CodeTD: Topology of Attention Detects Hallucinations in Code LLMs

Anonymous ACL submission

Abstract

As the AI-code assistant tools become widespread, automatic assessment of the correctness of the generated code becomes a significant challenge. Code LLMs are prone to hallucinations, which may lead to code that does not solve a required problem, or even to code with severe security vulnerabilities. In this paper, we introduce CodeTD – the first approach to pre-execution assessment of code correctness based on topological data analysis (TDA) of Code LLM attention maps. Our method quantifies prompt-generation mismatch from topological patterns of attention maps. We carry out experiments with common benchmarks (HumanEval, MBPP, BigCodeBench, MultiPL-E), 5 programming languages and 10 Code LLMs of size up to 34B parameters. The experimental results show that the proposed method is better than recent baselines. Moreover, CodeTD is transferable between coding benchmarks.

1 Introduction

Code LLMs have revolutionized software engineering, yet they remain prone to hallucinations of various types. For example, syntactic and runtime errors prevent proper program execution, while logical errors lead to incorrect solution of the problem. In some cases, the generated code might contain security issues or robustness issues, such as a memory leak. In this paper, we assume that *code hallucination* is a code which is not *functionally correct*, that is, doesn't solve a problem provided in a prompt. The correctness can be checked by running functional tests. However, in practice functional tests might be (a) unavailable, (b) incomplete, (c) expensive, or (d) unsafe to execute due to security concerns. In many applications creating functional tests is a challenging task: user interfaces, distributed/multi-threaded systems, microservice architecture, real-time systems, machine learning/AI systems with stochastic behavior.

For a wide adoption of Code LLMs, there is a high need for pre-execution verification of code correctness. Regarding the current state of technology, a significant amount of time is spent on debugging and automatic rewriting of generated code (Liang et al., 2024). We hypothesize that code correctness is intrinsically linked to the structural coherence of the model's attention flow. Unlike scalar metrics (e.g., entropy), topological features capture the connectivity patterns between the problem specification (prompt) and the solution (generation), offering a robust signal for hallucination detection. Attention maps of LLMs are shown to capture semantically meaningful information and serve as a proxy of the model's "thinking process". Previous studies have shown that transformer attention maps are useful for artificial text detection (Kushnareva et al., 2021), acceptability judgment (Cherniavskii et al., 2022), and speech classification (Tulchinskii et al., 2022).

Only a few methods like CodeJudge (Tong and Zhang, 2024) are specific to detection hallucinations in Code LLMs, while research community mostly focuses on general methods for preventing and detecting hallucinations (Peng et al., 2023; Zhang et al., 2024b; Feng et al., 2024; Zhang et al., 2024a; Yehuda et al., 2024).

Our contributions are the following:

- We propose CodeTD (Code Topology Divergence), a new approach to detecting hallucinations in LLM-generated code based on analyzing topology of attention maps;
- We carry out computational experiments with 10 Code LLMs of size up to 34B parameters, four benchmarks (HumanEval, MBPP, BigCodeBench, MultiPL-E), 5 programming languages, and show that the proposed method outperforms baselines;
- We empirically show that the proposed CodeTD classifier is transferable between code benchmarks and Code LLMs, suggesting

robustness against in-distribution overfitting;

- We empirically show that features in some attention heads are systematically good indicators of hallucinations across several programming languages.

2 Related Work

Code generation via LLMs is a topic of active research. The popular projects are CodeLlama (Roziere et al., 2023), StarCoder2 (Lozhkov et al., 2024), DeepSeek-Coder (Guo et al., 2024), Qwen2.5-Coder (Hui et al., 2024), to name a few. Code LLMs differ by the data used for training, tokenizers, training and fine-tuning protocols (like RLHF), variants of attention mechanism, etc.

Several works studied attention maps in transformer-based LLMs (Clark et al., 2019; Htut et al., 2019; Michel et al., 2019). Zhang et al. (2024c) proposed to merge human and machine attention for neural code summarization.

The phenomenon of code hallucinations is studied and categorized in several papers. Tian et al. (2025) introduces a categorization of code hallucinations into four main types: mapping, naming, resource, and logic hallucinations, with each category further divided into different subcategories. Tian et al. (2025) proposed the CodeHalu dataset and studied the frequencies of different types of hallucinations in popular Code LLMs. Liu et al. (2024), Jiang et al. (2024) introduced code hallucination benchmarks. Liu et al. (2024) categorized hallucinations as intent conflicting, inconsistency, repetition, knowledge conflicting, dead code. Jiang et al. (2024) found that code LLMs are less confident when hallucinating, since hallucinated tokens have a lower probability and hallucinated generation steps have a higher entropy. Tong and Zhang (2024) proposed to guide an LLM to work in the "slow thinking" regime to obtain a more accurate evaluation of generated code correctness.

In the broader context of NLP, several works have introduced methods for preventing and detecting hallucinations. Peng et al. (2023) proposed LLM-AUGMENTER, which grounds responses in external knowledge. Zhang et al. (2024b) introduced Self-Eval, prompting an LLM to validate its own responses using internal knowledge. Feng et al. (2024) developed hallucination detection through cooperative or competitive model collaboration. Zhang et al. (2024a) improved truthfulness by editing internal representations during inference. Yehuda et al. (2024) proposed InterrogateLLM, which detects inconsistency by reconstructing queries from generated answers. Bazarova et al. (2025) introduced TOHA, which quantifies structural properties of attention graphs to detect hallucinations in Retrieval-Augmented Generation (RAG). We provide a detailed comparison with (Bazarova et al., 2025) in Appendix Q.

3 Code hallucination as functional incorrectness

In this paper, we assume that *code hallucination* is a code which is not *functionally correct*, that is, doesn't solve the problem provided in a prompt. The correctness can be checked by running functional tests. However, in practice functional tests might be (a) unavailable, (b) incomplete, (c) expensive, or (d) unsafe to execute due to security concerns. The goal of our work is to develop a pre-execution verifier of code correctness. Hallucination types can be roughly classified into syntactic, runtime, logical errors. For modern Code LLMs, the major problem is logical errors (the output does not transform the input as specified in the prompt) which correspond to $\sim$78% of wrong code cases (see Appendix M). Tian et al. (2025) do not include syntax errors into code hallucinations. However, modern Code LLMs generate syntactically incorrect code in rare cases. Thus, our definition almost coincides with (Tian et al., 2025). So, static code analysis tools are of little help. Other definitions of code hallucinations exist (Jiang et al., 2024; Liu et al., 2024), but our work is focused on what matters in practice – detecting the outcome (incorrectness) regardless of the root cause.

4 Background. Transformer-based LLMs

All state-of-the art Code LLMs are based on different variants of the transformer architecture (Vaswani et al., 2017). A transformer architecture comprises L layers of multi-head self-attention blocks, each of them having H heads. Each attention head takes the matrix $X \in \mathbb{R}^{n \times d}$ as an input, and an output is $X^{out} = A(XW^V)$, where

$$A = \text{softmax}\left(\frac{(XW^Q)(XW^K)^T}{\sqrt{d}}\right),$$

and $W^Q, W^K, W^V \in \mathbb{R}^{d \times d}$ are projection matrices, and $A \in [0,1]^{n \times n}$ is an **attention map**. In the self-attention block, the attention map shows how each token in the input sequence "interacts" with

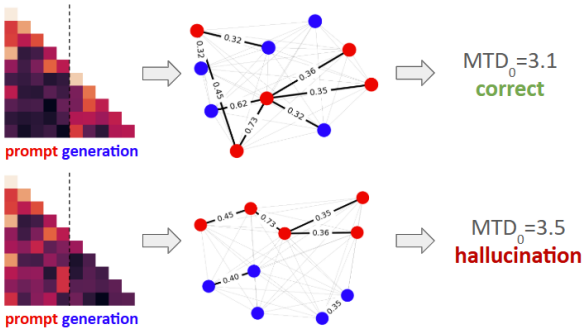


Figure 1: Two attention maps (left) have different prompt-generation connectivity strength. Attention maps are presented as weighted graphs (middle) where weights equal to attention values. Tokens of prompt and generation are depicted in red and blue respectively. Edges with weights ≥ 0.3 are shown with a bold line. Graph layout places vertices with high connecting weight closer. Correct code has high prompt-generation connectivity (top), while hallucinated code doesn't (bottom). MTD_0 is an integral characteristic of prompt-generation connectivity.

every other token in the same sequence. A token might attend more to other tokens that are contextually related. We interpret each element $a_{i,j}$ of an attention map as an "interaction force" between tokens i and j.

5 Motivation

Attention Map as a Weighted Graph. While an attention map is typically represented as a matrix, we treat it as a weighted graph. For n tokens in a sequence, we consider a fully-connected weighted graph with n vertices, where weights of edges are related to the "interaction force" between tokens (vertices).

Code Hallucinations. In the context of code generation, we naturally have two sets of tokens: a prompt and a generation. A common cause of hallucinations is when the model's attention drifts away from the prompt. We hypothesize that it means low connectivity between prompt-generation tokens. Figure 1 illustrates our intuition. Attention maps with high/low prompt-generation connectivity are shown in Figure 1 top/bottom respectively. A straightforward baseline would threshold attention scores to retain only strongly interacting tokens. However, this approach requires an unknown optimal threshold and introduces discontinuous topological changes with the change of a threshold or weights.

6 Method

Manifold Topology Divergence. To measure prompt-generation connectivity we use MTD (Manifold Topology Divergence) (Barannikov et al., 2021). MTD is a tool of Topological Data Analysis (Chazal and Michel, 2017) that can be used to evaluate the "dissimilarity" between two disjoint sets of vertices $V = P \sqcup G$ in a weighted graph $\mathcal{G} = (V, E, W)$ or, in other words, to which degree one set of vertices is connected to another set. MTD is computed via the following algorithm: (1) take vertices of $P \sqcup G$ with edges connecting vertices of P (2) add the rest of edges in ascending order by their weights (3) track changes of topological patterns (connected components, cycles[1], etc.) which appear and disappear (are "born" and "die"). The output of this algorithm is a Cross-Barcode_k – a multi-set of birth-death moments. Its integral characteristic is MTD_k – a sum of birth-death intervals' lengths (step-by-step calculation are shown in Fig. 14, 15)

$$\text{MTD}_k = \sum_{(b_i,d_i)\in CrossBarcode_k} d_i - b_i.$$

Here index k refers to a dimensionality of topological patterns: 0-connected components, 1-cycles, etc. The higher MTD_k is, the greater is the "dissimilarity" between sets of tokens. MTD_k scores of some heads have a significant discriminative power and are shown in Figure 3. MTD_k, as a kind of persistence barcode, enjoys stability w.r.t. perturbations of weights (Cohen-Steiner et al., 2005). We refer the reader to Appendix O for precise definitions.

Algorithm. Specifically, for n tokens in a prompt+generation sequence, we consider a fully-connected undirected weighted graph with n vertices. We define the edge weights as $w_{i,j} = 1 - a_{i,j}$ for $i > j$ (we use decoder-only LLMs with causal attention). Since the MTD filtration process adds edges in ascending order of weight, this transformation ensures that tokens with high attention scores (low $w_{i,j}$) are connected early in the filtration process. This allows the topological summary to prioritize strong semantic interactions between prompt and generation tokens. Then, Cross-Barcode_k and MTD_k for a weighted "attention graph" can be calculated[2]. Weak prompt-to-generation attention

[1] Appendix P explains why cycles matter.

[2] Appendix D shows examples of Cross-Barcodes and corresponding attention maps.

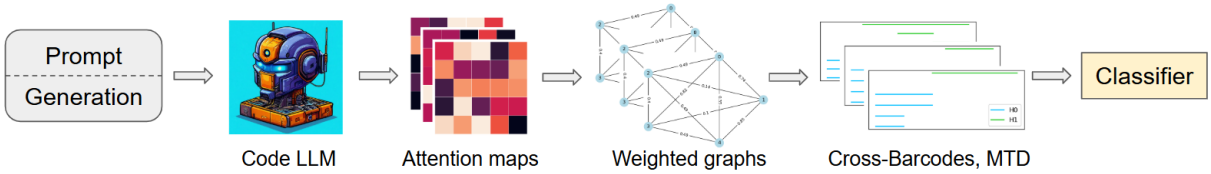
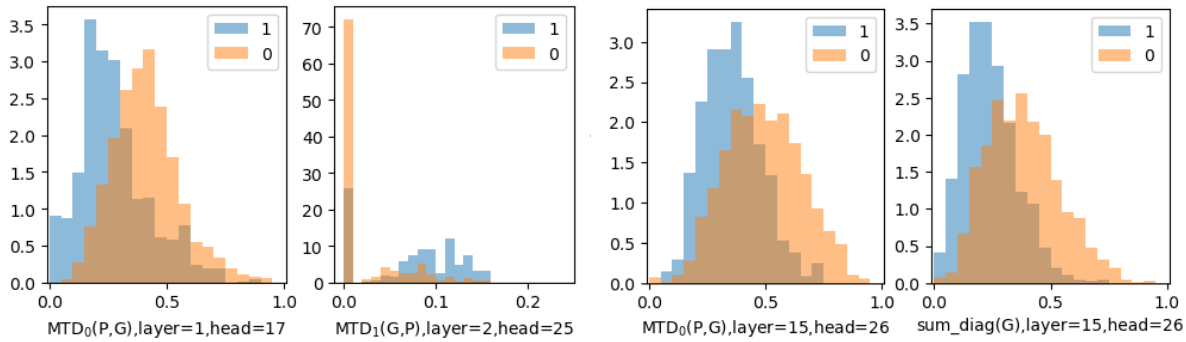


Figure 2: A pipeline of the proposed CodeTD method for hallucination detection: (1) a prompt concatenated with a generated code is fed into a Code LLM. (2) Attention maps from the Code LLM are obtained. (3) Attention maps are transformed into fully-connected weighted graphs. (4) Cross-Barcodes and MTD features for weighted graphs are calculated. (5) On the top of the generated features a binary classifier of hallucinations is fitted.



(a) HumanEval (b) MBPP

Figure 3: Distribution of classes (0-code is not correct, hallucination; 1-code is correct) vs. features from attention maps. Some of the most discriminative features are presented. Features are normalized with MinMaxScaler. Features come from CodeLlama-7B.

indicates that the generated tokens are not semantically grounded in the prompt. MTD measures the "semantic coupling" between the specification (prompt) and the implementation (generation). A low MTD score indicates the generation is structurally detached from the requirements. That is, Code LLM drifts away from the prompt during generation and hallucinates.

To predict code hallucinations, we use the following set of features which are calculated for every layer and head of a Code LLM:

- $\text{MTD}_0(P,G)/|G|$, $\text{MTD}_0(G,P)/|P|$
- $\text{MTD}_1(P,G)/|G|$, $\text{MTD}_1(G,P)/|P|$
- $\sum_{i \in P} a_{i,i}/|P|$, $\sum_{i \in G} a_{i,i}/|G|$

To ensure scale invariance across varying prompt and generation lengths, all topological features are normalized by the cardinality of the corresponding vertex sets ($|P|$ and $|G|$). In addition, averages of diagonal values of the attention matrices that are not directly present in edge weights are included. At the top of the proposed topological features, we trained an XGBoost (Chen and Guestrin, 2016) classifier [3]. Figure 2 presents the high-level pipeline of CodeTD. CodeTD is a supervised method requiring labels derived from functional tests for training; crucially, no test execution is required at inference, and the classifier is transferable across benchmarks.

7 Experiments

7.1 Dataset construction

Code LLMs and benchmarks. In our main experiments, we use the following popular code LLMs: StarCoder2-7B (Lozhkov et al., 2024), CodeLlama-7B (Roziere et al., 2023), DeepSeek-Coder-6.7B (Guo et al., 2024), Qwen2.5-Coder-7B (Hui et al., 2024), Magicoder-S-DS-6.7B (Wei et al., 2024). Additionally, we verify the proposed approach on smaller (Qwen2.5-Coder-1.5B, Qwen2.5-Coder-3B) and larger (CodeLlama-34B, DeepSeek-Coder-33B, Qwen2.5-Coder-32B) models. We adapted public benchmarks for evaluation of code generation: HumanEval (HE) (Chen et al., 2021), MBPP (Austin et al., 2021), BigCodeBench (BCB) (Zhuo et al., 2024), MultiPL-E (Cassano et al., 2023).

[3] See Appendix E for a classifier ablation study.

Generation procedure. To provide diversity and increase dataset size we do temperature sampling with $T = 0.8$ for each of the coding problems: for the small-sized (1.5B-3B) and medium-sized (6.7B-7B) models, we obtained 25 generations per task for HumanEval, 5 generations per task for MBPP, 1 generation per task for BCB, and 10 generations per task for MultiPL-E; for larger models (32B-33B), we used 10 generations per task for HumanEval. To address the quality of the proposed approach in different prompting regimes, we used a 0-shot prompt for the HumanEval dataset and a 1-shot/2-shot prompt for the MBPP dataset[4]. In addition, we provide evaluation with **greedy decoding** to account for a more practical setup, see Appendix K. CodeTD outperforms baselines in this setting. See Appendix A for further details on generation procedure.

Dataset statistics. Table 16 present a summary of code snippets generated. The correctness of the code is evaluated via functional tests provided together with the coding benchmarks. Functional tests check that the function called with certain arguments has the corresponding output (examples are shown in Figures 5, 6 in Appendix A). Incorrect code is considered a "hallucination"; prediction of code's correctness is a binary classification problem. Before moving further, note that there is a strong negative dependency between prompt and generation lengths and code quality, see Fig. 7, 8. The longer the prompt (i.e. task description) and generation (i.e. task solution) are, the lower is the probability of code's correctness. These attributes are natural baselines for hallucination's prediction.

7.2 Code Hallucination Detection

Train/test pipeline. Using the generated data, we estimate the classification quality of the proposed approach. We applied 5-fold stratified group cross-validation where different solutions of the same coding problem belonged to the same group. In this way, training and testing were performed always at non-overlapping coding problems (prompts). The reported results are the mean and std. deviation estimated over the 5 folds.

Baselines. We used the following baselines for comparison:

supervised:

- XGBoost classifier trained on tokenized prompt length, tokenized generation length;
- a linear classifier on top of a frozen CodeT5-base encoder (Wang et al., 2021);
- attention-based hallucination detectors: AttnLogDet, AttnEigvals, LapEigvals (Binkowski et al., 2025).

unsupervised/zero-shot:

- mean log. probability of generated tokens (Chen et al., 2021);
- Pylint[5], a static code analysis tool for Python;
- Self-Eval (Zhang et al., 2024b);
- CodeJudge (Tong and Zhang, 2024).

See Appendix B for further training details.

Discussion. Table 1 presents our main results. In the majority of cases, the proposed CodeTD classifier based on the features of the attention maps performed significantly better than the baselines and demonstrated stable results for all models and datasets as measured by the ROC-AUC score. Thus, we believe that the proposed topology-based attention features are strong enough to capture the most important information for code hallucination detection. Some individual features made a significant contribution to classification quality, see Figure 3. See Appendix G for details on computational complexity. CodeTD outperforms baselines on **BigCodeBench (BCB)**, the most practically relevant benchmark. The improvement over the runner-up (LapEigvals) is significant via Fisher's combined test with $p = 0.017$ (See Appendix R for full data). BCB is a stress-test for CodeTD since the proportion of correct generations can be as low as 5.9% (Table 16).

Experiments on the MultiPL-E benchmark. We evaluate CodeTD across languages with varying resource availability and ecosystem maturity: Java (high-resource), Go (medium), Rust (low), and Lua (niche) of the HumanEval subdivision of the MultiPL-E dataset (Cassano et al., 2023). We provide comparison with attention-based hallucination detectors as the strongest baselines in Table 2. The proposed CodeTD classifier detects hallucinations even in low resource and niche languages, and it outperforms other attention-based methods in approximately 84% of cases. Furthermore, there are several single features that exhibit a high predictive performance consistently across all programming languages, see Appendix J.

Evaluation with larger Code LLMs. The

[4]This follows a standard experimental setting of these benchmarks: 0-shot for HumanEval and few-shot for MBPP, e.g. (Roziere et al., 2023), (Guo et al., 2024).

[5]https://www.pylint.org/

proposed CodeTD consistently outperforms CodeJudge (Tong and Zhang, 2024) with CodeLlama-34B as evaluation model both in greedy decoding and sampling setups (see Table 12). Furthermore, the CodeTD outperforms zero-shot prompt evaluation for QwenCoder2.5-32B (Table 10), and proposed topological features of DeepSeek-Coder-6.7B improve the zero-shot prompt evaluation for DeepSeek-R1-671B (Table 11). See Appendix I for further details.

7.3 Ranking Ability

Next, we assess the usefulness of CodeTD for ranking code generations. For each problem, all generations were ranked according to the predicted probability of correctness and one with the highest probability was selected. Following pass@1 computation, a baseline was random selection of a code generation. The use of CodeTD in this setting leads to a significantly higher pass@1 score for all Code LLMs, see Table 4. This result justifies that the proposed CodeTD method can be successfully applied for best-of-N sampling.

7.4 Transferability

Cross-benchmark transferability. In this setting, hallucination classifiers for a fixed Code LLM were trained on data for one benchmark (HumanEval, MBPP) and evaluated on another, then repeated vice versa. Table 3 demonstrates that CodeTD is transferable, although performance is lower than when training and testing is done on the same benchmark. The proposed attention features achieve better transferability in 80% of cases, as measured by ROC-AUC for both the HE $\rightarrow$ MBPP and MBPP $\rightarrow$ HE transfer. We hypothesize the changes in the classifiers' performance compared to results of Section 7.2 are related to differences in the prompt structure: we use 0-shot prompt for HumanEval and 1-shot prompt for MBPP. The proposed CodeTD has consistent quality when the linguistic complexity of the prompt is increased, see Table 13 in Appendix.

Cross-model transferability. We analyze the hallucination detection quality when code-generating and feature-extracting models are different. For each of code-generating models, we use several Code LLMs to extract attention features for the generated candidate solutions, see Table 17. The best performance across the majority of code-generating models is achieved by Magicoder-S-DS-6.7B, which outperformed a larger DeepSeek-

Method	HE	MBPP	BCB
DeepSeek-Coder-6.7B			
Prompt. Len.	58.0 ± 6.7	49.5 ± 4.1	54.9 ± 2.3
Gen. Len.	58.5 ± 3.7	55.4 ± 1.7	54.7 ± 2.7
Mean Log. Prob.	68.9 ± 3.0	60.7 ± 2.2	50.1 ± 2.9
Pylint	54.7 ± 1.1	52.8 ± 0.9	61.2 ± 1.1
CodeT5-base ft.	63.1 ± 4.7	52.8 ± 3.2	54.5 ± 2.8
Self-Eval	57.3 ± 4.5	51.0 ± 0.8	52.9 ± 1.6
AttnLogDet	78.8 ± 3.6	77.3 ± 5.1	67.5 ± 4.8
AttnEigvals	75.1 ± 3.2	75.9 ± 3.5	$\underline{67.6 \pm 1.9}$
LapEigvals	$\underline{81.7 \pm 1.9}$	$\underline{77.6 \pm 4.5}$	64.6 ± 0.5
CodeTD (ours)	$\mathbf{86.4 \pm 1.8}$	$\mathbf{81.2 \pm 3.7}$	$\mathbf{69.2 \pm 2.8}$
StarCoder2-7B			
Prompt. Len.	56.3 ± 5.9	55.2 ± 4.0	51.3 ± 6.5
Gen. Len.	58.9 ± 2.4	59.5 ± 1.2	61.7 ± 3.1
Mean Log. Prob.	69.4 ± 1.7	61.9 ± 3.6	79.4 ± 3.3
Pylint	58.9 ± 1.5	53.0 ± 0.9	59.5 ± 1.9
CodeT5-base ft.	62.2 ± 1.8	56.4 ± 1.9	62.5 ± 5.3
Self-Eval	50.6 ± 1.7	59.0 ± 2.1	46.2 ± 3.6
AttnLogDet	80.4 ± 4.3	75.3 ± 4.2	75.3 ± 4.1
AttnEigvals	78.2 ± 2.1	73.2 ± 3.1	73.8 ± 2.3
LapEigvals	$\underline{80.6 \pm 2.9}$	$\underline{79.3 \pm 2.8}$	$\mathbf{85.3 \pm 2.4}$
CodeTD (ours)	$\mathbf{82.5 \pm 2.1}$	$\mathbf{81.4 \pm 3.6}$	$\underline{83.7 \pm 4.3}$
CodeLlama-7B			
Prompt. Len.	61.6 ± 4.4	59.1 ± 4.2	51.3 ± 4.3
Gen. Len.	60.1 ± 5.3	60.8 ± 2.5	58.7 ± 2.4
Mean Log. Prob.	64.1 ± 2.0	61.0 ± 3.7	57.1 ± 3.5
Pylint	55.1 ± 0.3	53.1 ± 0.5	60.1 ± 0.8
CodeT5-base ft.	74.5 ± 6.3	61.7 ± 3.0	50.7 ± 3.9
Self-Eval	49.7 ± 1.7	50.0 ± 0.0	51.1 ± 1.8
AttnLogDet	77.3 ± 5.2	75.6 ± 1.8	71.4 ± 2.9
AttnEigvals	80.2 ± 6.0	75.6 ± 3.1	68.8 ± 3.4
LapEigvals	$\underline{83.3 \pm 3.6}$	$\underline{79.7 \pm 2.7}$	$\underline{73.1 \pm 2.9}$
CodeTD (ours)	$\mathbf{85.6 \pm 3.9}$	$\mathbf{83.4 \pm 3.3}$	$\mathbf{75.0 \pm 3.4}$
Qwen2.5-Coder-7B			
Prompt. Len.	56.6 ± 4.1	53.1 ± 2.9	51.2 ± 1.8
Gen. Len.	57.3 ± 1.1	57.2 ± 2.5	55.7 ± 1.8
Mean Log. Prob.	61.4 ± 3.6	61.6 ± 0.7	51.5 ± 3.0
Pylint	63.7 ± 2.4	62.2 ± 2.3	63.8 ± 1.0
CodeT5-base ft.	62.5 ± 4.8	54.1 ± 3.1	54.2 ± 6.1
Self-Eval	73.3 ± 5.0	66.7 ± 1.7	52.5 ± 3.0
AttnLogDet	72.5 ± 3.6	70.6 ± 3.8	$\mathbf{70.3 \pm 3.3}$
AttnEigvals	74.2 ± 3.5	72.2 ± 4.3	67.8 ± 3.5
LapEigvals	$\underline{78.4 \pm 2.6}$	$\underline{76.8 \pm 4.3}$	66.4 ± 0.9
CodeTD (ours)	$\mathbf{81.6 \pm 4.7}$	$\mathbf{80.9 \pm 3.5}$	$\underline{70.0 \pm 1.4}$
Magicoder-S-DS-6.7B			
Prompt. Len.	55.7 ± 6.7	52.2 ± 2.0	56.5 ± 4.9
Gen. Len.	52.2 ± 3.6	53.7 ± 1.3	53.9 ± 4.2
Mean Log. Prob.	68.8 ± 4.1	60.0 ± 2.4	54.7 ± 5.0
Pylint	52.7 ± 0.9	52.0 ± 1.2	60.5 ± 2.0
CodeT5-base ft.	56.8 ± 4.0	52.3 ± 3.9	56.5 ± 5.5
Self-Eval	46.1 ± 5.1	49.2 ± 1.1	51.7 ± 2.7
AttnLogDet	75.8 ± 2.6	78.3 ± 3.1	$\underline{68.2 \pm 2.6}$
AttnEigvals	71.5 ± 3.4	75.1 ± 2.7	$\mathbf{68.4 \pm 3.2}$
LapEigvals	$\underline{76.7 \pm 3.9}$	$\underline{79.5 \pm 1.4}$	66.8 ± 1.0
CodeTD (ours)	$\mathbf{79.3 \pm 3.5}$	$\mathbf{80.7 \pm 2.5}$	$\mathbf{68.4 \pm 1.5}$

Table 1: ROC-AUC of code hallucination detection for generation with sampling temperature $T = 0.8$.

Method	Java	Go	Rust	Lua
StarCoder2-7B				
AttnLogDet	80.7	82.4	77.8	74.6
AttnEigvals	81.6	80.4	75.2	77.8
LapEigvals	81.6	82.5	77.1	78.8
CodeTD (ours)	**82.5**	**86.6**	**82.5**	**82.0**
CodeLlama-7B				
AttnLogDet	**85.2**	55.5	63.4	–
AttnEigvals	78.2	66.7	76.7	–
LapEigvals	75.7	68.6	74.3	–
CodeTD (ours)	76.8	**81.9**	**77.5**	–
DeepSeek-Coder-6.7B				
AttnLogDet	80.1	81.9	78.3	74.7
AttnEigvals	78.2	79.0	74.5	84.6
LapEigvals	83.1	**89.0**	82.1	86.1
CodeTD (ours)	**84.5**	85.2	**82.3**	**86.2**
Qwen2.5-Coder-7B				
AttnLogDet	65.7	76.7	72.4	80.3
AttnEigvals	78.3	85.9	84.0	85.8
LapEigvals	80.8	87.9	79.3	87.3
CodeTD (ours)	**81.7**	**91.1**	**87.5**	**90.1**
Magicoder-S-DS-6.7B				
AttnLogDet	74.5	70.2	69.8	68.8
AttnEigvals	69.2	70.9	69.9	75.4
LapEigvals	76.2	**84.0**	**74.8**	79.6
CodeTD (ours)	**77.8**	80.7	**74.8**	**80.1**

Table 2: Average ROC-AUC of the proposed method over 5 folds on different programming languages from MultiPL-E dataset. Due to space limitations, see Table 21 in Appendix for full statistics with std. deviations.

Coder-33B model. On the contrary, the worst detection quality is consistently obtained with StarCoder2-7B model. This result demonstrates that medium-sized models can be powerful feature extractors in code hallucination detection.

7.5 Feature Importance

In its base setup, CodeTD extracts features from the attention maps of all layers and heads. However, we observed that the trained XGBoost classifier exhibited natural sparsity – only about 25% of features were meaningful, based on XGBoost's feature importance scores. To further investigate feature importance and selection, we adopted a two-stage pipeline. First, for a given sparsity level, we selected the most critical features according to the importance scores from a classifier trained on all attention features simultaneously. Second, we trained a new XGBoost classifier using only the selected features. As shown in Figure 4, this procedure retained just 5% of all attention features with no significant loss in classification quality, underscoring that only a small subset of attention heads is relevant for hallucination detection.

Method	HE → MBPP	MBPP → HE
StarCoder2-7B		
Mean Log. Prob.	63.8	**71.8**
CodeT5-base ft.	53.7	59.1
AttnLogDet	57.7	59.3
AttnEigvals	53.0	60.8
LapEigvals	65.2	66.6
CodeTD (ours)	**67.7**	66.0
CodeLlama-7B		
Mean Log. Prob.	57.7	65.0
CodeT5-base ft.	54.9	62.4
AttnLogDet	60.2	68.7
AttnEigvals	57.8	66.4
LapEigvals	60.7	64.2
CodeTD (ours)	**69.5**	**80.3**
DeepSeek-Coder-6.7B		
Mean Log. Prob.	62.7	69.1
CodeT5-base ft.	53.4	55.9
AttnLogDet	59.9	58.9
AttnEigvals	55.1	64.3
LapEigvals	63.5	66.4
CodeTD (ours)	**69.9**	**72.4**
Qwen2.5-Coder-7B		
Mean Log. Prob.	60.3	64.7
CodeT5-base ft.	49.1	51.6
AttnLogDet	55.8	56.5
AttnEigvals	49.9	57.9
LapEigvals	65.5	62.1
CodeTD (ours)	**70.9**	**65.8**
Magicoder-S-DS-6.7B		
Mean Log. Prob.	63.8	**69.8**
CodeT5-base ft.	49.3	45.9
AttnLogDet	65.4	65.0
AttnEigvals	59.5	58.6
LapEigvals	68.1	65.9
CodeTD (ours)	**73.5**	56.9

Table 3: Transferability of code hallucination detectors, ROC-AUC. Each classifier was trained on HE (MBPP) dataset and tested on MBPP (HE) dataset.

7.6 A Fine-Grained Error Type Classification

We treat specific Python exception types as fine-grained hallucination categories and train a multi-class XGBoost classifier (Table 5). The results demonstrate that CodeTD capable of identifying the majority of errors of different types, consistently across the models. Here is a breakdown of

Model	Random	Clf. Prob.
HE		
StarCoder2-7B	28.6 ± 5.5	$\mathbf{43.3 \pm 9.0}$
CodeLlama-7B	26.0 ± 5.1	$\mathbf{39.7 \pm 7.2}$
DeepSeek-Coder-6.7B	39.1 ± 4.9	$\mathbf{56.7 \pm 7.4}$
Qwen2.5-Coder-7B	51.8 ± 8.0	$\mathbf{64.0 \pm 7.3}$
Magicoder-S-DS-6.7B	72.5 ± 10.0	$\mathbf{74.3 \pm 6.1}$
MBPP		
StarCoder2-7B	43.0 ± 3.6	$\mathbf{49.6 \pm 4.6}$
CodeLlama-7B	35.2 ± 3.3	$\mathbf{43.6 \pm 3.4}$
DeepSeek-Coder-6.7B	53.0 ± 2.5	$\mathbf{61.4 \pm 2.3}$
Qwen2.5-Coder-7B	52.6 ± 3.6	$\mathbf{62.0 \pm 2.4}$
Magicoder-S-DS-6.7B	61.4 ± 3.4	$\mathbf{64.8 \pm 2.1}$

Table 4: pass@1 scores across variants of ranking of code generations.

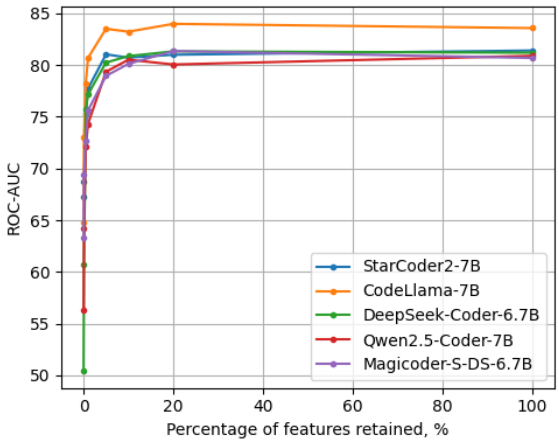


Figure 4: ROC-AUC vs. percentage of retained features, MBPP.

detection accuracy of particular types of errors for CodeLlama-7B: AssertionError: 82.0%, IndexError: 97.8%, NameError: 75.7%, RecursionError: 100%, SyntaxError: 93.3%, TypeError: 80.6%, ValueError: 87.5%, ModuleNotFoundError, ZeroDivisionError, UnboundLocalError, IndentationError, AttributeError, timed out: 80%. Some error types were grouped because of a very low frequency. Individual features of some heads can classify logic error vs. other errors, see Appendix N.

7.7 Ablation Study

We assess the contribution of each proposed CodeTD feature type through an ablation study. We progressively trained an XGBoost classifier, starting with diagonal attention values and then adding 0-dim and 1-dim MTD features, following the order of computational complexity. As shown in Appendix Tables 22, 14, adding topological features consistently improved hallucination detection

Model	Accuracy	F1-Score
StarCoder2-7B	0.7 ± 0.02	0.68 ± 0.02
CodeLlama-7B	0.66 ± 0.02	0.62 ± 0.02
DeepSeek-Coder-6.7B	0.7 ± 0.03	0.68 ± 0.04
Qwen2.5-Coder-7B	0.64 ± 0.02	0.6 ± 0.02
Magicoder-S-DS-6.7B	0.73 ± 0.02	0.71 ± 0.02

Table 5: Performance of multi-classif. of error types.

in 92% of cases and led to more robust, transferable classifiers compared to using diagonal features alone.

8 Conclusion

In this paper, we propose CodeTD, a new hallucination detection approach for Code LLMs. Our approach provides pre-execution code verification and is useful when functional tests are (a) unavailable, (b) incomplete, (c) expensive, or (d) unsafe to run due to security concerns. Our approach is based on the introspection of an LLM: we get attention maps, transform them to weighed graphs, and study prompt-generation connectivity patterns. The proposed topological features of these graphs have been empirically shown to be relevant for code hallucination detection. CodeTD outperformed recent baselines in the task of code hallucination detection. CodeTD is transferable across the coding benchmarks and Code LLMs.

Application of CodeTD. Beyond hallucination detection, the natural next step is hallucination mitigation. We envision three deployment pathways: (1) best-of-N sampling by picking a sample with the highest CodeTD score (improves pass@1 up to 17.6%, Section 7.3) (2) tracing hallucination up to the first wrong token and regenerating from this step. This is a natural scenario for an agentic loop. We leave this topic for further research (3) abstention/fallback mechanisms for human-in-the-loop workflows.

We believe that our work facilitates the safer deployment of Code LLMs in security-critical applications by providing a pre-execution verification layer. In a wider context, our work contributes to the study of interpretation and generalization in NLP models, since hallucinations and generalization ability are intrinsically tied.

Limitations

Although we have achieved good experimental results, we realize that our research has several limitations. Our method targets hallucinations that

manifest in the geometry of the model's attention; it does not unravel all root causes (e.g., spurious pre-training correlations, decoding drift, RLHF bias). Extending the analysis to these factors is left for future work. Finally, our approach can predict whether a code is correct as a whole but can not point to a specific place with a bug.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Serguei Barannikov, Ilya Trofimov, Grigorii Sotnikov, Ekaterina Trimbach, Alexander Korotin, Alexander Filippov, and Evgeny Burnaev. 2021. Manifold topology divergence: a framework for comparing data manifolds. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 7294–7305.

Alexandra Bazarova, Aleksandr Yugay, Andrey Shulga, Alina Ermilova, Andrei Volodichev, Konstantin Polev, Julia Belikova, Rauf Parchiev, Dmitry Simakov, Maxim Savchenko, and 1 others. 2025. Hallucination detection in llms with topological divergence on attention graphs. *arXiv preprint arXiv:2504.10063*.

Jakub Binkowski, Denis Janiak, Albert Sawczyn, Bogdan Gabrys, and Tomasz Jan Kajdanowicz. 2025. Hallucination detection in llms using spectral features of attention maps. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 24365–24396.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691.

Frédéric Chazal and Bertrand Michel. 2017. An introduction to topological data analysis: fundamental and practical aspects for data scientists. *CoRR*, abs/1710.04019.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794.

Daniil Cherniavskii, Eduard Tulchinskii, Vladislav Mikhailov, Irina Proskurina, Laida Kushnareva, Ekaterina Artemova, Serguei Barannikov, Irina Piontkovskaya, Dmitri Piontkovski, and Evgeny Burnaev. 2022. Acceptability judgements via examining the topology of attention maps. *arXiv preprint arXiv:2205.09630*.

Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D Manning. 2019. What does BERT look at? an analysis of BERT's attention. *Proceedings of the 2019 ACL Workshop BlackboxNLP*.

David Cohen-Steiner, Herbert Edelsbrunner, and John Harer. 2005. Stability of persistence diagrams. In *Proceedings of the twenty-first annual symposium on Computational geometry*, pages 263–271.

Shangbin Feng, Weijia Shi, Yike Wang, Wenxuan Ding, Vidhisha Balachandran, and Yulia Tsvetkov. 2024. Don't hallucinate, abstain: Identifying llm knowledge gaps via multi-llm collaboration. *arXiv preprint arXiv:2402.00367*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Phu Mon Htut, Jason Phang, Shikha Bordia, and Samuel R Bowman. 2019. Do attention heads in bert track syntactic dependencies? *arXiv preprint arXiv:1911.12246*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Nan Jiang, Qi Li, Lin Tan, and Tianyi Zhang. 2024. Collu-bench: A benchmark for predicting language model hallucinations in code. *arXiv preprint arXiv:2410.09997*.

Laida Kushnareva, Daniil Cherniavskii, Vladislav Mikhailov, Ekaterina Artemova, Serguei Barannikov, Alexander Bernstein, Irina Piontkovskaya, Dmitri Piontkovski, and Evgeny Burnaev. 2021. Artificial text detection via examining the topology of attention maps. *arXiv preprint arXiv:2109.04825*.

Jenny T Liang, Chenyang Yang, and Brad A Myers. 2024. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13.

Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. 2024. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Paul Michel, Omer Levy, and Graham Neubig. 2019. Are sixteen heads really better than one? *Advances in neural information processing systems*, 32.

Baolin Peng, Michel Galley, Pengcheng He, Hao Cheng, Yujia Xie, Yu Hu, Qiuyuan Huang, Lars Liden, Zhou Yu, Weizhu Chen, and 1 others. 2023. Check your facts and try again: Improving large language models with external knowledge and automated feedback. *arXiv preprint arXiv:2302.12813*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Yuchen Tian, Weixiang Yan, Qian Yang, Xuandong Zhao, Qian Chen, Wen Wang, Ziyang Luo, Lei Ma, and Dawn Song. 2025. Codehalu: Investigating code hallucinations in llms via execution-based verification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 25300–25308.

Weixi Tong and Tianyi Zhang. 2024. Codejudge: Evaluating code generation with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP*.

Eduard Tulchinskii, Kristian Kuznetsov, Laida Kushnareva, Daniil Cherniavskii, Serguei Barannikov, Irina Piontkovskaya, Sergey Nikolenko, and Evgeny Burnaev. 2022. Topological data analysis for speech processing. *arXiv preprint arXiv:2211.17223*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *Preprint*, arXiv:2109.00859.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering code generation with OSS-instruct. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 52632–52657. PMLR.

Yakir Yehuda, Itzik Malkiel, Oren Barkan, Jonathan Weill, Royi Ronen, and Noam Koenigstein. 2024. Interrogatellm: Zero-resource hallucination detection in llm-generated answers. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL*, pages 9333–9347.

Shaolei Zhang, Tian Yu, and Yang Feng. 2024a. TruthX: Alleviating hallucinations by editing large language models in truthful space. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8908–8949, Bangkok, Thailand. Association for Computational Linguistics.

Xiaoying Zhang, Baolin Peng, Ye Tian, Jingyan Zhou, Lifeng Jin, Linfeng Song, Haitao Mi, and Helen Meng. 2024b. Self-alignment for factuality: Mitigating hallucinations in llms via self-evaluation. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL*.

Yifan Zhang, Jiliang Li, Zachary Karas, Aakash Bansal, Toby Jia-Jun Li, Collin McMillan, Kevin Leach, and Yu Huang. 2024c. Eyetrans: Merging human and machine attention for neural code summarization. *Proceedings of the ACM on Software Engineering*, 1(FSE):115–136.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Details on Generation Procedure

We generate solutions for the coding problems with a temperature of 0.8. For the HumanEval and MultiPL-E datasets, the maximum length of the model output (i.e., input prompt + generation) was limited to 512 tokens. For the MBPP dataset, the maximum number of new tokens to generate was set to 256. Figures 5, 6 provide examples of prompts and generations for HumanEval and MBPP datasets. We followed the guidelines[6] to post process the model output and extract the valid problem solution. For the BCB dataset, we strictly follow the original generation procedure available at the official repository[7]. To compute the attention features according to the method proposed in Section 6, we used the attention submatrix corresponding to the input prompt and the valid solution to the problem. Experiments with each dataset-Code

[6]https://github.com/bigcode-project/bigcode-evaluation-harness

[7]https://github.com/bigcode-project/bigcodebench

```
from typing import List


def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
```
problem description

```
    for i in range(len(numbers) - 1):
        for j in range(i+1, len(numbers)):
            if abs(numbers[i] - numbers[j]) <= threshold:
                return True
    return False
```
generation

Figure 5: Example of prompt (problem description) and model generation for the HumanEval dataset.

```
You are an expert Python programmer, and here is your task: Write a function to find the
similar elements from the given two tuple lists. Your code should pass these tests:

assert similar_elements((3, 4, 5, 6),(5, 7, 4, 10)) == (4, 5)
assert similar_elements((1, 2, 3, 4),(5, 4, 3, 7)) == (3, 4)
assert similar_elements((11, 12, 14, 13),(17, 15, 14, 13)) == (13, 14)
[BEGIN]
def similar_elements(test_tup1, test_tup2):
  res = tuple(set(test_tup1) & set(test_tup2))
  return (res)
[DONE]
```
one-shot

```
You are an expert Python programmer, and here is your task: Write a python function to
remove first and last occurrence of a given character from the string. Your code should
pass these tests:

assert remove_Occ("hello","l") == "heo"
assert remove_Occ("abcda","a") == "bcd"
assert remove_Occ("PHP","P") == "H"
[BEGIN]
```
problem description

```
def remove_Occ(s,c):
  return s.replace(c,'',s.count(c)-1)
[DONE]
```
generation

Figure 6: Example of prompt (one-shot example and problem description) and model generation for the MBPP dataset.

LLM took $\approx$ 8 GPU hours. For computational experiments, we used NVIDIA TITAN RTX.

B Details on Training Procedure

In CodeTD, we utilized the XGBClassifier with an approximation tree method "hist" from the XGBoost library[8]. For the code hallucination detector based on CodeT5-base embeddings, we used the pre-trained frozen CodeT5-base encoder with a trainable classification head consisting of 2 linear layers with hidden dimensionality 768. The classification head was trained for 100 epochs with batch size 32 and learning rate $3e-5$.

Self-Eval (Zhang et al., 2024b) is a way to evaluate the responses of an LLM using its internal knowledge. Self-Eval extracts a list of atomic claims from the responses and then prompts an LLM itself to validate the factuality of the claims. Self-Eval is not directly applicable to Code LLMs, since there are no "facts" in the code. However, we applied the core idea of Self-Eval by prompting Code LLMs to evaluate the functional correctness of a generated code.

In addition, we have adapted Interrogate-LLM (Yehuda et al., 2024) to detect hallucinations in LLM-generated code. As an embedding model, we used the CodeT5+ 110M Embedding model, $K = 5$ and a fixed temperature.

For comparison with attention-based hallucination detectors (AttnLogDet, AttnEigvals, LapEigvals), we followed the procedure in Binkowski et al. (2025) and utilized the code[9] to compute attention features. For fair comparison, we keep the number of top eigenvalues to be equal to the number of proposed CodeTD features and use XGBoost as a classifier.

C Evaluation Metrics

This section briefly introduces the evaluation metrics utilized throughout the work.

As the main evaluation metric, we use ROC-AUC to account for possible class imbalance. The *ROC curve* demonstrates the quality of a binary classifier for all possible classification thresholds. The X-axis corresponds to the *False Positive Rate (FPR)* and the Y-axis corresponds to the *True Positive Rate (TPR)* which can be defined as follows: $FPR = \frac{FP}{FP+TN}$, $TPR = \frac{TP}{FP+FN}$, where *TP* – true positive samples, *FP* – false positive samples, *TN* – true negative samples, *FN* – false negative samples. *ROC-AUC* is defined as the area under the ROC curve. ROC-AUC of a random model is equal to 0.5, ROC-AUC of a perfect model is 1.

Accuracy measures the proportion of correctly classified objects out of the total number of examples:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}.$$

The *F1-score* is a harmonic mean of *Precision* and *Recall*:

$$F_1 = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}},$$

where

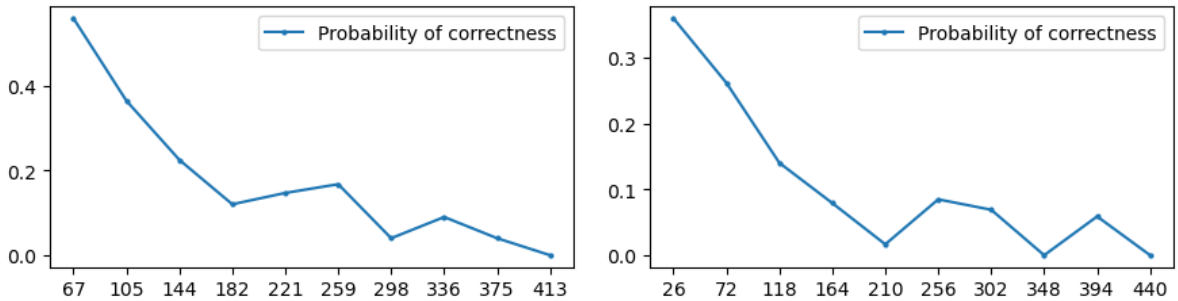
$$Precision = \frac{TP}{TP + FP}, Recal = \frac{TP}{TP + FN}.$$

In multiclass classification, we used weighted average across classes.

To study the ranking ability of the hallucinations detector, we used the *pass@1* metric. pass@1 is a proportion of coding problems from a benchmark for which a Code LLM generated the correct solution passing all the tests, with the restriction that only one solution is executed.

[8] https://xgboost.readthedocs.io/en/latest/index.html

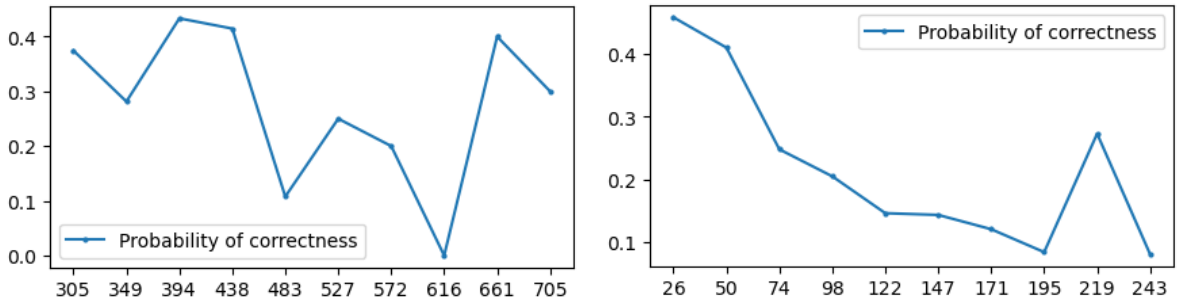
[9] https://github.com/graphml-lab-pwr/lapeigvals



(a) Prompt length, tokens. HumanEval.

(b) Generation length, tokens. HumanEval.

Figure 7: The individual conditional expectations for prompt and generation lengths, CodeLlama-7B.



(a) Prompt length, tokens. MBPP.

(b) Generation length, tokens. MBPP.

Figure 8: The individual conditional expectations for prompt and generation lengths, CodeLlama-7B.

D Examples of Cross-Barcodes

For the MTD feature which strongly distinguishes distribution of classes (Figure 3, (a), left), we show Cross-Barcodes having high and small values of this feature, see Figure 9. See corresponding code samples in Appendix U. Note, that the number of bars (the same as number of tokens in generation) is not a distinguishing statistic; the MTD feature is the average length of a bar. So out MTD features don't have spurious correlations with the length of generated code.

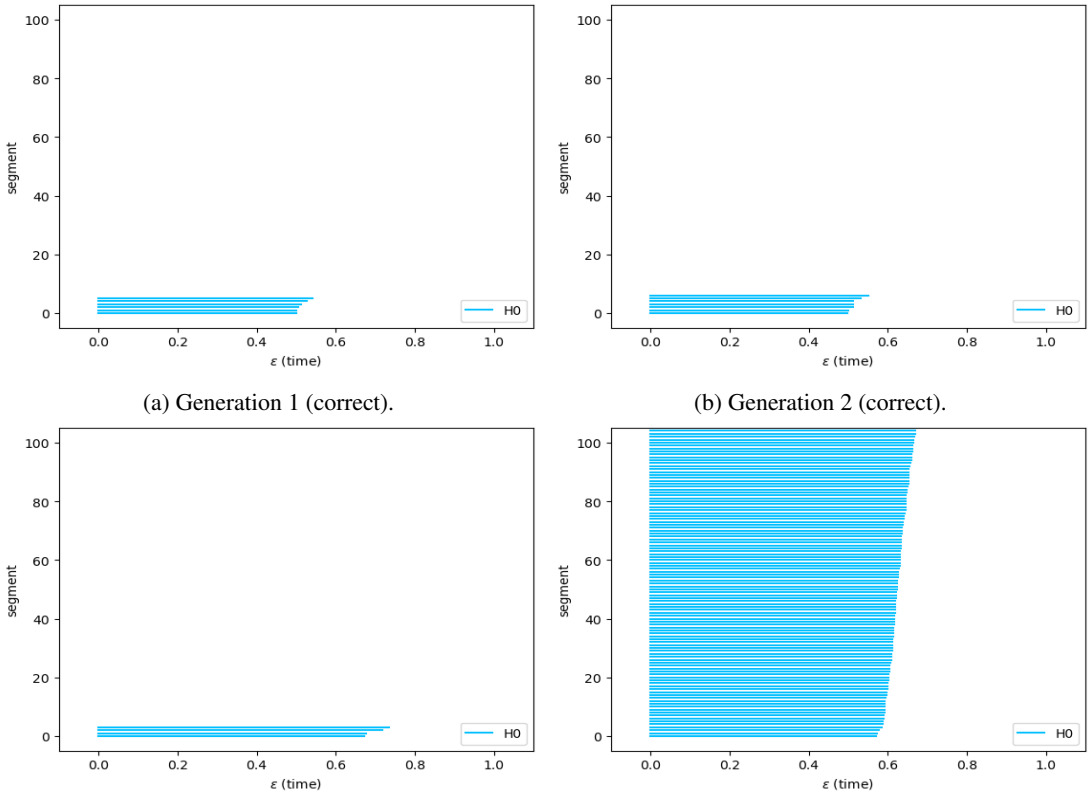
Figure 10 shows examples of Cross-Barcode$_0$ for a fixed attention head. The Cross-Barcode$_1$(P, G) is empty for these attention maps. Correct generations (a), (b) tend to have more and more H_0 bars than not-correct ones (c), (d).

Figure 11 shows examples of Cross-Barcode$_0$, Cross-Barcode$_1$ for a fixed attention head. Correct generations (a), (b) tend to have more and longer H_1 bars than not-correct ones (c), (d).

Attention maps for the corresponding heads are shown in Figures 12, 13.

E Ablation for the Classifier Model Selection

We carried out additional experiments with the feed-forward network (MLP), logistic regression, and support vector classifier (SVC) instead of XGBoost as a classifier for hallucination detection (the rightmost block in Figure 2). We used MLP with two hidden layers of size 256 and ReLU activations. This configuration was selected after moderate optimization of an architecture. For logistic regression and SVC, we tuned the value of regularization strength. Table 18 presents the results. XGBoost offers (a) strong average performance across all code LLMs, (b) negligible training cost ($\approx$ 30 s per fold), and (c) no hyper-parameter tuning in our setting. XGBoost guarantees low computational overhead while providing a single, robust baseline for subsequent work. This experiment demonstrates that the high performance of the proposed method is caused by the relevance of the extracted attention features and not by a specific choice of classifier.



(a) Generation 1 (correct).

(b) Generation 2 (correct).

(c) Generation 3 (not correct, hallucination).

(d) Generation 4 (not correct, hallucination).

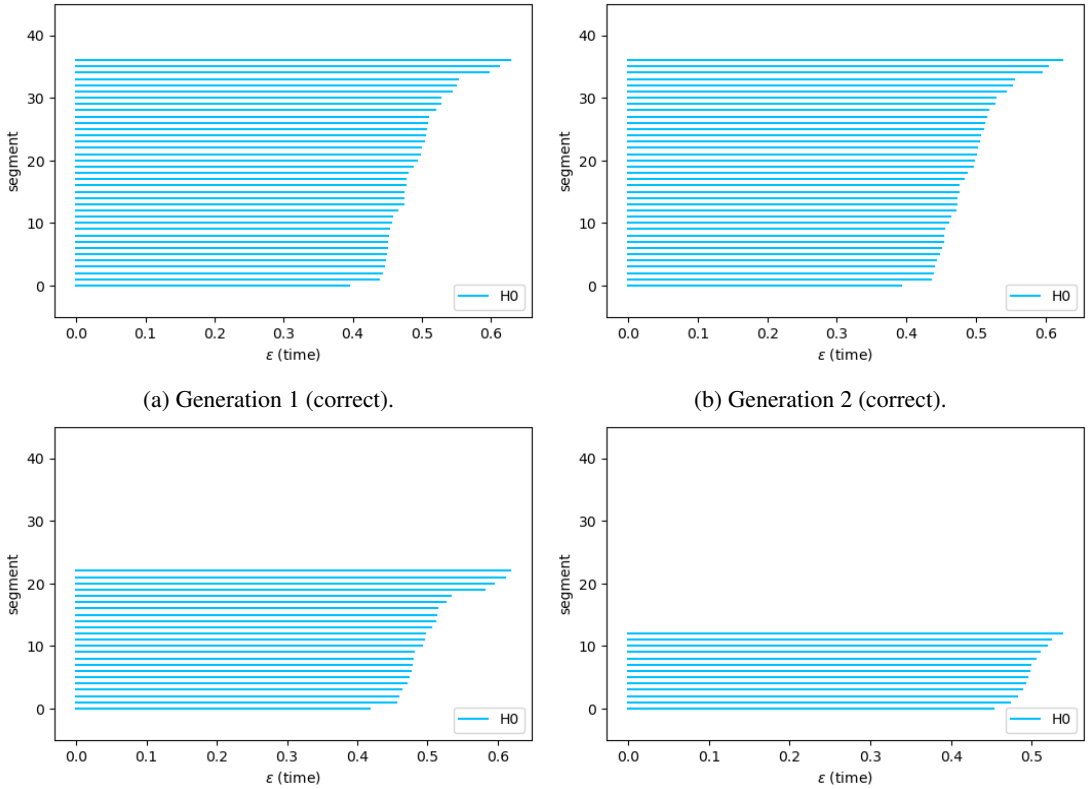
Figure 9: Examples of Cross-Barcode$_0$(P, G), CodeLLama-7B, HumanEval dataset, layer 1, head 17. The number of bars (the same as number of tokens in generation) is not a distinguishing statistic (see Generation 4). The distinguishing statistic is the average length of a bar (normalized MTD).

F Applicability of GNNs to Attention Maps

To train a GNN-based approach on graphs with edge weights obtained from attention matrices, these attention matrices need to be stored. We can estimate the approximate memory footprint to store attention matrices of size (seq_len_k)^2 for a model with n_layers and n_heads for a dataset of size N using the formula: n_layers $\times$ n_heads $\times$ s $\times \sum_{i=1}^{k}$ (seq_len_k)^2 where s is the size of the float type. We assume s = 4 bytes. If one uses only attention matrices from the last layer of the model, we obtain the memory footprint approximately 20.7 - 31.1 Gb for the Human Eval dataset and 36.6 - 53.7 Gb for MBPP (depending on the model). However, to store the attention matrices for all layers and all heads, the memory footprint is about 578.2 - 996.4 Gb for Human Eval and 1023.5 - 1718.7 Gb for MBPP. We highlight that even for datasets of moderate size (i.e. 4100 generations for HumanEval and 2500 generations for MBPP), the memory footprint becomes prohibitively high. Thus, it is not always feasible to store such features. In contrast, in our approach, we do not need to store the attention matrices since we compute all the features immediately during generation. Hence, the size of our training dataset is negligible. Moreover, our approach demonstrates high performance without hyperparameter tuning. Therefore, we believe that the proposed approach has better scalability and is more practical.

G Computational Complexity

We provide an estimate of computational time using the CodeLlama-7B model and HumanEval dataset as an example. The average time taken to generate a solution for one problem without CodeTD is 6.2 sec, with CodeTD 11.5 sec, when topological features are calculated for all layers and heads, which are processed in parallel. Feature computation time during inference can be further decreased if only the most important features are used for classification: according to Section 7.5, it is possible to use only 5% of all attention features without significant loss of classification qual-



(a) Generation 1 (correct).

(b) Generation 2 (correct).

(c) Generation 3 (not correct, hallucination).

(d) Generation 4 (not correct, hallucination).

Figure 10: Examples of Cross-Barcode$_0$(P, G), CodeLLama-7B, HumanEval dataset, problem 14, layer 4, head 18. Cross-Barcode$_1$(P, G) is empty for these attention maps.

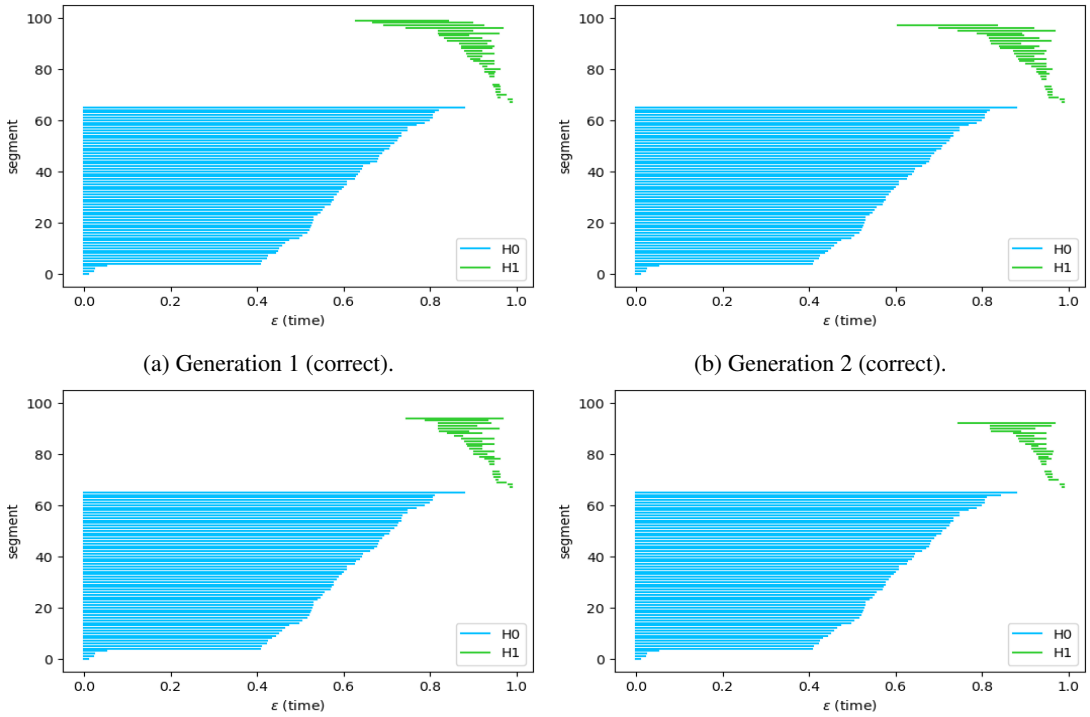
ity The average memory footprint is ≈190MB for HumanEval and ≈544MB for MBPP. This size is a small fraction of GPU footprint during generation.

H On Definition of a Code Hallucination

In our approach, the topological features obtained from attention maps account for the dissimilar structures in the prompt and generation subsets. Our intuition is that a correct solution should correspond to the structure of the prompt as their high-level semantic meanings cohere. Although other reasons behind hallucinations are possible, our approach estimates the correctness of code based only on the internal information flow of the model that does not require additional resources. Nevertheless, the proposed approach can be further integrated into other code hallucination detection tools to achieve better performance. Also, the most popular benchmarks like HumanEval and MBPP check only functional correctness by running functional tests, that is, whether a code solves the corresponding problem as stated in a prompt.

Hallucinations and generalization. The study of hallucinations in LLMs is intrinsically tied to generalization in NLP models. Both challenges stem from the way models learn, represent, and apply knowledge. Improving generalization through robust training, diverse data, and better uncertainty handling reduces hallucinations by ensuring that the models produce contextually appropriate and factually grounded output. In contrast, analyzing hallucinations provides an insight into generalization failures, guiding the development of more reliable NLP systems. This symbiotic relationship underscores the importance of addressing both issues holistically in AI research.

MTD as manifold-topology divergence in value space (per head). For a decoder-only attention head, the output representation at token (i) is an attention-weighted mixture of value vectors $o_i = \sum_{j \le i} a_{ij} v_j$, , so the attention coefficients determine how tokens are embedded into the attended value-representation space of that head. Moreover, in decoder models the attention coefficients can be viewed (under the standard approximation used in our intuition) as tracking similarity structure induced by scalar products among the corresponding token representations that drive value



(a) Generation 1 (correct).

(b) Generation 2 (correct).

(c) Generation 3 (not correct, hallucination).

(d) Generation 4 (not correct, hallucination).

Figure 11: Examples of Cross-Barcode$_0$(G, P), Cross-Barcode$_1$(G, P). CodeLLama-7B, HumanEval dataset, problem 14, layer 15, head 5.

mixing; under norm-concentration / approximate isotropy assumptions (vectors lying on a sphere of roughly fixed radius), this motivates treating $w_{ij} = 1 - a_{ij}$ as an angular-dissimilarity-like transform for the filtration. In this sense, the Cross-Barcode / $\mathrm{MTD}_0(P, G)$ construction can be interpreted as measuring a multiscale topological divergence between the latent manifolds underlying the prompt-token point cloud (P) and the generation-token point cloud (G) in the head's attended value space (i.e., how much scale is needed before generation structures become topologically connected to prompt structures).

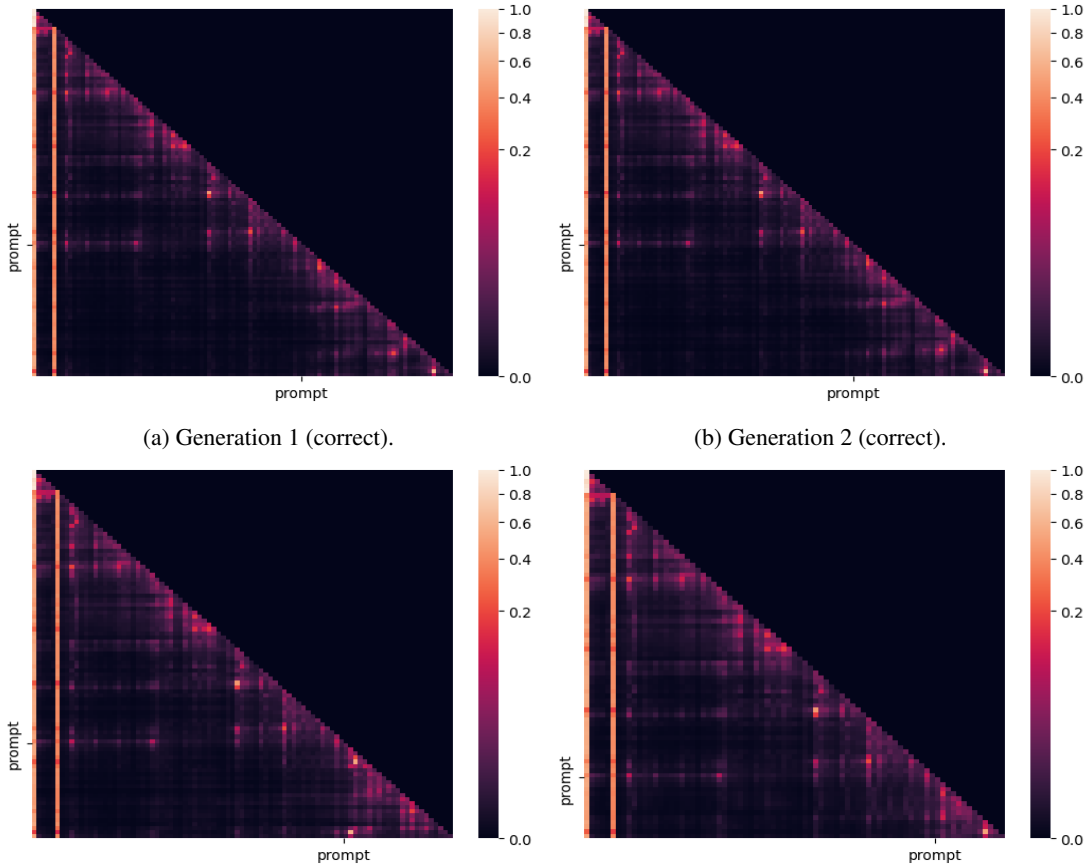
MTD as an area-under-the-curve multiscale integration cost. An equivalent and intuitive H_0 interpretation is to define $U_{\mathrm{comp}}(\tau)$ as the number of connected components in the filtered attention graph at threshold that contain generation tokens but are not yet connected to the prompt set (P). Then $\mathrm{MTD}_0(P, G)$ is the area under this curve across thresholds, i.e., the total lifetime of prompt-disconnected generation components over the filtration. This quantity is not only a measure of delayed prompt grounding, but also a multiscale integration cost: it accumulates how long generation-side components remain disconnected from the prompt-conditioned structure and from one another before merging as the threshold grows. Hence, small MTD_0 indicates early integration (globally coherent, prompt-aligned generation), whereas large MTD_0 indicates persistent fragmentation and delayed integration, which is empirically associated with functional incorrectness.

Stability. By (Cohen-Steiner et al., 2005), Cross-Barcodes satisfy $d_B(\mathbf{B}(W), \mathbf{B}(W')) \leq |W - W'|_\infty$. Hence MTD_k inherits Lipschitz stability under attention perturbations — a property *not enjoyed* by spectral baselines, since eigenvalue ordering can be unstable under near-degeneracies. This is, in our view, a key reason CodeTD outperforms LapEigenvals on transferability (Table 3, 80% of cells).

I Experiments with Larger Models

First, Table 9 demonstrates CodeTD outperforms other baselines even when the model ability to generate correct code candidates increases.

DeepSeek R1 671B. We carried out additional experiments, where code hallucinations are detected by a recent reasoning DeepSeek R1 model having 671 billion parameters with Chain of Thought inference on MBPP dataset. We used the



(a) Generation 1 (correct).

(b) Generation 2 (correct).

(c) Generation 3 (not correct, hallucination).

(d) Generation 4 (not correct, hallucination).

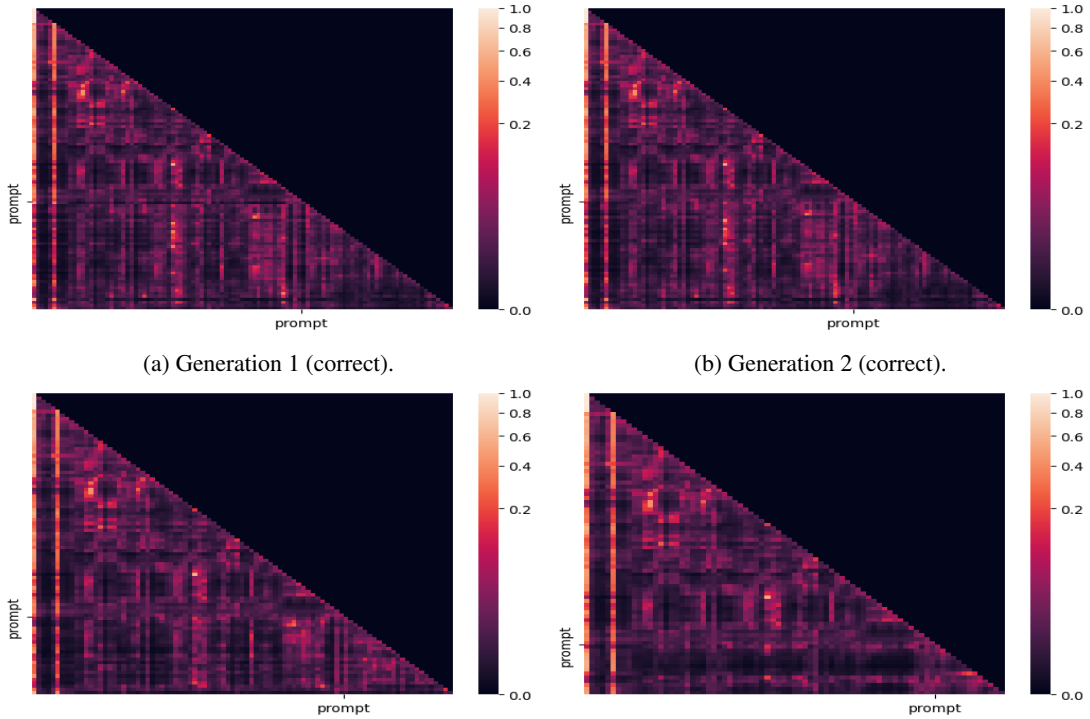
Figure 12: Attention maps. CodeLLama-7B, HumanEval dataset, problem 14, layer 4, head 18.

following prompt:

You are provided with two coding tasks with solutions. The first one is just an example and does not need an assessment. Tell whether the second task is correctly solved by the code provided in the second [BEGIN] [DONE] block. The answer must be Yes or No

For code generated with the DeepSeek-6.7B model, we asked DeepSeek R1 using the above prompt, extracted the final answers (i.e. "Yes" or "No") from the generated responses, and trained the XGboost classifier using these features. The quality of hallucination detection via such zero-shot prompting is in Table 11, row "DeepSeek R1, (671B model)". The quality of the proposed approach is shown in Table 11, row "CodeTD (ours, from 6.7B model)". Note that in this case, we use *only* the attention features of the DeepSeek-6.7B model obtained during code generation. Finally, we combine both types of features to train a classifier and report its performance in Table 11, row "CodeTD (ours, from 6.7B model) + DeepSeek R1". The larger DeepSeek R1 model demonstrates better performance than the classifier trained on attention features. However, by adding an output of DeepSeek R1 to our attention-based features and training the XGBoost classifier, we can achieve the best ROC-AUC score. The study of DeepSeek R1's Chain of Thoughts shows that this LLM is doing verification of code by interpreting Python code step by step for unit tests. This can explain the high accuracy of Deep Seek R1. At the same time, our method opens opportunities for a deeper understanding of inner working and information flow inside transformer models. Our attention features were based on a small 6.7B model in this experiment, however, our features were able to improve the performance of DeepSeek R1.

QwenCoder2.5-32B. Additionally, we performed a similar experiment with QwenCoder2.5-32B. In this case, we use the same QwenCoder2.5-32B to generate code, extract attention features, and evaluate its performance with zero-shot prompting. Table 10 provides the experimental results. The proposed CodeTD is capable of achieving a better detection quality than the zero-shot



(a) Generation 1 (correct).

(b) Generation 2 (correct).

(c) Generation 3 (not correct, hallucination).

(d) Generation 4 (not correct, hallucination).

Figure 13: Attention maps. CodeLLama-7B, HumanEval dataset, problem 14, layer 15, head 5.

prompting, which supports the applicability of the proposed approach to larger models.

CodeJudge. Furthermore, we provide a comparison with CodeJudge (Tong and Zhang, 2024), a code evaluation framework that uses LLM to analyze code functionality and then decide on code correctness. In our comparison, we use CodeLlama-Instruct 34B as an evaluation model to produce binary output to show whether the generated code is correct or not. We report the mean results over 3 runs in accordance with the original setup; see Table 12.

We highlight that for a fair comparison we should consider the setup when CodeJudge is run without reference and CodeTD is run for large 32-34B models. In our work, we did not include reference (i.e. correct solution) to the prompt as this setup is not practical (typically, one does not know the correct solution). Also, since CodeJudge is evaluated with a 34B model, we evaluate CodeTD in a similar way to keep the experimental design of both methods as close as possible. In this comparison, CodeTD outperforms CodeJudge, winning by a large margin for ROC-AUC. However, to further explore the capabilities of the proposed CodeTD, we provide additional comparison when CodeJudge is run with reference and when CodeTD is run for smaller 6.7-7B models. In these cases, CodeTD still outperforms CodeJudge as measured by ROC-AUC.

J Contribution of Individual Heads

We carry out additional experiments with 7B-sized models and MultiPL-E benchmark[10], which is a translation of HumanEval to several popular programming languages; we used Go, Java, Rust, Lua among them. We find that features of some heads have quite a high correlation with the target value (presence of a hallucination) and can be used as individual predictors. In Table 15 we report the ROC-AUC scores of the top-performing features.

K Evaluation with Greedy Decoding

In the main experiments in Section 7.2, we use sampling with temperature 0.8 to generate diverse solutions for each coding problem and obtain a larger train and test samples. However, production systems often use greedy decoding, and in this section we fill this gap. We evaluate the performance of the proposed CodeTD method when both train and test data are generated with greedy decoding. In this setup, the sample sizes are 164 for

[10]https://github.com/nuprl/MultiPL-E

HumanEval and 500 for MBPP with 1 generation per task, and we use cross-validation with 5 folds. Although the sample size is sufficiently smaller than in Section 7.2, the proposed CodeTD achieves high performance across all models and datasets and can be applied to small samples; see Table 19. The proposed CodeTD consistently outperforms CodeJudge (see Table 12) both with greedy decoding and sampling. Moreover, while CodeJudge performance may deteriorate when increasing the sample size with sampling, the proposed approach demonstrates stable improvement.

L Failure Case Study

Table 20 provides a failure case study. For each trained classifier, we analyzed the execution results of the generated code snippets which were incorrectly classified by the classifier. We report the fraction of top-3 most popular code categories w.r.t. number of samples in the test sample. We used the 5-fold stratified group cross-validation, and we report the average values over the 5 folds. The table reveals the most popular failure cases: misclassification of correct codes that passed the tests (referred to as "passed") and misclassification of wrong solutions (i.e., code snippets that did not pass any test and caused an AssertionError). The fraction of other errors (top-3 and others) is sufficiently lower.

M Breakdown by Error Type

We consider a Python exception type after running the functional test as a hallucination type. Here are the common exceptions from the HumanEval and MBPP benchmarks: AssertionError, AttributeError, IndentationError, IndexError, ModuleNotFoundError, NameError, RecursionError, SyntaxError, TypeError, UnboundLocalError, ValueError, ZeroDivisionError, timed out. Most of the errors are due to failing functional tests (AssertionError fraction), see Table 5.

N Individual features detect error types

We carried out additional experiments and found that some of the proposed features can distinguish logical errors (AssertionError) vs. syntax and runtime errors (SyntaxError, ZeroDivisionError, NameError, etc.). Here are the results for CodeLLama-7B, HumanEval dataset:

- answer diag.attention, layer 29, head 24: ROC_AUC = 0.67
- MTD_0(P, G) layer 0, head 4: ROC_AUC = 0.66
- MTD_0(P, G) layer 1, head 10: ROC_AUC = 0.66

O Manifold Topology Divergence

MTD (Manifold Topology Divergence) (Barannikov et al., 2021) is a tool of TDA that can be used to evaluate the "dissimilarity" between two sets of vertices in a weighted graph $\mathcal{G} = (V, E, W)$ or, in other words, to which degree one set of vertices is covered by another set.

Let a set of vertices $V = P \sqcup G$, be split into disjoint sets P, G. We consider a nested sequence of graphs $\mathcal{G}_0 \subset \ldots \subset \mathcal{G}_i \subset \mathcal{G}_{i+1} \subset \ldots \subset \mathcal{G}$ in the following way. $\mathcal{G}_0$ has all the vertices P, G and all the edges that connect the vertices of P. The sequence $\mathcal{G}_i$ is obtained by adding the rest of the edges one by one in ascending order of their weights; see Figure 14. During this process, graph topology naturally changes: connected components are merged, cycles appear and disappear, etc. This process is rigorously described by the theory of *persistence barcodes* (Chazal and Michel, 2017). Each topological feature, such as connected component or cycle, has a "birth time" and a "death time", by a corresponding edge weight. The multiset of these birth-death pairs (intervals) altogether is called a Cross-Barcode$_k$, see Figure 15. Here k is an index of a *persistence homology*, each of them reflects a kind of topological feature: 0 - connected components, 1 - cycles, 2 - voids, etc. MTD_k is an integral characteristic of a Cross-Barcode$_k$ and it is defined as a sum of birth-death intervals' lengths

$$\text{MTD}_k = \sum_{(b_i,d_i)\in CrossBarcode_k} d_i - b_i.$$

The higher MTD_k is, the greater is the "dissimilarity" between sets of tokens. Note that according to the definition, MTD_k is not symmetric. MTD_k, as a kind of persistence barcode, enjoys stability w.r.t. perturbations of weights (Cohen-Steiner et al., 2005).

P On importance of cycles

Graph cycles through the prompt graph P are abundant, since P is fully connected. Cross-Barcode$_k$(P, G) is computed on the clique complex $K(G_k)$, not on the graph itself. Because $K(G_0)$ contains the full simplex on P, and the clique complex of a

Code LLM	HumanEval		MBPP	
	Assertion error, %	Other errors, %	Assertion error, %	Other errors, %
StarCoder2-7B	71.5	28.5	85.1	14.9
CodeLlama-7B	79.3	20.7	83.4	16.5
DeepSeek-Coder-6.7B-base	81.7	18.3	86.3	13.4
Qwen2.5-Coder-7B	62.9	37.1	66.9	33.1
Magicoder-S-DS-6.7B	83.1	16.9	90.6	9.4

Table 6: Breakdown of code LLMs by error type.

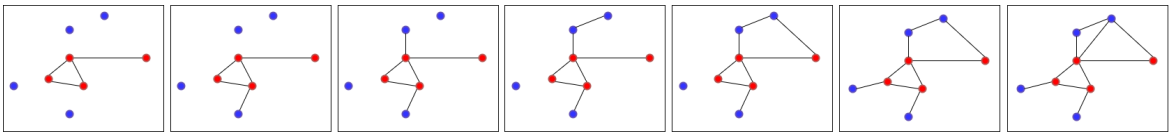


Figure 14: An example of MTD evaluation for a graph having two groups of vertices – red and blue. (0): initially, only edges connecting red vertices are present. (1)-(6): the rest of edges are added sequentially in an ascending order by their weights. While adding edges, connected components merge with each other. These moments are depicted by H_0 bars in Fig. 15. At moment (4) a cycle appears, at moment (6) this cycle disappears. These moments are depicted by the H_1 bar in Fig. 15.

complete graph is contractible, every P-internal 1-cycle is already filled by a 2-chain at $k = 0$:

$$\dim H_1(K(G_0)) = 0.$$

A non-trivial 1-cycle is therefore a **detour loop**: $p_1 \to g_1 \to \dots g_k \to p_2$, i.e., two prompt nodes joined through a G-path while a path $p_1 \to p_2$ in P is already present. The cycle is "born" when the last-detour edge enters the filtration, and cycle "dies" when its nodes become pairwise connected. See Fig. 14 for step-by-step illustration for cycles detection. Cycle persistence is therefore how long precisely generation tokens fail to follow the prompt's local geometry: they connect prompt nodes p_1, p_2 via a G-path but remain weakly aligned with the prompt themselves. Correct generations bind tightly, detour loops collapse fast, MTD_1 is small; incorrect generations drift, detour loops persist, MTD_0 is large. MTD_0 relates to connected component measures whether G-components attach to P at all; MTD_1 measures *how cleanly*. This is an explanation of results from (Barannikov et al., 2021).

Q Comparison with (Bazarova et al., 2025)

(Bazarova et al., 2025) introduced TOHA, a Topology-based Hallucination detector, which leverages a topological divergence metric to quantify the structural properties of graphs induced by attention matrices in Retrieval-Augmented Generation (RAG) settings where models may generate outputs unsupported by the provided context. The TDA-on-attention research line on which our work builds – (Barannikov et al., 2021), (Kushnareva et al., 2021), (Cherniavskii et al., 2022), (Tulchinskii et al., 2022) – is already cited in the manuscript. We welcome the comparison: it sharpens what is novel in CodeTD. The central methodological difference is that (Bazarova et al., 2025) operate head-by-head: per attention head they compute, rank heads by individual discriminative power on a held-out split, and use a small selected subset as the predictor. This select-then-aggregate pipeline bakes in a strong prior - that signal lives in a few "best" heads — and discards the rest of the model. CodeTD takes the opposite stance: extract the complete set of topological features across every layer and head and let a gradient-boosted classifier learn the joint structure. Per head selection is a degenerate special case (a rank-1 sparse classifier on a single slice). Section 7.5 / Fig. 4 show the optimal sparse subset is not "a few best heads" but a structured 5% pattern across layers, heads, and homological dimensions that XGBoost discovers automatically. Table 7 presents a detailed comparison.

Our work provides a theoretical contribution beyond using MTD merely as a feature. Appendix H develops an interpretation specific to decoder-only transformer attention, viewing MTD as a topologi-

Axis	(Bazarova et al., 2025)	CodeTD (Ours)
Task	RAG hallucination in QA/summarization	Functional incorrectness of code
Ground Truth	Text-level factuality	Execution-based labels
Role of the Prompt	Knowledge context	Formal specification
MTD Orientation	(P, G) only	Asymmetric: both (P, G) and (G, P) used
Homology Dimensions	H_0	H_0 and H_1
Aggregation	Per-head selection	Joint GBDT over all features
Diagonal Self-Attention	—	Included as per-vertex features complementing edge-level topology
Models	General LLMs	10 Code LLMs (1.5B – 34B)
Languages	Natural language	Prompt in natural language; generation in 5 programming languages: Python, Java, Go, Rust, Lua.
Cross-language Stability	—	The same predictive (layer, head) features transfer across all 5 programming languages (see Tab. 11).
Fine-grained Error Types	—	Multi-class detection of Python exceptions (see Tab. 5, App. N).
Ranking of samples	—	pass@1 gains up to +17.6 pp.

Table 7: Detailed comparison with (Bazarova et al., 2025).

cal invariant of distribution supports in the head's attended value space. AttnLogDet, AttnEigvals, and LapEigvals (Binkowski et al., 2025) are the strongest published attention-based detectors in the broader research; CodeTD outperforms them on 84% of (model, language) cells across HumanEval, MBPP, BigCodeBench, and four MultiPL-E languages. The gap widens under transfer (Tab. 3): per-head methods like LapEigvals drop – 13-20 ROC-AUC pts cross-benchmark, while CodeTD remains more stable.

R On statistical significance for BCB

For BCB, CodeTD significantly outperforms LapEigvals on **3 of 5** under a paired one-sided t-test across folds (Shapiro-Wilk does not reject normality, $p > 0.48$ in all cells):

- DeepSeek-Coder-6.7B: $p = 0.003$
- Qwen2.5-Coder-7B: $p = 0.022$
- Magicoder-S-DS-6.7B: $p = 0.041$

CodeLlama-7B is a tie ($p = 0.225$, mean favors CodeTD; $n = 5$ folds gives limited power). The StarCoder2-7B cell (83.7 ± 4.3 vs 85.3 ± 2.4) is the single case where LapEigvals wins. Combining the five paired tests via Fisher's method yields $p = 0.017$ in favor of CodeTD globally on BCB.

S On streaming variant of CodeTD

CodeTD admits a streaming variant. Let G_t be the tokens generated by step t. Because MTD is built from a *threshold filtration* of the prompt-generation graph $\mathcal{G}_t = (P \sqcup G_t, E_t, W_t)$, we can compute $\text{MTD}_k(P, G_t)$ at every $t = k, 2k, 3k, \ldots$ and trigger an early-stop if hallucination probability is larger than some threshold. Two facts make this *efficient*: (i) the union-find used in H_0 persistence supports incremental edge insertion in near-constant amortized time, so MTD_0 can be updated in $O(|E_{t+k}| - |E_t|)$ per chunk; (ii) only $\sim 5\%$ of heads need to be tracked. This is a compelling extension for long generations (e.g., BCB and repository-level code).

Experimental validation. We generate 100, 300, or 500 tokens, and train CodeTD hallucination detector on partial generations. We evaluated this approach using the DeepSeek-Coder-6.7B model on the BigCodeBench benchmark. Table 8 shows results. The results show that the number of generated tokens can be reduced at the cost of a slight decrease in classification performance.

T Licenses

Licenses of pretrained models and benchmarks used in this paper permit use for research purposes.

# tokens	ROC-AUC
100	0.633 ± 0.034
300	0.673 ± 0.012
500	0.656 ± 0.048
Full generation	0.69 ± 0.02

Table 8: Performance of CodeTD when training of quanta generation level.

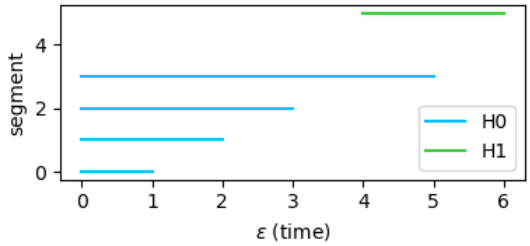


Figure 15: Cross-Barcode for a filtration from Fig. 14.

Method	**HE**
CodeLlama-34B	
Prompt Len.	57.2 ± 6.6
Gen. Len.	62.7 ± 1.8
Mean Log. Prob.	72.9 ± 3.4
CodeT5-base ft.	54.7 ± 5.5
CodeTD (ours)	$\mathbf{84.1 \pm 3.2}$
Qwen2.5-Coder-32B	
Prompt Len.	53.2 ± 9.2
Gen. Len.	58.0 ± 3.1
Mean Log. Prob.	67.0 ± 4.0
CodeT5-base ft.	53.3 ± 4.5
CodeTD (ours)	$\mathbf{85.0 \pm 2.7}$
DeepSeek-Coder-33B	
Prompt Len.	53.3 ± 6.5
Gen. Len.	56.5 ± 4.2
Mean Log. Prob.	69.9 ± 2.6
CodeT5-base ft.	57.6 ± 3.9
CodeTD (ours)	$\mathbf{88.2 \pm 3.0}$

Table 9: Code hallucination detection for HumanEval dataset for larger models.

Method	**HE**
QwenCoder2.5-32B	
CodeTD (ours)	$\mathbf{85.0 \pm 2.7}$
Zero-shot prompt	67.4 ± 3.5

Table 10: HumanEval features for larger models. See Section I for details.

Method	**MBPP**
CodeTD (ours) (from DeepSeek-Coder-6.7B)	81.2 ± 3.7
DeepSeek R1 (671B model)	89.9 ± 2.7
CodeTD (ours) (from DeepSeek-Coder-6.7B) + DeepSeek R1 (671B)	$\mathbf{93.2 \pm 3.4}$

Table 11: MBPP features for larger models. See Section I for details.

Method	**Greedy Dec.**	**T=0.8**
StarCoder2-7B		
CodeJudge w/o ref.	70.6 ± 4.7	69.6 ± 1.9
CodeJudge w/ ref.	75.8 ± 4.7	71.1 ± 2.8
CodeTD (ours)	$\mathbf{82.2 \pm 4.3}$	$\mathbf{82.5 \pm 2.1}$
CodeLlama-7B		
CodeJudge w/o ref.	71.3 ± 4.1	72.1 ± 1.3
CodeJudge w/ ref.	73.0 ± 3.3	71.9 ± 3.6
CodeTD (ours)	$\mathbf{78.1 \pm 4.3}$	$\mathbf{85.6 \pm 3.9}$
DeepSeekCoder-6.7B		
CodeJudge w/o ref.	69.2 ± 3.2	68.4 ± 1.1
CodeJudge w/ ref.	68.8 ± 7.9	68.3 ± 3.1
CodeTD (ours)	$\mathbf{83.9 \pm 8.4}$	$\mathbf{86.4 \pm 1.8}$
QwenCoder-7B		
CodeJudge w/o ref.	67.2 ± 5.7	69.1 ± 3.0
CodeJudge w/ ref.	70.4 ± 7.4	69.3 ± 2.6
CodeTD (ours)	$\mathbf{73.7 \pm 5.2}$	$\mathbf{81.6 \pm 4.7}$
Magicoder-S-DS-6.7B		
CodeJudge w/o ref.	55.3 ± 2.7	60.4 ± 1.8
CodeJudge w/ ref.	57.9 ± 7.1	61.2 ± 2.9
CodeTD (ours)	$\mathbf{68.3 \pm 3.5}$	$\mathbf{79.3 \pm 3.5}$
CodeLlama-34B		
CodeJudge w/o ref.	65.3 ± 3.0	70.7 ± 5.1
CodeJudge w/ ref.	68.7 ± 3.8	73.2 ± 4.4
CodeTD (ours)	$\mathbf{75.0 \pm 5.1}$	$\mathbf{84.1 \pm 3.2}$
DeepSeek-Coder-33B		
CodeJudge w/o ref.	72.0 ± 7.2	69.0 ± 2.8
CodeJudge w/ ref.	75.8 ± 5.6	70.9 ± 1.3
CodeTD (ours)	$\mathbf{86.5 \pm 7.2}$	$\mathbf{88.2 \pm 3.0}$
Qwen2.5-Coder-32B		
CodeJudge w/o ref.	61.6 ± 7.3	66.5 ± 3.5
CodeJudge w/ ref.	62.0 ± 7.5	68.2 ± 3.7
CodeTD (ours)	$\mathbf{74.0 \pm 6.4}$	$\mathbf{85.0 \pm 2.7}$

Table 12: Comparison with CodeJudge A. S. (Tong and Zhang, 2024) on HumanEval in two setups: code generation with greedy decoding and sampling with T=0.8.

Method	2-Shot
DeepSeek-Coder-6.7B	
Prompt Len.	56.0 ± 2.0
Gen. Len.	54.4 ± 1.3
Mean Log. Prob.	61.9 ± 1.7
Self-Eval	50.0 ± 0.0
Interrogate-LLM	61.1 ± 3.1
CodeTD (ours)	$\mathbf{81.5 \pm 2.0}$
Qwen2.5-Coder-7B	
Prompt Len.	53.5 ± 2.9
Gen. Len.	56.9 ± 3.3
Mean Log. Prob.	56.6 ± 3.4
Self-Eval	68.4 ± 1.0
Interrogate-LLM	56.6 ± 1.5
CodeTD (ours)	$\mathbf{78.1 \pm 3.1}$

Table 13: Code hallucination detection with attention features for MBPP dataset with increasing complexity of prompt: two-shot prompts.

Method	HE → MBPP	MBPP → HE
QwenCoder2.5-3b		
Diag. Feat.	58.7	61.3
- w/ MTD 0-dim	60.6	63.8
- w/ MTD 0,1-dim	**60.8**	**65.7**
StarCoder2-7B		
Diag. Feat.	67.5	**68.3**
- w/ MTD 0-dim	**71.4**	67.4
- w/ MTD 0,1-dim	67.7	66.0
DeepSeek-Coder-6.7B		
Diag. Feat.	65.5	63.5
- w/ MTD 0-dim	64.5	69.8
- w/ MTD 0,1-dim	**69.9**	**72.4**
QwenCoder2.5-7B		
Diag. Feat.	64.6	62.9
- w/ MTD 0-dim.	70.3	60.7
- w/ MTD 0,1-dim	**70.9**	**65.8**

Table 14: Dataset transferability attention feature ablation.

Feature	HE					MBPP
	Python	**Go**	**Rust**	**Java**	**Lua**	**Python**
StarCoder2-7B						
avg. prompt's self-attention, layer 14, head 0	70.8	76.8	74.6	68.9	70.3	55.2
avg. prompt's self-attention, layer 15, head 5	71.1	78.4	75.1	72.9	72.4	58.1
avg. prompt's self-attention, layer 23, head 20	69.2	72.2	74.2	64.7	67.7	50.7
CodeLlama-7B						
-MTD_1(P, G)/|P|, layer 15, head 27	67.6	71.1	73.7	71.0	–	66.1
avg. prompt's self-attention, layer 7, head 22	74.6	75.7	76.5	69.7	–	63.1
MTD_0(P, G)/|P|, layer 11, head 23	69.1	71.0	71.9	65.4	–	69.1
DeepSeek-Coder-6.7B						
MTD_0(P, G)/|P|, layer 30, head 11	67.5	65.7	67.0	66.3	69.1	58.1
avg. prompt's self-attention, layer 12, head 17	65.0	73.6	70.0	69.8	69.8	58.2
avg. prompt's self-attention, layer 12, head 23	64.1	71.0	66.8	64.4	67.6	58.2
Qwen2.5-Coder-7B						
avg. generation's self-attention, layer 24, head 2	65.8	60.2	58.8	60.2	66.7	60.8
MTD_0(P, G)/|P|, layer 11, head 5	66.2	68.8	59.5	61.6	72.7	59.4
avg. prompt's self-attention, layer 9, head 26	65.4	75.8	66.9	67.0	71.1	61.0
Magicoder-S-DS-6.7B						
MTD_0(P, G)/|P|, layer 30, head 11	68.8	70.2	60.0	65.1	63.8	58.2
avg. prompt's self-attention, layer 13, head 13	63.1	64.9	69.0	67.3	63.2	58.8
avg. prompt's self-attention, layer 12, head 17	63.0	63.2	68.0	67.1	60.8	58.9

Table 15: Top-performing features across several programming languages and benchmarks.

Model	**HumanEval**		**MBPP**		**MultiPL-E**				**BCB**
	Python		Python		Java	Go	Rust	Lua	Python
	GD	T = 0.8	GD	T = 0.8	T = 0.8	T = 0.8	T = 0.8	T = 0.8	T = 0.8
StarCoder2-7B	35.4	28.9	46.2	42.8	24.5	17.5	20.9	19.1	5.9
CodeLlama-7B	29.3	25.9	38.8	35.2	25.8	17.6	20.8	0.0	21.5
DeepSeek-Coder-6.7B	48.8	40.3	59.2	52.6	33.5	23.6	28.7	16.6	32.5
Qwen2.5-Coder-7B	57.3	47.8	58.6	52.1	22.7	11.9	22.6	23.5	36.5
Magicoder-S-DS-6.7B	73.2	65.5	63.2	61.3	48.9	40.3	44.2	34.6	36.6

Table 16: Characteristics of generated data, pass@1. We use two regimes to generate solutions – greedy decoding (GD) and sampling with temperature $T = 0.8$.

	StarCoder2-7B	**DeepSeek-Coder-6.7B**	**Qwen2.5-Coder-7B**	**Magicoder-S-DS-6.7B**	**DeepSeekCoder-33B**
StarCoder2-7B	82.5 ± 2.1	89.5 ± 2.7	87.0 ± 2.9	$\mathbf{91.7 \pm 2.1}$	$\underline{90.3 \pm 1.1}$
DeepSeek-Coder-6.7B	81.0 ± 1.7	86.4 ± 1.8	85.7 ± 1.9	$\mathbf{89.0 \pm 1.6}$	$\underline{87.9 \pm 2.3}$
Qwen2.5-Coder-7B	80.4 ± 2.5	85.3 ± 2.0	81.6 ± 4.7	$\mathbf{87.7 \pm 1.6}$	$\underline{86.3 \pm 3.1}$
Magicoder-S-DS-6.7B	70.9 ± 3.4	74.7 ± 3.5	79.0 ± 1.3	$\underline{79.3 \pm 3.5}$	$\mathbf{81.2 \pm 4.1}$

Table 17: Cross-model transferability: rows correspond to code-generating models, columns correspond to feature-extracting models.

Classifier	HE	MBPP
StarCoder2-7B		
XGBoost	82.5 ± 2.1	81.4 ± 3.6
MLP	80.9 ± 4.7	81.1 ± 1.7
Logistic Regression	84.3 ± 3.9	$\mathbf{82.9 \pm 2.3}$
SVC	$\mathbf{84.9 \pm 4.0}$	82.3 ± 2.6
CodeLlama-7B		
XGBoost	$\mathbf{85.6 \pm 3.9}$	$\mathbf{83.4 \pm 3.3}$
MLP	81.8 ± 7.2	81.6 ± 2.2
Logistic Regression	81.8 ± 7.1	82.6 ± 2.1
SVC	83.2 ± 5.4	82.4 ± 1.2
DeepSeek-Coder-6.7B		
XGBoost	86.4 ± 1.8	81.2 ± 3.7
MLP	84.3 ± 2.4	81.7 ± 1.1
Logistic Regression	85.8 ± 3.2	$\mathbf{81.8 \pm 2.4}$
SVC	$\mathbf{87.0 \pm 2.4}$	81.4 ± 1.6
Qwen2.5-Coder-7B		
XGBoost	$\mathbf{81.6 \pm 4.7}$	80.9 ± 3.5
MLP	81.3 ± 1.6	79.5 ± 2.0
Logistic Regression	80.0 ± 1.6	$\mathbf{81.0 \pm 1.9}$
SVC	79.6 ± 1.7	$\mathbf{81.0 \pm 1.7}$
Magicoder-S-DS-6.7B		
XGBoost	79.3 ± 3.5	80.7 ± 2.5
MLP	76.5 ± 3.3	78.6 ± 3.6
Logistic Regression	$\mathbf{81.7 \pm 1.6}$	81.3 ± 2.8
SVC	80.5 ± 2.4	$\mathbf{82.5 \pm 2.6}$

Table 18: Ablation study for the choice of a classification model in CodeTD.

Method	HE	MBPP
StarCoder2-7B		
Prompt. Len.	68.9 ± 8.7	52.0 ± 8.1
Gen. Len.	55.6 ± 7.2	49.4 ± 2.4
Mean Log. Prob.	62.0 ± 10.7	57.1 ± 8.1
Pylint	56.7 ± 2.6	54.5 ± 0.9
CodeT5-base ft.	60.2 ± 4.8	56.5 ± 5.4
Self-Eval	52.0 ± 2.6	61.2 ± 4.8
AttnLogDet	70.6 ± 7.5	72.8 ± 2.8
AttnEigvals	73.6 ± 7.5	62.5 ± 6.3
LapEigvals	76.4 ± 4.4	74.5 ± 4.5
CodeTD (ours)	$\mathbf{82.2 \pm 4.3}$	$\mathbf{79.2 \pm 6.1}$
CodeLlama-7B		
Prompt. Len.	53.2 ± 10.3	61.8 ± 4.2
Gen. Len.	59.9 ± 5.6	54.6 ± 5.7
Mean Log. Prob.	56.6 ± 14.0	64.5 ± 6.2
Pylint	55.3 ± 3.0	53.4 ± 1.6
CodeT5-base ft.	64.1 ± 7.1	54.7 ± 8.0
Self-Eval	47.9 ± 3.7	50.0 ± 0.0
AttnLogDet	74.5 ± 7.7	72.7 ± 3.9
AttnEigvals	$\mathbf{81.1 \pm 6.2}$	69.0 ± 7.5
LapEigvals	73.3 ± 7.7	77.0 ± 4.2
CodeTD (ours)	78.1 ± 4.3	$\mathbf{82.8 \pm 3.2}$
DeepSeek-Coder-6.7B		
Prompt. Len.	59.1 ± 6.6	52.0 ± 6.2
Gen. Len.	45.5 ± 7.9	55.2 ± 5.4
Mean Log. Prob.	64.1 ± 6.6	63.5 ± 4.3
Pylint	52.3 ± 2.0	52.6 ± 2.7
CodeT5-base ft.	58.4 ± 10.3	49.4 ± 3.1
Self-Eval	52.9 ± 6.4	50.0 ± 0.0
AttnLogDet	80.3 ± 6.0	76.4 ± 3.7
AttnEigvals	79.5 ± 8.7	72.9 ± 4.6
LapEigvals	79.5 ± 5.7	77.2 ± 2.3
CodeTD (ours)	$\mathbf{83.9 \pm 8.4}$	$\mathbf{79.3 \pm 2.4}$
Qwen2.5-Coder-7B		
Prompt. Len.	53.6 ± 8.7	57.5 ± 7.2
Gen. Len.	54.0 ± 11.0	54.6 ± 3.8
Mean Log. Prob.	54.6 ± 8.7	61.7 ± 6.6
Pylint	64.0 ± 3.0	61.6 ± 4.0
CodeT5-base ft.	59.0 ± 7.3	54.1 ± 4.2
Self-Eval	72.1 ± 7.8	65.7 ± 5.0
AttnLogDet	61.9 ± 8.3	68.9 ± 4.2
AttnEigvals	63.7 ± 8.2	65.7 ± 5.1
LapEigvals	65.9 ± 12.3	72.1 ± 4.1
CodeTD (ours)	$\mathbf{73.7 \pm 5.2}$	$\mathbf{74.0 \pm 2.7}$
Magicoder-S-DS-6.7B		
Prompt. Len.	53.0 ± 6.1	58.3 ± 4.1
Gen. Len.	59.0 ± 4.8	52.1 ± 2.3
Mean Log. Prob.	61.4 ± 4.8	61.1 ± 4.3
Pylint	50.8 ± 1.5	50.8 ± 0.7
CodeT5-base ft.	57.8 ± 15.3	54.5 ± 7.7
Self-Eval	57.3 ± 8.6	49.1 ± 2.4
AttnLogDet	64.5 ± 8.5	75.7 ± 2.9
AttnEigvals	63.1 ± 8.4	72.7 ± 5.4
LapEigvals	66.9 ± 6.2	78.1 ± 3.6
CodeTD (ours)	$\mathbf{68.3 \pm 3.5}$	$\mathbf{80.6 \pm 1.9}$

Table 19: ROC-AUC of code hallucination detection for generation with greedy decoding.

Model	HE			MBPP		
	Top-1	**Top-2**	**Top-3**	**Top-1**	**Top-2**	**Top-3**
StarCoder2-7B	P 17.2 ± 4.5	AE 2.9 ± 2.2	NE 0.8 ± 0.5	P 15.0 ± 3.5	AE 9.7 ± 2.2	NE 0.7 ± 0.4
CodeLLama-7B	P 14.6 ± 3.6	AE 3.8 ± 2.7	NE 0.2 ± 0.2	P 15.5 ± 1.6	AE 7.4 ± 1.2	NE 0.5 ± 0.4
DeepSeekCoder-6.7B	P 14.6 ± 2.9	AE 4.9 ± 1.2	IE 0.5 ± 0.6	AE 12.6 ± 3.2	P 11.1 ± 3.8	NE 0.7 ± 0.5
QwenCoder-7B	P 16.0 ± 5.4	AE 6.9 ± 2.8	NE 2.1 ± 0.7	P 11.3 ± 0.5	AE 10.8 ± 1.4	NE 1.3 ± 0.5
Magicoder-6.7B	AE 12.1 ± 4.8	P 10.4 ± 4.8	IE 0.7 ± 0.6	AE 14.5 ± 4.4	P 9.2 ± 2.3	TE 0.6 ± 0.5

Table 20: Failure case study. “P” - Passed, “AE” - AssertionError, “NE” - NameError, “IE” - IndexError, “TE” - TypeError.

Method	Java	Go	Rust	Lua
		StarCoder2-7B		
AttnLogDet	80.7 ± 6.2	82.4 ± 4.8	77.8 ± 5.9	74.6 ± 3.0
AttnEigvals	81.6 ± 6.2	80.4 ± 4.1	75.2 ± 8.5	77.8 ± 7.8
LapEigvals	81.6 ± 5.4	82.5 ± 3.3	77.1 ± 5.3	78.8 ± 5.1
CodeTD (ours)	$\mathbf{82.5 \pm 5.6}$	$\mathbf{86.6 \pm 4.6}$	$\mathbf{82.5 \pm 5.8}$	$\mathbf{82.0 \pm 2.8}$
		CodeLlama-7B		
AttnLogDet	$\mathbf{85.2 \pm 4.4}$	55.5 ± 10.4	63.4 ± 10.3	–
AttnEigvals	78.2 ± 9.6	66.7 ± 10.7	76.7 ± 12.8	–
LapEigvals	75.7 ± 8.7	68.6 ± 16.6	74.3 ± 9.2	–
CodeTD (ours)	76.8 ± 6.1	$\mathbf{81.9 \pm 6.2}$	$\mathbf{77.5 \pm 10.8}$	–
		DeepSeek-Coder-6.7B		
AttnLogDet	80.1 ± 5.5	81.9 ± 3.4	78.3 ± 4.2	74.7 ± 7.8
AttnEigvals	78.2 ± 5.8	79.0 ± 5.2	74.5 ± 4.4	84.6 ± 3.8
LapEigvals	83.1 ± 4.5	$\mathbf{89.0 \pm 2.5}$	82.1 ± 5.8	86.1 ± 2.7
CodeTD (ours)	$\mathbf{84.5 \pm 4.8}$	85.2 ± 3.1	$\mathbf{82.3 \pm 7.8}$	$\mathbf{86.2 \pm 4.7}$
		Qwen2.5-Coder-7B		
AttnLogDet	65.7 ± 9.1	76.7 ± 4.5	72.4 ± 5.3	80.3 ± 4.0
AttnEigvals	78.3 ± 2.9	85.9 ± 4.0	84.0 ± 3.2	85.8 ± 2.5
LapEigvals	80.8 ± 3.7	87.9 ± 2.8	79.3 ± 5.4	87.3 ± 2.4
CodeTD (ours)	$\mathbf{81.7 \pm 4.6}$	$\mathbf{91.1 \pm 1.7}$	$\mathbf{87.5 \pm 2.3}$	$\mathbf{90.1 \pm 3.2}$
		Magicoder-S-DS-6.7B		
AttnLogDet	74.5 ± 3.0	70.2 ± 4.7	69.8 ± 3.9	68.8 ± 3.7
AttnEigvals	69.2 ± 5.8	70.9 ± 2.9	69.9 ± 6.9	75.4 ± 2.7
LapEigvals	76.2 ± 1.5	$\mathbf{84.0 \pm 2.6}$	$\mathbf{74.8 \pm 5.3}$	79.6 ± 0.3
CodeTD (ours)	$\mathbf{77.8 \pm 4.5}$	80.7 ± 1.9	$\mathbf{74.8 \pm 6.1}$	$\mathbf{80.1 \pm 2.1}$

Table 21: ROC-AUC for different programming languages from MultiPL-E dataset: mean and standard deviation over 5 folds.

Model	HE	MBPP
QwenCoder2.5-1.5b		
Diag. Feat.	83.2 ± 4.3	78.8 ± 2.2
- w/ MTD 0-dim	84.7 ± 3.8	81.7 ± 1.1
- w/ MTD 0,1-dim	$\mathbf{85.0 \pm 3.9}$	$\mathbf{82.5 \pm 1.0}$
QwenCoder2.5-3b		
Diag. Feat.	75.5 ± 6.6	76.5 ± 1.6
- w/ MTD 0-dim	80.8 ± 2.5	78.8 ± 1.5
- w/ MTD 0,1-dim	$\mathbf{81.3 \pm 2.5}$	$\mathbf{79.3 \pm 1.9}$
StarCoder2-7B		
Diag. Feat.	$\mathbf{83.4 \pm 2.0}$	80.1 ± 3.3
- w/ MTD 0-dim	$\mathbf{83.4 \pm 2.9}$	80.4 ± 3.3
- w/ MTD 0,1-dim	82.5 ± 2.1	$\mathbf{81.4 \pm 3.6}$
DeepSeek-Coder-6.7B		
Diag. Feat.	84.2 ± 1.5	81.0 ± 3.5
- w/ MTD 0-dim	85.7 ± 3.3	$\mathbf{81.7 \pm 2.7}$
- w/ MTD 0,1-dim	$\mathbf{86.4 \pm 1.8}$	81.2 ± 3.7
Qwen2.5-Coder-7B		
Diag. Feat.	80.0 ± 2.8	76.5 ± 4.1
- w/ MTD 0-dim	80.6 ± 4.0	79.4 ± 3.3
- w/ MTD 0,1-dim	$\mathbf{81.6 \pm 4.7}$	$\mathbf{80.9 \pm 3.5}$
Magicoder-S-DS-6.7B		
Diag. Feat.	79.4 ± 2.9	80.4 ± 1.0
- w/ MTD 0-dim	$\mathbf{80.5 \pm 2.9}$	79.5 ± 2.7
- w/ MTD 0,1-dim	79.3 ± 3.5	$\mathbf{80.7 \pm 2.5}$

Table 22: Attention feature ablation in CodeTD.

U Code Samples

```
Generation 1, Problem #53, Correct

def add(x: int, y: int):
    """Add two numbers x and y
    >>> add(2, 3)
    5
    >>> add(5, 7)
    12
    """
    return x + y

Generation 2, Problem #23, Correct

def strlen(string: str) -> int:
    """ Return length of given string
    >>> strlen('')
    0
    >>> strlen('abc')
    3
    """
    return len(string)

Generation 3, Problem #41, Not correct - hallucination

def car_race_collision(n: int):
    """
    Imagine a road that's a perfectly straight infinitely long line.
    n cars are driving left to right;  simultaneously, a different set of n cars
    are driving right to left.   The two sets of cars start out being very far from
    each other.  All cars move in the same speed.  Two cars are said to collide
    when a car that's moving left to right hits a car that's moving right to left.
   However, the cars are infinitely sturdy and strong; as a result, they continue moving
    in their trajectory as if they did not collide.

    This function outputs the number of such collisions.
    """
    pass

Generation 4, Problem #41, Not correct - hallucination

def car_race_collision(n: int):
    """
    Imagine a road that's a perfectly straight infinitely long line.
    n cars are driving left to right;  simultaneously, a different set of n cars
    are driving right to left.   The two sets of cars start out being very far from
    each other.  All cars move in the same speed.  Two cars are said to collide
    when a car that's moving left to right hits a car that's moving right to left.
   However, the cars are infinitely sturdy and strong; as a result, they continue moving
    in their trajectory as if they did not collide.
```

```
        This function outputs the number of such collisions.
        """
        # The cars are moving at the same speed. So, the distance between them is always n.
        # If we find the sum of the distances between two cars in the initial state
        # and at the end of the race, we can simply subtract them to get the number
        # of collisions.

        total_distance_before = 0
        total_distance_after = 0

        for car_distance in range(1, n + 1):
            total_distance_before += car_distance
            total_distance_after += n - car_distance + 1

        return total_distance_before - total_distance_after
```