Topology of Attention Detects Hallucinations in Code LLMs

Anonymous ACL submission

Abstract

While 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 propose a new approach to assessment of code correctness. Our solution is based on topological data analysis (TDA) of attention maps of code LLMs. We carry out experiments with common benchmarks (HumanEval, MBPP, 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, the trained classifiers are transferable between coding benchmarks.

1 Introduction

Large Language Models (LLMs) are now widespread and have great potential to transform natural language processing and artificial intelligence. As far as code generation is concerned, LLMs that are trained on large amounts of code are capable of generating human-level code for a plethora of simple problems and are expected to revolutionize software engineering. At the same time, code-generating LLMs are 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. While many definitions of hallucinations exist, in this paper we assume that code hallucination is a code which is not functionally correct, that is, does not pass functional tests. For a wide adoption of code LLMs, there is a high need for automatic assessment of code quality. 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 quality can be inferred before its execution from an internal state of LLM, in particular its attention maps. 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).

Attention maps of LLMs are shown to capture semantically meaningful information and might be an illustration of the model's "thinking process". The research community actively studies approaches to mitigate hallucinations of LLMs by external knowledge bases (Peng et al., 2023) or to reduce them to some extent (Elaraby et al., 2023). It is highly desirable to evaluate the quality of the code before its execution and a run of tests since the code might contain security vulnerabilities.

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.

Our contributions are the following:

- We propose a new approach to detection of hallucinations in LLM-generated code based on analyzing a topology of attention maps;
- We carry out computational experiments with 10 code LLMs of size up to 34B parameters, two benchmarks (HumanEval, MBPP), 5 pro-

gramming languages, and show that the proposed method outperforms baselines;
- We empirically show that the proposed hallucination classifier is transferable between code benchmarks and code LLMs;
- 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) studied BERT's attention patterns: attending to delimiter tokens, specific positional offsets, broadly attending over the whole sentence, and showed that certain attention heads correspond well to the linguistic notions of syntax and coreference. Htut et al. (2019) found that some self-attention heads act as a proxy for syntactic structure, recovering the dependency type on parsed English text, for some universal dependency tree relation types. Michel et al. (2019) showed that for downstream tasks, a large proportion of attention heads can be removed at test time without affecting performance.

The phenomenon of code hallucinations is studied and categorized in several papers. Tian et al. (2024) 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. (2024) 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 LMM 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 introduced methods for preventing and detecting hallucinations. Peng et al. (2023) proposed to mitigate hallucination with an LLM-AUGMENTER, a system that allows the LLM to generate responses grounded in external knowledge, for example, stored in task-specific databases. Zhang et al. (2024b) proposed Self-Eval, a self-evaluation component, to prompt an LLM to validate the factuality of its own generated responses solely based on its internal knowledge. Feng et al. (2024) proposed two novel approaches for hallucination detection that are based on model collaboration, i.e., LLMs that investigate other LLMs for knowledge gaps, cooperatively or competitively. Zhang et al. (2024a) proposed to improve the truthfulness of LLMs by editing their internal representation during inference in the "truthful" space. Yehuda et al. (2024) introduced InterrogateLLM, a method that prompts the model multiple times to reconstruct the input query using the generated answer. Subsequently, InterrogateLLM quantifies the level of inconsistency between the original query and the reconstructed queries.

3 Background

3.1 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 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.

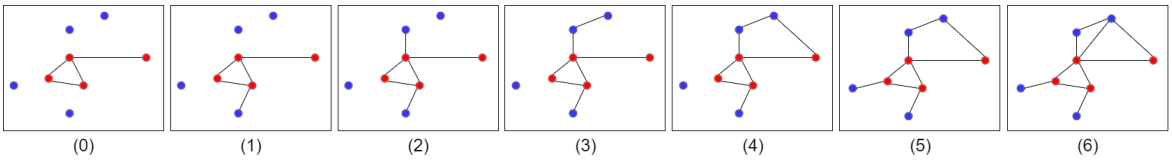


Figure 1: An example of MTD evaluation for a graph having two groups of vertices – red and blue. (0): initially, only edges connecting red vertices are present. (1)-(6): the rest of edges are added sequentially in an ascending order by their weights. While adding edges, connected components merge with each other. These moments are depicted by H_0 bars in Fig. 2. At moment (4) a cycle appears, at moment (6) this cycle disappears. These moments are depicted by the H_1 bar in Fig. 2.

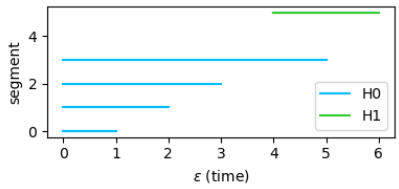


Figure 2: Cross-Barcode for a filtration from Fig. 1.

3.2 Attention Map as a Weighted Graph

While 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). The natural idea is to leave only the most interacting tokens, that is, attending to each other higher than some threshold. However, the optimal threshold is not known in advance. Moreover, the topology of such a graph changes discontinuously with the change of a threshold (or weights). Topological Data Analysis (TDA) (Chazal and Michel, 2017) introduces a principled way to access the topology of such graphs for all thresholds simultaneously.

3.3 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 1. 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 2. 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. The higher MTD$_k$ is, the greater is the "dissimilarity" between sets of tokens. Note that according to a definition, MTD$_k$ is not symmetric. Also, MTD$_k$, as a kind of persistence barcode, enjoys stability w.r.t. small perturbations of weights (Cohen-Steiner et al., 2005).

4 Methods

In the context of code generation, we naturally have two sets of tokens – a prompt and a generation. A common cause of hallucination is when the model's attention drifts away from the prompt[1]. Our topology-based features quantify this mismatch, yet the same signal helps to identify other error patterns as shown below (Section 5.6). As was pointed out in Section 3.2, attention matrices can be analyzed as weighted graphs. Specifically, for n tokens in a sequence, we consider a fully-connected undirected weighted graph with n vertices, where weights of edges are obtained from an attention map: $w_{i,j} = 1 - a_{i,j}$, for $i > j$ (we use decoder-only LLMs with causal attention). Then, Cross-Barcode and MTD for a weighted "atten-

[1]The definition of hallucination is discussed in the Appendix H.

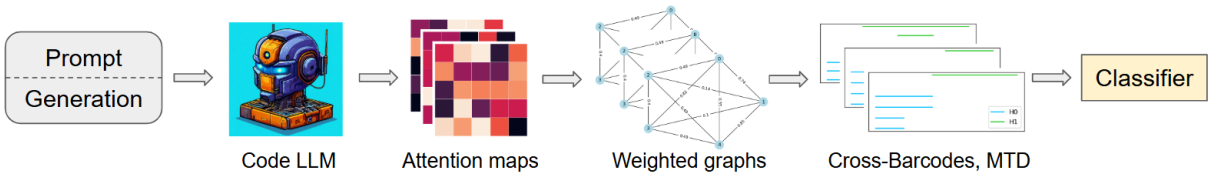


Figure 3: A pipeline of the proposed method for hallucination detection: (1) a prompt concatenated with a generated code is fed into a Code LLM. (2) Attention maps from the Code LLM are obtained. (3) Attention maps are transformed into fully-connected weighted graphs. (4) Cross-Barcodes and MTD features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

tion graph" can be calculated[2]. To predict code hallucinations, we use the following set of features:

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

Here, all features are normalized by the size of the set of corresponding vertices for better transferability. In addition, averages of diagonal values of the attention matrices that are not directly present in edge weights are included. These features are calculated for every layer and head of a code LLM. At the top of the proposed topological features, we applied XGBoost (Chen and Guestrin, 2016) for a classification[3]. The high-level pipeline of the proposed method is shown in Figure 3.

MTD, when applied to attention graphs, measures the strength of connectivity between tokens of the generation and tokens of the prompt. Low values of MTD mean the high connectivity 1) between tokens of the generation and the prompt 2) between tokens inside the generation. MTD scores of some heads have a significant discriminative power and are shown in Fig. 4. Lack of generation-prompt attention means that representations of tokens from generation does not depend on prompt. That is, LLM drifts away from the prompt during generation and hallucinates.

5 Experiments

5.1 Generation Procedure

To assess the efficacy of the proposed method for hallucination prediction, we carried out a set of computational experiments. 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), MultiPL-E (Cassano et al., 2023) [4]. In order to account for various possible code generations, for each of the coding problems, several solutions were generated by each of the selected code LLMs unless otherwise specified: for the small-sized and medium-sized models, we obtained 25 generations per task for HumanEval, 5 generations per task for MBPP and 10 generations per task for MultiPL-E; for larger models, we used 10 generations per task for HumanEval. To address the quality of the proposed approach in different LLM prompting regimes, we used a 0-shot prompt for the HumanEval dataset and a 1-shot/2-shot prompt for the MBPP dataset. To enable diversity of the generated solutions, a temperature sampling was done (see Appendix K for evaluation with greedy decoding). See Appendix A for further details on generation procedure. Tables 15, 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 6, 7 in Appendix A). Incorrect code is considered a "hallucination"; prediction of code's correctness is a binary classification problem. The pass@1 metric is slightly lower than reported in original papers, mostly because we have used temperature sampling

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

[3] Appendix E contains an ablation study for the choice of classifier.

[4] Licenses of pretrained models and benchmarks permit use for research purposes.

instead of greedy decoding. Before moving further, note that there is a strong negative dependency between prompt and generation lengths and code quality, see Figures 8, 9. The longer the prompt (i.e. task description) and generation (i.e. task solution) are, the lower is the probability of code's correctness. This dependency is more pronounced for HumanEval than for MBPP, because MBPP employed 1-shot prompts. These attributes are natural baselines for hallucination's prediction.

5.2 Code Hallucination Detection

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.

We used the following baselines for comparison:

- XGBoost classifier trained on tokenized prompt length, tokenized generation length;
- mean log. probability of generated tokens (Chen et al., 2021);
- a linear classifier on top of a frozen CodeT5-base encoder (Wang et al., 2021);
- Pylint[5], a static code analysis tool for Python;
- Self-Eval (Zhang et al., 2024b);
- Interrogate-LLM (Yehuda et al., 2024);
- CodeJudge (Tong and Zhang, 2024).

Finally, we utilize a combination of all features, i.e. tokenized prompt length, tokenized generation length, mean log. probability, and the proposed attention features, to train a classifier (we refer to it 'All Feat.' for brevity). Training details are provided in Appendix B.

Table 1 presents our main results. In the majority of cases, the proposed 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. The use of additional features can both decrease and increase overall performance. Thus, we believe that the proposed 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 4. In additional comparison, the proposed approach con-

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

Method	HE	MBPP
StarCoder2-7B		
Prompt Len.	55.2 ± 5.9	51.2 ± 2.3
Gen. Len.	57.8 ± 5.7	57.7 ± 0.9
Mean Log. Prob.	70.9 ± 1.3	62.1 ± 2.0
Pylint	58.9 ± 1.0	53.0 ± 0.6
CodeT5-base ft.	70.1 ± 7.1	58.5 ± 3.5
Self-Eval	50.0 ± 2.9	58.9 ± 2.6
Interrogate-LLM	63.9 ± 2.1	58.6 ± 2.6
Attn. Feat. (ours)	$\underline{82.8 \pm 3.4}$	$\mathbf{81.5 \pm 2.8}$
All Feat.	$\mathbf{83.8 \pm 3.2}$	$\underline{80.7 \pm 2.4}$
CodeLlama-7B		
Prompt Len.	61.6 ± 4.4	59.1 ± 4.2
Gen. Len.	60.1 ± 5.3	60.8 ± 2.5
Mean Log. Prob.	64.1 ± 2.0	61.0 ± 3.7
Pylint	55.1 ± 0.3	53.1 ± 0.5
CodeT5-base ft.	74.5 ± 6.3	61.7 ± 3.0
Self-Eval	49.7 ± 1.7	50.0 ± 0.0
Interrogate-LLM	67.9 ± 2.5	57.6 ± 2.4
Attn. Feat. (ours)	$\underline{85.6 \pm 3.9}$	$\underline{83.4 \pm 3.3}$
All Feat.	$\mathbf{85.7 \pm 3.8}$	$\mathbf{83.8 \pm 2.0}$
DeepSeek-Coder-6.7B		
Prompt Len.	56.3 ± 4.6	52.5 ± 2.5
Gen. Len.	57.9 ± 2.4	54.6 ± 1.9
Mean Log. Prob.	69.9 ± 2.6	61.0 ± 1.9
Pylint	54.6 ± 0.8	52.8 ± 0.4
CodeT5-base ft.	69.1 ± 4.2	55.7 ± 3.0
Self-Eval	56.7 ± 2.5	51.1 ± 0.6
Interrogate-LLM	71.8 ± 2.4	60.1 ± 5.1
Attn. Feat. (ours)	$\mathbf{86.0 \pm 3.1}$	$\underline{82.6 \pm 1.9}$
All Feat.	$\underline{85.7 \pm 2.6}$	$\mathbf{82.9 \pm 0.9}$
Qwen2.5-Coder-7B		
Prompt Len.	54.3 ± 8.7	51.8 ± 3.6
Gen. Len.	57.6 ± 3.6	55.6 ± 2.1
Mean Log. Prob.	63.1 ± 2.4	61.5 ± 1.3
Pylint	64.1 ± 2.2	62.0 ± 2.5
CodeT5-base ft.	65.9 ± 3.7	56.0 ± 1.3
Self-Eval	73.8 ± 1.6	66.6 ± 2.3
Interrogate-LLM	60.1 ± 3.7	55.0 ± 1.8
Attn. Feat. (ours)	$\underline{81.9 \pm 2.8}$	$\mathbf{82.3 \pm 1.6}$
All Feat.	$\mathbf{82.2 \pm 2.5}$	$\underline{81.9 \pm 2.7}$
Magicoder-S-DS-6.7B		
Prompt Len.	57.3 ± 5.4	54.0 ± 2.6
Gen. Len.	52.5 ± 2.1	52.3 ± 2.1
Mean Log. Prob.	70.9 ± 5.3	60.5 ± 3.7
Pylint	52.7 ± 1.3	52.0 ± 0.8
CodeT5-base ft.	64.7 ± 2.7	61.0 ± 3.7
Self-Eval	44.3 ± 3.2	49.2 ± 1.4
Interrogate-LLM	65.3 ± 2.1	59.0 ± 7.1
Attn. Feat. (ours)	$\mathbf{83.2 \pm 4.8}$	$\mathbf{81.7 \pm 1.7}$
All Feat.	$\underline{82.0 \pm 4.5}$	$\underline{80.8 \pm 2.2}$

Table 1: ROC-AUC of code hallucination detection for the HumanEval and MBPP datasets.

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

sistently outperforms CodeJudge (Tong and Zhang, 2024) both in greedy decoding and sampling setups (see Appendix I for details). The proposed method can also outperform or improve the concurrent approaches for larger LLMs, see Table 11 and Appendix I.

Furthermore, we evaluated the proposed approach on Java (high resource), Go (medium resource), Rust (low resource), and Lua (niche) programming languages of the HumanEval subdivision of the MultiPL-E dataset (Cassano et al., 2023). The proposed Attn. Feat. classifier can detect hallucinations even in low resource and niche languages (see Table 3), and there are several separate features that perform robust across the languages, see Tables 14 in Appendix. Finally, the proposed Attn. Feat. classifier has consistent quality when the linguistic complexity of the prompt is increased, see Table 13.

Model	Java	Go
StarCoder2-7B	82.7 ± 4.9	86.5 ± 3.6
CodeLlama-7B	76.8 ± 6.1	81.9 ± 6.2
DeepSeek-Coder-6.7B	84.9 ± 4.2	84.7 ± 2.9
Qwen2.5-Coder-7B	82.6 ± 4.1	91.8 ± 0.9
Magicoder-S-DS-6.7B	77.8 ± 4.5	80.8 ± 2.0
	Rust	**Lua**
StarCoder2-7B	82.4 ± 5.2	82.0 ± 3.5
CodeLlama-7B	77.5 ± 10.8	–
DeepSeek-Coder-6.7B	82.8 ± 7.1	86.7 ± 4.3
Qwen2.5-Coder-7B	87.3 ± 2.6	90.2 ± 3.0
Magicoder-S-DS-6.7B	75.1 ± 5.9	79.0 ± 3.2

Table 3: ROC-AUC of the proposed method on different programming languages from MultiPL-E dataset.

Method	HE → MBPP	MBPP → HE
StarCoder2-7B		
Mean Log. Prob.	63.8	**71.8**
CodeT5-base ft.	53.7	59.1
Attn. Feat.	**67.7**	66.0
CodeLlama-7B		
Mean Log. Prob.	57.7	65.0
CodeT5-base ft.	54.9	62.4
Attn. Feat.	**69.5**	**80.3**
DeepSeek-Coder-6.7B		
Mean Log. Prob.	62.7	69.1
CodeT5-base ft.	53.4	55.9
Attn. Feat.	**69.9**	**72.4**
Qwen2.5-Coder-7B		
Mean Log. Prob.	60.3	64.7
CodeT5-base ft.	49.1	51.6
Attn. Feat.	**70.9**	**65.8**
Magicoder-S-DS-6.7B		
Mean Log. Prob.	63.8	**69.8**
CodeT5-base ft.	49.3	45.9
Attn. Feat.	**73.5**	56.9

Table 4: Transferability of code hallucination detectors, ROC-AUC. Each classifier was trained on HumanEval (MBPP) dataset and tested on MBPP (HumanEval) dataset.

5.3 Ranking Ability

Next, we assess the usefulness of the proposed code hallucination classifier for ranking of code generations. For each problem, all generations were ranked according to the predicted probability of correctness and one with the highest probability was selected. A baseline was random selection of a code generation. The use of a classifier always leads to a significantly higher pass@1 score, see Table 2. This result justifies that the proposed Att. Feat. classifiers can successfully rank candidate solutions of the same problem and improve pass@1, even for more recent Qwen2.5-Coder-7B and Magicoder-S-DS-6.7B.

5.4 Transferability

Cross-benchmark transferability We further study the transferability of the classifiers based on topological features. 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 vise versa. Table 4 shows the results: the proposed classifiers are 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 → MBPP and MBPP → HE transfer. We relate the changes in the classifiers' performance compared to results from Section 5.2 to differences in the prompt structure: we use 0-shot prompt for HumanEval and 1-shot prompt for MBPP.

Cross-classifier transferability. We verify whether the proposed XGBoost classifier trained on the attention features of one model can be effectively applied to detect hallucinations using the attention features of another model. A formal requirement is that the number of attention features of the two models should be the same. The results of cross-model transferability are shown in Table 18. We conclude that a classifier trained on DeepSeek-Coder-6.7B is transferable to Magicoder-S-DS-6.7B and vice versa, which is not the case for CodeLlama-7B. We suppose the reason is that Magicoder-S-DS-6.7B is a fine-tuned DeepSeek-Coder-6.7B and a correspondence between attention heads persists. This observation suggests that the proposed approach can potentially be further used as a reward model in RL-based LLM finetuning.

Cross-model transferability. Second, 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 19. The best detection performance across all code-generating models is achieved with Magicoder-S-DS-6.7B, which outperformed a larger DeepSeek-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.

5.5 Feature Importance

In its base setup, the proposed approach requires computation of attention features from attention maps for all layers and heads. However, we observed that the trained XGBoost classifier experienced a natural sparsity with only about 25% meaningful features, as measured by the feature importance attributed by the classifier. To further explore the importance and selection of features, we followed the two-stage pipeline. First, for a given sparsity level, we selected the most important features as measured by the feature importance attributed by the classifier trained on all attention features simultaneously. Second, we trained a new XGBoost classifier on the selected set of attention features. As indicated by Figure 5, the proposed feature selection procedure could retain only 5% of all attention features without a significant loss of classification quality, highlighting that only a limited number of all attention heads are relevant for 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

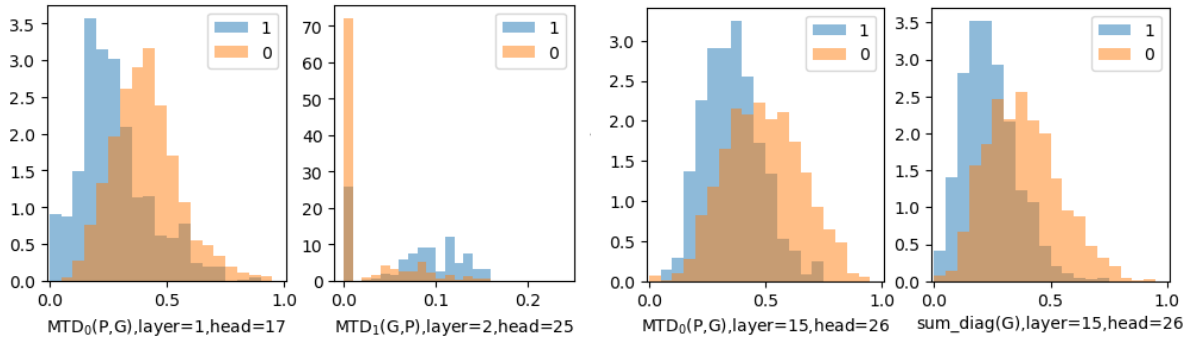
Table 5: Performance of multi-classification of error types.

We carry out additional experiments benchmarking different programming languages (Python, Go, Rust, Java, Lua) and find that features of some heads exhibit a high predictive performance consistently across all programming languages, see Appendix J.

5.6 A Fine-Grained Error Type Classification

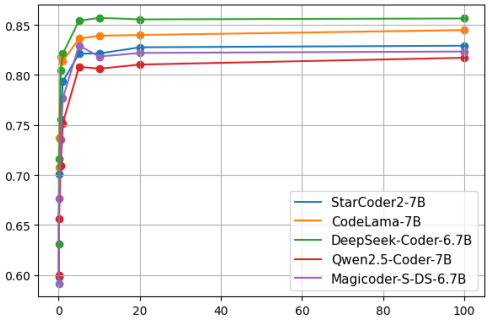
We carried out additional experiments to study the ability of the proposed approach to detect specific types of hallucinations. We consider an exception type in Python as a hallucination type. Here are the common exceptions from the HumanEval and MBPP benchmarks: *AssertionError, AttributeError, IndentationError, IndexError, ModuleNotFoundError, NameError, RecursionError, SyntaxError, TypeError, UnboundLocalError, ValueError, ZeroDivisionError, timed out*

Therefore, we did multi-classification instead of binary classification in XGBoost. Table 5 shows the results. This result demonstrates that the proposed attention features are capable of identifying the majority of errors of different types, consistently across the models. Here is a breakdown of 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.

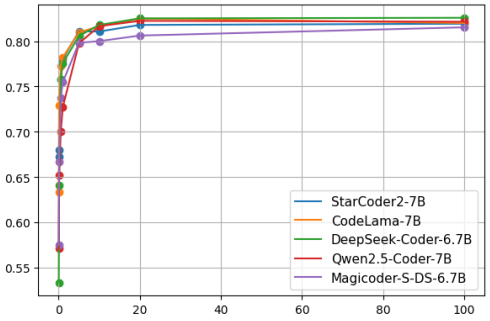


(a) HumanEval (b) MBPP

Figure 4: 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.



(a) HumanEval



(b) MBPP

Figure 5: Analysis of feature importance of the proposed method. ROC-AUC vs. percentage of retained features.

5.7 Ablation Study

The proposed approach is based on the three types of attention features: the diagonal elements of attention maps corresponding to a prompt and a generation, and topological (MTD) features of 0 and 1 dimension computed for the corresponding "attention graph" (see Section 4 for details). In this section, we provide an ablation study to estimate the contribution of each type of attention features. For this purpose, we trained the XGBoost classifier starting with diagonal attention values and gradually incorporating 0-dim and 1-dim MTD features. This order corresponds to an increase in computational complexity. As demonstrated in Table 7 in Appendix, the usage of topological features consistently improves the quality of hallucination detection over diagonal features. Topological features typically lead to more robust classifiers with better transferability across datasets, see Table 8 in Appendix. In order to account for various information available via attention maps, we propose using all types of features as the most universal choice. Nevertheless, in certain cases, a particular type of attention features may perform better than combination of all features.

6 Conclusion

In this paper, we propose a new hallucination detection approach for code LLMs. Our approach is based on the introspection of an LLM: we get attention maps for a prompt and generation and study their topology after transforming to weighed graphs. The proposed topological features of these graphs have been empirically shown to be relevant for the detection of code hallucinations. A classifier built on top of these features outperformed several baselines. These classifiers are transferable across the coding benchmarks and models. The natural extension of our research is the detection of specific places of code with bugs, and we leave it for further research. We believe that our work may lead to a wider application of code LLMs by making them more reliable. 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.

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.

Mohamed Elaraby, Mengyin Lu, Jacob Dunn, Xueying Zhang, Yu Wang, Shizhu Liu, Pingchuan Tian, Yuping Wang, and Yuxuan Wang. 2023. Halo: Estimation and reduction of hallucinations in open-source weak large language models. *arXiv preprint arXiv:2308.11764*.

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, Qian Chen, Wen Wang, Ziyang Luo, and Lei Ma. 2024. Codehalu: Code hallucinations in llms driven by execution-based verification. *arXiv preprint arXiv:2405.00253*.

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.

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

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*.

A Details on Generation Procedure

We generate solutions for the coding problems with a temperature of 0.8. For the HumanEval dataset, 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 6, 7 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. To compute the attention features according to the method proposed in Section 4, we used the attention submatrix corresponding to the input prompt and the valid solution to the problem. For computational experiments, we used NVIDIA TITAN RTX.

B Details on Training Procedure

For the code hallucination detectors, based on the XGBoost classifier training, we utilized the XGBClassifier with an approximation tree method "hist" from the XGBoost library[7]. For the code hallucination detector based on CodeT5-base embeddings,

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

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

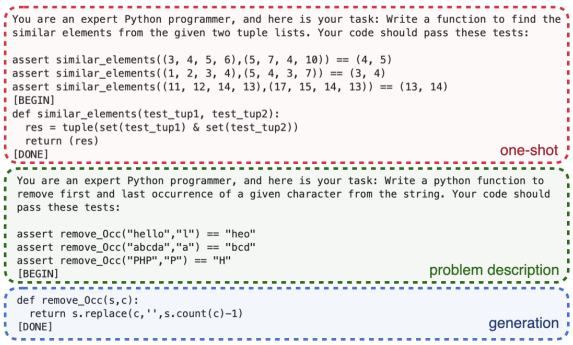


Figure 7: Example of prompt (one-shot example and problem description) and model generation for the MBPP dataset.

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.

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.

D Examples of Cross-Barcodes

For the MTD feature which strongly distinguishes distribution of classes (Figure 4, (a), left), we show Cross-Barcodes having high and small values of this feature, see Figure 10. See corresponding code samples in Appendix M. 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 generrated code.

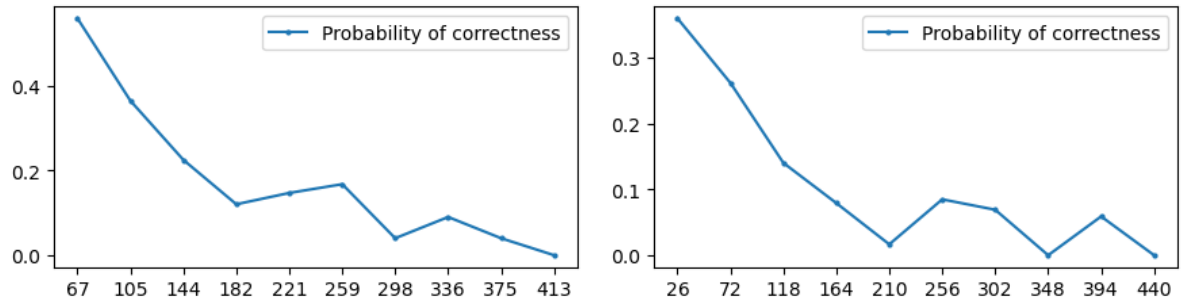
Figure 11 shows examples of Cross-Barcode$_0$ for a fixed attention head. The Cross-Barcode$_1$(P, G) is empty for these attention maps. Correct generations (a), (b) tend to have more and more H_0 bars than not-correct ones (c), (d).

Figure 12 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 13, 14.

E Ablation for the Classifier Model Selection

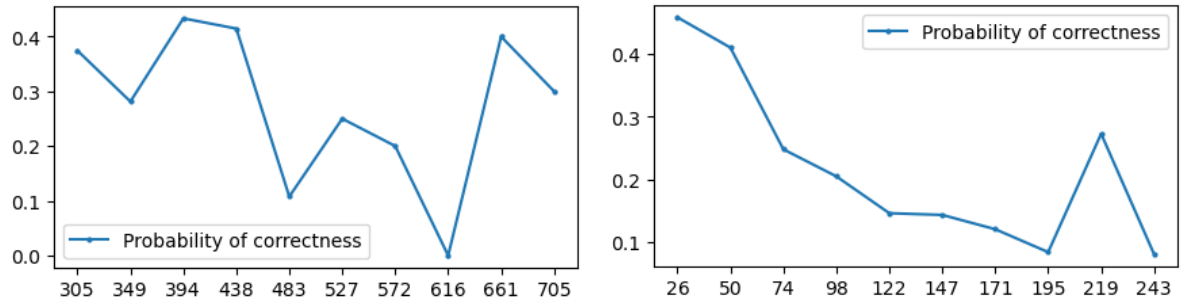
We carried out additional experiments with the feed-forward network (MLP), logistic regression,



(a) Prompt length, tokens. HumanEval.

(b) Generation length, tokens. HumanEval.

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



(a) Prompt length, tokens. MBPP.

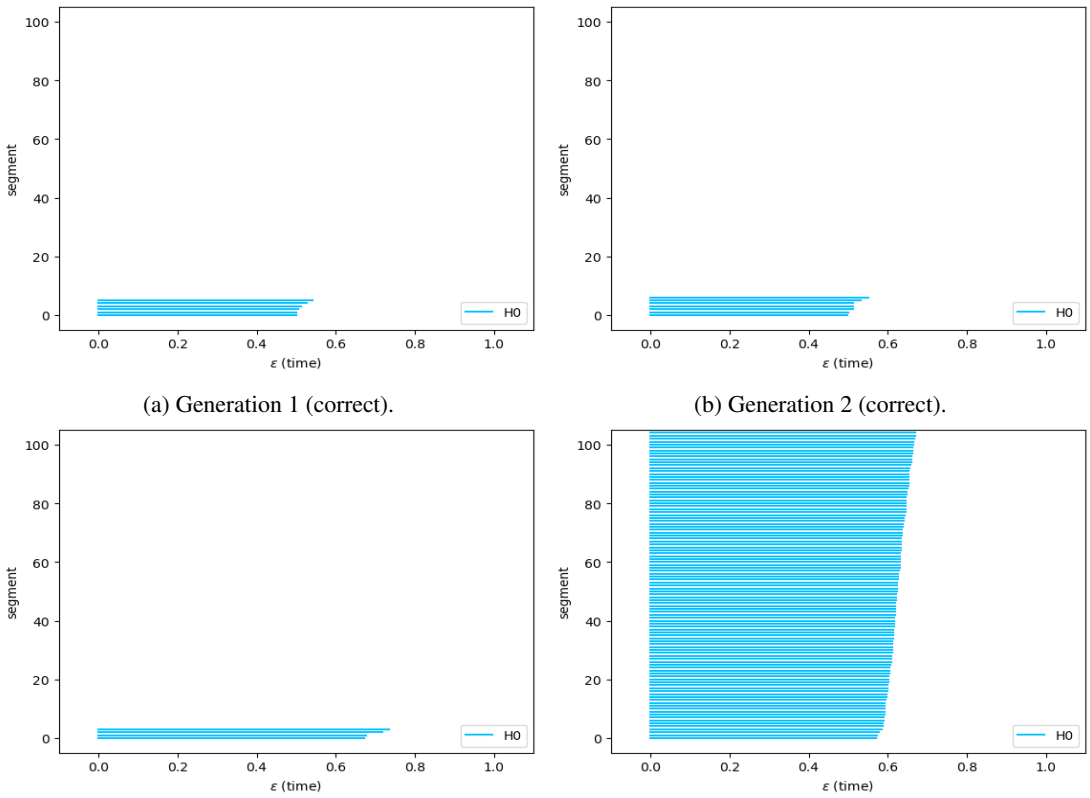
(b) Generation length, tokens. MBPP.

Figure 9: The individual conditional expectations for prompt and generation lengths, CodeLlama-7B.

and support vector classifier (SVC) instead of XGBoost as a classifier for hallucination detection (the rightmost block in Figure 3). 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 6 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.

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 $(\text{seq\_len\_k})^2$ 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 $\times \sum_{i=1}^{k} (\text{seq\_len\_k})^2$ 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.



(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, 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).

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 feature extraction is 6.2 sec, with feature extraction 11.5 sec 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 5.5, it is possible to use only 5% of all attention features without significant loss of classification quality The average memory footprint is ≈190MB for HumanEval and ≈544MB for MBPP. This size is a small fraction of GPU footprint during generation.

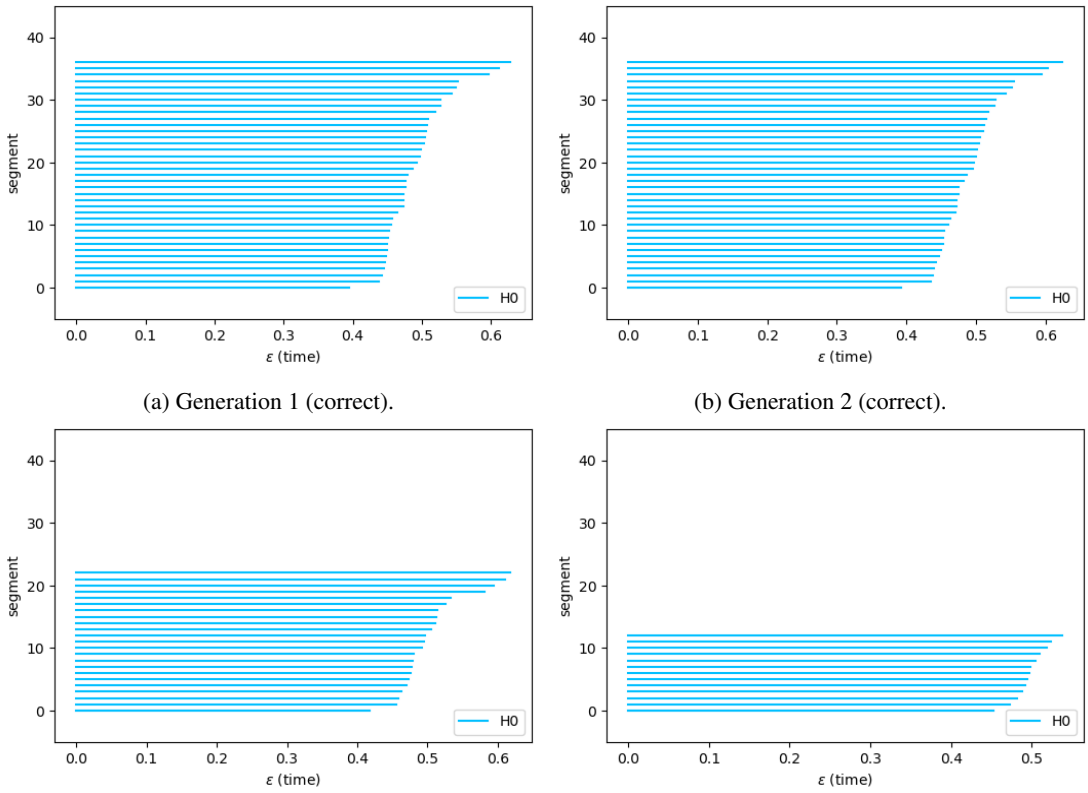
H On Definition of a Code Hallucination

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 correspond. 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, that is, whether a code solves the corresponding problem as stated in a prompt. Verification of a code is done by running functional tests, as explained in Section 5.1.

I Experiments with Larger Models

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 following prompt:

You are provided with two coding tasks with so-



(a) Generation 1 (correct).

(b) Generation 2 (correct).

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

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

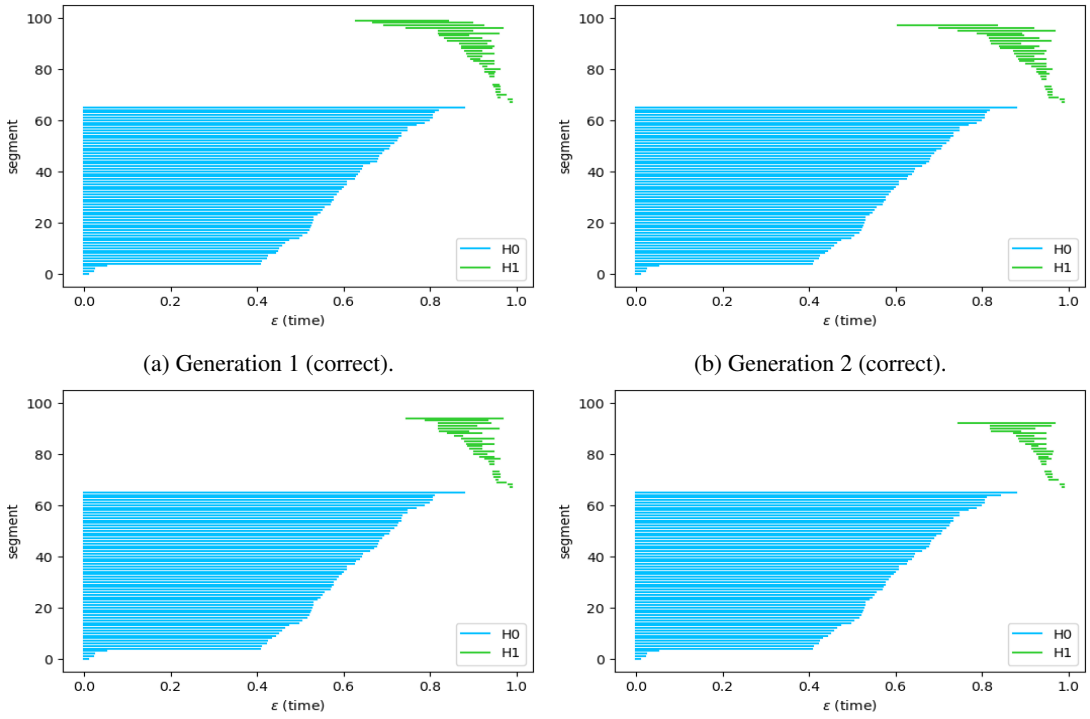
Figure 11: Examples of Cross-Barcode$_0$(P, G), CodeLLama-7B, HumanEval dataset, problem 14, layer 4, head 18. Cross-Barcode$_1$(P, G) is empty for these attention maps.

lutions. 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 9, row "DeepSeek R1, (671B model)". The quality of the proposed approach is shown in Table 9, row "Attn. Feat (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 9, row "Attn. Feat. (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 approach is capable of achieving a better detection quality than the zero-shot 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



(a) Generation 1 (correct).

(b) Generation 2 (correct).

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

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

Figure 12: Examples of Cross-Barcode$_0$(G, P), Cross-Barcode$_1$(G, P). CodeLLama-7B, HumanEval dataset, problem 14, layer 15, head 5.

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 Attn. Feat. 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 the Attn. Feat. in a similar way to keep the experimental design of both methods as close as possible. In this comparison, Attn. Feat. outperforms CodeJudge, winning by a large margin for ROC-AUC. However, to further explore the capabilities of the proposed Attn. Feat., we provide additional comparison when CodeJudge is run with reference and when Attn. Feat. is run for smaller 6.7-7B models. In these cases, Attn. Feat. still outperforms CodeJudge as measured by ROC-AUC.

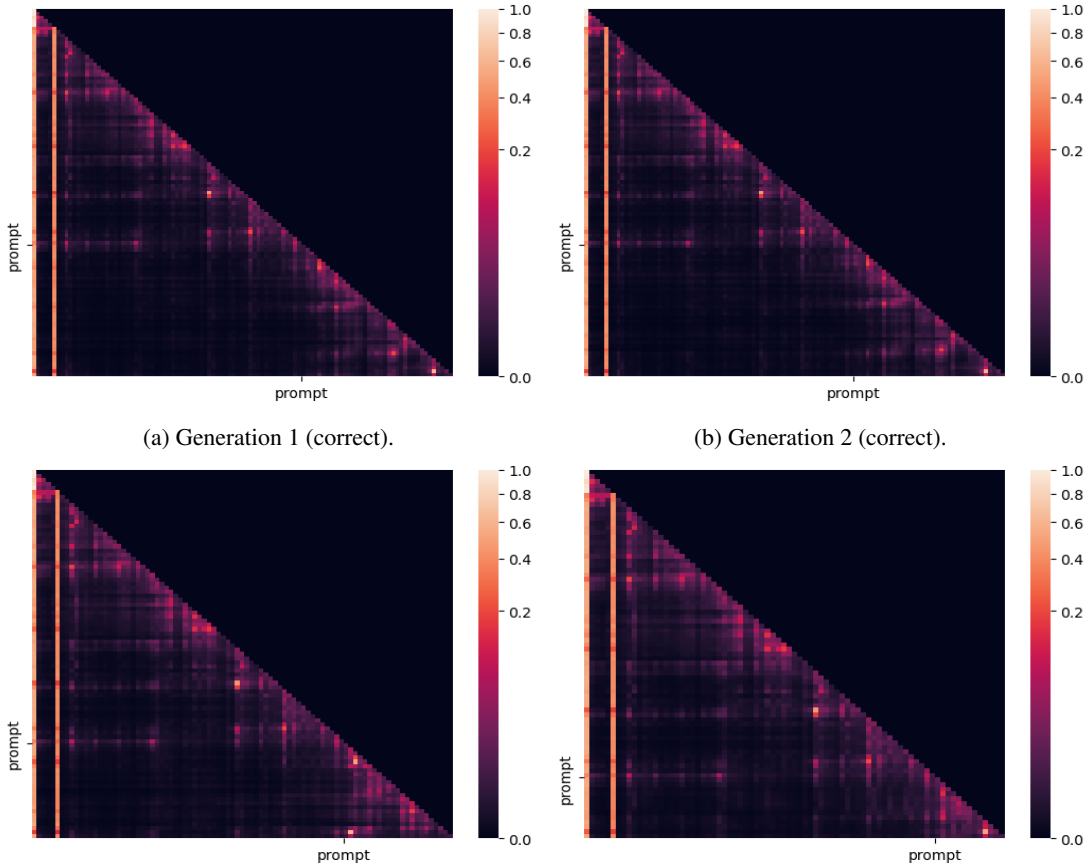
J Contribution of Individual Heads

We carry out additional experiments with 7B-sized models and MultiPL-E benchmark[8], 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 14 we report the ROC-AUC scores of the top-performing features.

K Evaluation with Greedy Decoding

In the main experiments in Section 5.2, 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 classifier when both train and test data are generated with greedy decoding. In this setup, the sample sizes are 164 for 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 5.2,

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



(a) Generation 1 (correct).

(b) Generation 2 (correct).

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

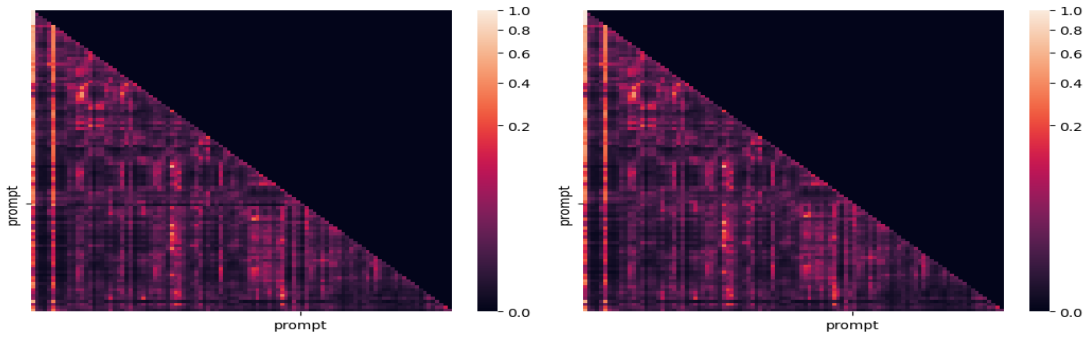
(d) Generation 4 (not correct, hallucination).

Figure 13: Attention maps. CodeLLama-7B, HumanEval dataset, problem 14, layer 4, head 18.

the proposed classifier achieves high performance across all models and datasets and can be applied to small samples; see Table 17. The proposed approach 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.

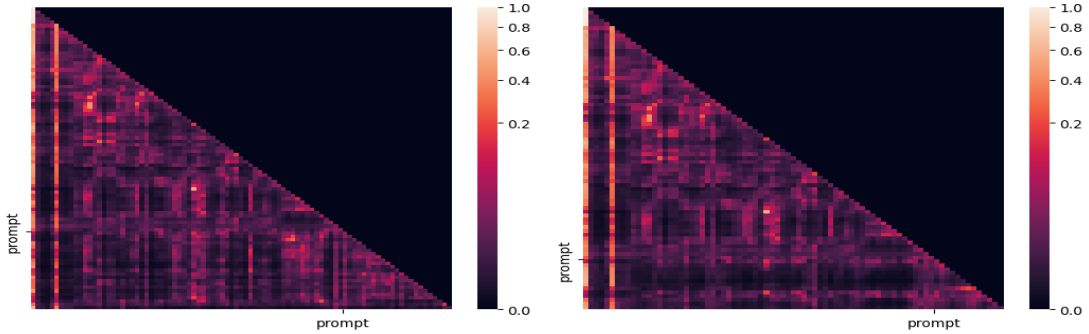
L Failure Case Study

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.



(a) Generation 1 (correct).

(b) Generation 2 (correct).



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

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

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

Classifier	HE	MBPP
StarCoder2-7B		
XGBoost	82.8 ± 3.4	81.5 ± 2.8
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	86.0 ± 3.1	$\mathbf{82.6 \pm 1.9}$
MLP	84.3 ± 2.4	81.7 ± 1.1
Logistic Regression	85.8 ± 3.2	81.8 ± 2.4
SVC	$\mathbf{87.0 \pm 2.4}$	81.4 ± 1.6
Qwen2.5-Coder-7B		
XGBoost	$\mathbf{81.9 \pm 2.8}$	$\mathbf{82.3 \pm 1.6}$
MLP	81.3 ± 1.6	79.5 ± 2.0
Logistic Regression	80.0 ± 1.6	81.0 ± 1.9
SVC	79.6 ± 1.7	81.0 ± 1.7
Magicoder-S-DS-6.7B		
XGBoost	$\mathbf{83.2 \pm 4.8}$	81.7 ± 1.7
MLP	76.5 ± 3.3	78.6 ± 3.6
Logistic Regression	81.7 ± 1.6	81.3 ± 2.8
SVC	80.5 ± 2.4	$\mathbf{82.5 \pm 2.6}$

Table 6: Ablation study for the choice of a classification model in code hallucination detection pipeline.

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.	$\mathbf{84.2 \pm 2.7}$	81.2 ± 3.1
- w/ MTD 0-dim	83.4 ± 3.3	$\mathbf{81.8 \pm 2.3}$
- w/ MTD 0,1-dim	82.8 ± 3.4	81.5 ± 2.8
DeepSeek-Coder-6.7B		
Diag. Feat.	84.8 ± 2.2	82.2 ± 1.7
- w/ MTD 0-dim	85.2 ± 2.6	82.0 ± 2.0
- w/ MTD 0,1-dim	$\mathbf{86.0 \pm 3.1}$	$\mathbf{82.6 \pm 1.9}$
Qwen2.5-Coder-7B		
Diag. Feat.	78.8 ± 2.0	76.9 ± 2.2
- w/ MTD 0-dim	$\mathbf{82.4 \pm 1.5}$	80.6 ± 2.1
- w/ MTD 0,1-dim	81.9 ± 2.8	$\mathbf{82.3 \pm 1.6}$
Magicoder-S-DS-6.7B		
Diag. Feat.	81.4 ± 2.7	80.3 ± 2.5
- w/ MTD 0-dim	82.5 ± 3.8	80.7 ± 1.5
- w/ MTD 0,1-dim	$\mathbf{83.2 \pm 4.8}$	$\mathbf{81.7 \pm 1.7}$

Table 7: Attention feature ablation.

Method	HE $\rightarrow$ MBPP	MBPP $\rightarrow$ 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 8: Dataset transferability attention feature ablation.

Method	MBPP
Attn. Feat (from DeepSeek-Coder-6.7B)	82.6 ± 1.9
DeepSeek R1 (671B model)	92.3 ± 1.1
Attn. Feat. (from DeepSeek-Coder-6.7B) + DeepSeek R1 (671B)	$\mathbf{95.1 \pm 0.7}$

Table 9: MBPP features for larger models. See Section I for details.

Method	HE
QwenCoder2.5-32B	
Attn. Feat (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	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
Attn. Feat. (ours)	$\mathbf{84.1 \pm 3.2}$
All. Feat.	$\underline{83.5 \pm 2.4}$
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
Attn. Feat. (ours)	$\underline{85.0 \pm 2.7}$
All. Feat.	$\mathbf{85.3 \pm 2.7}$
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
Attn. Feat. (ours)	$\underline{88.2 \pm 3.0}$
All. Feat.	$\mathbf{88.5 \pm 2.2}$

Table 11: Code hallucination detection for HumanEval dataset for larger models.

Method	Greedy Dec.	T=0.8
StarCoder2-7B		
CodeJudge w/o ref.	70.6 ± 4.7	69.9 ± 1.4
CodeJudge w/ ref.	$\underline{75.8 \pm 4.7}$	$\underline{71.3 \pm 1.7}$
Attn. Feat.	$\mathbf{80.8 \pm 3.7}$	$\mathbf{82.8 \pm 3.4}$
CodeLlama-7B		
CodeJudge w/o ref.	71.3 ± 4.1	$\underline{72.1 \pm 1.3}$
CodeJudge w/ ref.	$\underline{73.0 \pm 3.3}$	71.9 ± 3.6
Attn. Feat.	$\mathbf{80.1 \pm 3.9}$	$\mathbf{85.6 \pm 3.9}$
DeepSeekCoder-6.7b		
CodeJudge w/o ref.	$\underline{69.2 \pm 3.2}$	$\underline{68.4 \pm 1.8}$
CodeJudge w/ ref.	68.8 ± 7.9	68.2 ± 2.7
Attn. Feat.	$\mathbf{83.9 \pm 8.4}$	$\mathbf{86.0 \pm 3.1}$
QwenCoder-7b		
CodeJudge w/o ref.	67.2 ± 5.7	69.3 ± 1.7
CodeJudge w/ ref.	$\underline{70.4 \pm 7.4}$	$\underline{69.4 \pm 2.9}$
Attn. Feat.	$\mathbf{73.3 \pm 7.1}$	$\mathbf{81.9 \pm 2.8}$
Magicoder-S-DS-6.7b		
CodeJudge w/o ref.	55.3 ± 2.7	60.3 ± 2.8
CodeJudge w/ ref.	$\underline{57.9 \pm 7.1}$	$\underline{61.3 \pm 5.2}$
Attn. Feat.	$\mathbf{65.1 \pm 3.4}$	$\mathbf{83.2 \pm 4.8}$
CodeLlama-34B		
CodeJudge w/o ref.	65.3 ± 3.0	70.7 ± 5.1
CodeJudge w/ ref.	$\underline{68.7 \pm 3.8}$	$\underline{73.2 \pm 4.4}$
Attn. Feat.	$\mathbf{75.0 \pm 5.1}$	$\mathbf{84.1 \pm 3.2}$
DeepSeek-Coder-33B		
CodeJudge w/o ref.	72.0 ± 7.2	69.0 ± 2.8
CodeJudge w/ ref.	$\underline{75.8 \pm 5.6}$	$\underline{70.9 \pm 1.3}$
Attn. Feat.	$\mathbf{86.5 \pm 7.2}$	$\mathbf{88.2 \pm 3.0}$
Qwen2.5-Coder-32B		
CodeJudge w/o ref.	61.6 ± 7.3	66.5 ± 3.5
CodeJudge w/ ref.	$\underline{62.0 \pm 7.5}$	$\underline{68.2 \pm 3.7}$
Attn. Feat.	$\mathbf{74.0 \pm 6.4}$	$\mathbf{85.0 \pm 2.7}$

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
Qwen2.5-Coder-7B	
Prompt Len.	53.5 ± 2.9
Gen. Len.	56.9 ± 3.3
Mean Log. Prob.	58.1 ± 2.6
Attn. Feat. (ours)	$\mathbf{78.3 \pm 3.1}$
All. Feat.	$\underline{76.7 \pm 2.7}$
Self-Eval	68.4 ± 1.0
Interrogate-LLM	56.6 ± 1.5
DeepSeek-Coder-6.7B	
Prompt Len.	56.0 ± 2.0
Gen. Len.	54.4 ± 1.3
Mean Log. Prob.	64.9 ± 1.7
Attn. Feat. (ours)	$\mathbf{81.4 \pm 2.2}$
All. Feat.	$\underline{80.4 \pm 0.7}$
Self-Eval	50.0 ± 0.0
Interrogate-LLM	61.1 ± 3.1

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

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

Model	pass@1	#Pos.	#Neg.
HE			
StarCoder2-7B	28.9	1186	2914
CodeLlama-7B	25.9	1064	3036
DeepSeek-Coder-6.7B	40.3	1653	2447
Qwen2.5-Coder-7B	47.8	1961	2139
Magicoder-S-DS-6.7B	65.5	2689	1411
MBPP			
StarCoder2-7B	42.8	1071	1429
CodeLlama-7B	35.2	879	1621
DeepSeek-Coder-6.7B	52.6	1315	1185
Qwen2.5-Coder-7B	52.1	1302	1198
Magicoder-S-DS-6.7B	61.3	1533	967

Table 15: Characteristics of generated data: Pass@1, number of correct (#Pos.) and incorrect (#Neg.) solutions for each of the selected code LLMs.

Model	MultiPL-E			
	Java	Go	Rust	Lua
StarCoder2-7B	24.5	17.5	20.9	19.1
CodeLlama-7B	25.8	17.6	20.8	0.0
DeepSeek-Coder-6.7B	33.5	23.6	28.7	16.6
Qwen2.5-Coder-7B	22.7	11.9	22.6	23.5
Magicoder-S-DS-6.7B	48.9	40.3	44.2	34.6

Table 16: Characteristics of generations for MultiPL-E dataset, pass@1. For each problem, we generated 10 candidate solutions for Java, Go, Rust, Lua.

Model	HE	MBPP
StarCoder2-7B	80.8 ± 3.7	79.3 ± 6.1
CodeLlama-7B	80.1 ± 3.9	82.3 ± 3.1
DeepSeek-Coder-6.7B	83.9 ± 8.4	79.6 ± 3.4
Qwen2.5-Coder-7B	73.3 ± 7.1	74.0 ± 1.9
Magicoder-S-DS-6.7B	65.1 ± 3.4	79.3 ± 2.0

Table 17: Quality of code hallucination detection when greedy decoding is used to generate candidate solutions.

	CodeLlama-7B	**DeepSeek-Coder-6.7B**	**Magicoder-S-DS-6.7B**
CodeLlama-7B	$\mathbf{85.6 \pm 3.9}$	51.9 ± 13.2	51.1 ± 7.1
DeepSeek-Coder-6.7B	45.4 ± 10.3	$\mathbf{86.0 \pm 3.1}$	$\underline{79.4 \pm 3.7}$
Magicoder-S-DS-6.7B	$\underline{51.8 \pm 5.4}$	$\underline{83.0 \pm 2.2}$	$\mathbf{83.2 \pm 4.8}$

Table 18: Cross-classifier transferability: rows correspond to XGBoost train data, columns correspond to test data.

	StarCoder2-7B	**DeepSeek-Coder-6.7B**	**Qwen2.5-Coder-7B**	**Magicoder-S-DS-6.7B**	**DeepSeekCoder-33B**
StarCoder2-7B	82.8 ± 3.4	89.3 ± 2.2	86.6 ± 4.2	$\mathbf{92.0 \pm 2.4}$	$\underline{90.0 \pm 2.7}$
DeepSeek-Coder-6.7B	80.0 ± 3.6	86.0 ± 3.1	84.8 ± 3.2	$\mathbf{88.2 \pm 3.2}$	$\underline{87.5 \pm 3.2}$
Qwen2.5-Coder-7B	80.5 ± 3.2	85.6 ± 3.3	81.9 ± 2.8	$\mathbf{87.9 \pm 3.4}$	$\underline{86.6 \pm 2.9}$
Magicoder-S-DS-6.7B	73.4 ± 4.4	77.2 ± 3.6	80.2 ± 3.8	$\mathbf{83.2 \pm 4.8}$	$\underline{81.3 \pm 3.5}$

Table 19: Cross-model transferability: rows correspond to code-generating models, columns correspond to feature-extracting models.

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.

M 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
```