	3									
	Label	Non-sibling	Sibling	Non-sibling	Sibling									
	Size	2	1153	1208	37									
CodeT5	Cluster	0	1	2	3	4	5	6						
	Label	Sibling	Non-sibling	Non-sibling	Sibling	Sibling	Sibling	Non-sibling						
	Size	664	458	135	157	365	4	617						
PLBART	Cluster	0	1	2	3	4								
	Label	Sibling	Sibling	Non-sibling	Sibling	Non-sibling								
	Size	610	126	33	454	1177								
CodeT5+220M	Cluster	0	1	2										
	Label	Non-Sibling	Non-Sibling	Sibling										
	Size	608	597	1195										
Codegen	Cluster	0	1	2	3									
	Label	Sibling	Non-Sibling	Sibling	Non-sibling									
	Size	428	2	794	1176									

Table 9: Cluster size and label for last layer of models for siblings prediction task

CodeBERT	Cluster	0	1	2	3									
	Label	NoEdge	NoEdge	Comes	Computed									
	Size	1	1208	1206	1185									
GraphCodeBERT	Cluster	0	1	2	3	4	5	6						
	Label	Computed	NoEdge	Computed	NoEdge	Computed	NoEdge	Comes						
	Size	1	1	1008	549	176	659	1206						
UniXcoder	Cluster	0	1	2	3									
	Label	NoEdge	Computed	NoEdge	Comes									
	Size	1	1185	1208	1206									
CodeT5	Cluster	0	1	2	3									
	Label	NoEdge	Computed	NoEdge	Comes									
	Size	1	1185	1208	1206									
PLBART	Cluster	0	1	2	3									
	Label	NoEdge	Computed	NoEdge	Comes									
	Size	1	1185	1208	1206									
CodeT5+220M	Cluster	0	1	2	3									
	Label	NoEdge	Computed	NoEdge	Comes									
	Size	1	1191	1201	1207									
Codegen	Cluster	0	1	2	3	4								
	Label	Computed	NoEdge	Computed	NoEdge	Comes								
	Size	1145	1126	28	101	1200								

Table 10: Cluster size and label for last layer of models for data flow edge prediction task