`pydra`: Probing Code Representations With Synthetic Clones and Bugs

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

We introduce `pydra`: an open-source dataset of ∼9k Python examples with synthetic clones and buggy variants for each. Our augmentation pipeline generates both semantics-preserving and bug-injecting code variants via AST transforms and stores rich metadata for analysis. Using `pydra`, we probe state-of-the-art code embedding models and find a stark limitation in their ability to rank correct variants above incorrect ones. Our analysis suggests that embeddings remain dominated by token overlap and code length rather than true program semantics. We hope that `pydra` serves the research community by filling several gaps in the Python code dataset ecosystem as well as providing a general tool for training and evaluating code embedding models.

1 Introduction

In the era of agentic AI, the use of Large Language Models (LLMs) for code has expanded beyond single-step code generation to complex workflows that involve analysis, repair, and maintenance on the scale of full codebases. Embedding models, lighter weight and optimized for semantic similarity, are frequently leveraged alongside generative models to handle tasks such as clone detection [1], code search [2], retrieval augmented generation [3], code ranking/reranking [4], and fault localization [5]. These tasks benefit from high quality embeddings that meaningfully represent the functional semantics of the code. Despite this, evaluation of code embedding quality remains under-explored.

Prominent code benchmarks (e.g. HumanEval [6], Mostly Basic Python Problems (MBPP) [7], BigCodeBench [8]) are geared towards generation and scored with pass@k [6], an execution-based metric that relies on high-coverage tests to assess the functional correctness of generated code. For tasks based on similarity, retrieval, or ranking, where fixed size vector embeddings are the relevant artifact rather than generated code, pass@k is is not directly applicable. Furthermore, execution-based approaches cannot generally evaluate partial generations or code fragments, further limiting their usefulness for such tasks.

Instead, code embedding models are usually judged by their performance on downstream tasks. In particular, code clone detection is often treated as an indicator for code understanding. However, it is unclear to what extent performance on existing clone benchmarks is based on semantic equivalence versus surface-level syntactic or lexical similarities. For example, [9] found that many clones in BigCloneBench [10] shared the same identifier names, leading to misleading performance metrics on this benchmark. Other tasks similarly entangle semantic and syntactic features. For example, [11] investigates fault localization as a proxy for code understanding, and notes that performance degrades on the same bugs if semantics-preserving augmentations are also applied.

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

To facilitate a more nuanced probe of code embeddings, we propose an AST-based augmentation pipeline that creates synthetic clones and/or injects synthetic bugs. We describe 10 "positive" (semantics-preserving) and 11 "negative" (bug-injecting) transforms for on-the-fly generation of Type II/III clones and buggy mutations, respectively, with fine-grained control over the type, frequency, and location of each. Using our approach, we build `pydra`,[1] an open-source dual clone and bug dataset with extensive meta-data to enable precise analysis. Since existing datasets are primarily in Java and C, we focus on Python, but our approach can be extended easily. As a standalone contribution, we open-source our underlying dataset `unified_code_contests_python`,[2] a de-duplicated, validated, quality-filtered superset of $\sim$9k Python examples from code competition sources.

Additionally, we perform a preliminary empirical analysis across existing code embedding models. In Section 5.1, we determine each model's baseline similarity for random, unrelated code pairs, finding large variance between models and significant dependence on both the overall and relative lengths of the code pairs. We then evaluate model performance on positive and negative pairs through a combined lens of similarity analysis (Section 5.2), binary classification (Section 5.3, and retrieval (Section 5.4).

2 Base dataset construction

We construct `unified_code_contests_python` by merging several existing code competition datasets along with additional data collected directly from competition websites. Table 1 describes the composition of our dataset, with a more granular breakdown by original source in Appendix B.1. We apply simple code normalization, thoroughly de-duplicate the merged dataset, and perform several additional quality filtering steps; we refer the reader to Appendix B.2 for full details. Some code competitions provide multiple verified solutions. For these, we choose the 2-3 most mutually dissimilar solutions, randomly picking one to be our main solution and the others to be alternates, using the process described in Appendix B.3. These alternates are included in the dataset to enable additional potential ablations e.g. layering augmentations on top of natural clones.

Table 1: Composition of `unified_code_contests_python`

Source	**# Examples**	**# Tests / Example**		**# Solutions* / Example**	
		Min	**Mean**	**Min**	**Mean**
DeepMind Code Contests	5,459	1	2.2	1	6
Project CodeNet	1,536	1	2.5	1	6
LeetCode Dataset	1,379	3	100	1	1
GeeksForGeeks	293	3	10	1	1
Google Code Jam	82	1	1	1	4
Project Euler	19	1	1	1	1
Total	**8,768**				

* after filtering for solution quality (Appendix B.2) and diversity (Appendix B.3)

3 Augmentation pipeline

The original code is parsed into an Abstract Syntax Tree (AST), manipulated at the syntactic level, and then serialized back into code and re-verified. We use the `tree_sitter`[3] library with Python grammar and add our own helpers to preserve scoping, indentation, protected names, etc. Our code transforms can be semantics-preserving (positive transforms), or semantics-altering in controlled ways, such as injecting subtle bugs (negative transforms). All transforms are initialized with a

[1] LINK TO BE ADDED
[2] LINK TO BE ADDED
[3] https://github.com/tree-sitter/tree-sitter

sampling probability that controls how many valid nodes in a given code example are actually transformed, with a floor of 1 node and a default of p=1.0 (all valid nodes) if not otherwise specified.

3.1 Positive transforms

We define 10 semantics-preserving transforms in Table 2. In the terminology of clone literature, applying `ChangeNames` alone produces Type II clones, while the combination of `ChangeNames` and any of the other transforms (with the exception of `CommentDeletion`) produces Type III clones. Specific transforms can only be applied to certain nodes under certain conditions, and are thus not necessarily possible in every example. In practice, `ChangeNames` is always applicable. Figure 1 shows the distribution of valid examples and nodes for the other transforms. For all positive pairs, we verify that the positive augmented code has no syntax errors and still passes all tests.

Table 2: Positive transforms

Transform	Description
ArithmeticTransform	Converts augmented assignments to expanded form and vice versa.
SwapCondition	Swaps the operands of simple binary comparisons e.g. `a < b` → `b > a`, `x == y` → `y == x`, etc.
ForInRangeToWhile	Transforms for-loops where the iterator is range, xrange, enumerate, zip, or tqdm into equivalent while loops.
CommentDeletion	Removes comments.
ListCompToForLoop	Rewrites list comprehensions into equivalent explicit for-loops.
BooleanSimplify	Performs boolean simplifications by shortening comparisons e.g. `x is True` → `x`, `x is None` → `not x`, `not (x > y)` → `x <= y)`, etc.
ChainedComparisonToAnd	Splits chained comparisons into boolean `and` expressions, e.g. `a < b <= c` → `(a < b) and (b <= c)`.
ConditionalExprToIfElse	Converts ternary conditional expressions in assignments (e.g. `var = val_1 if cond else val_2`) into multi-line if/else blocks.
FStringToFormat	Converts simple f-strings into equivalent .format() calls.
ChangeNames	Consistently renames functions/classes, variables, parameters, etc. while preserving scope. Multiple strategies for name generation (Appendix B.4).

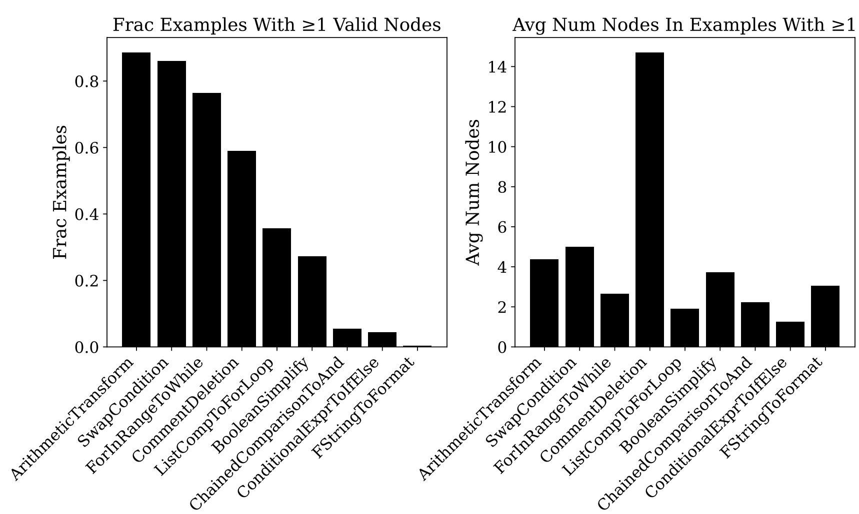


Figure 1: Fraction of examples with at least one valid node for each positive transform (left) and average number of nodes (right). `ChangeNames` is excluded as all examples have many valid nodes.

3.2 Negative transforms

We define 11 semantics non-preserving transforms in Table 3. In the terminology of test mutation literature, these are sometimes called “mutants.” For negative pairs, we verify that the augmented code fails to pass tests; this is to avoid accidental so-called “equivalent mutants.” Again, aside from `DeletedStatement` and `TypoInName`, the other transforms aren’t necessarily applicable to all code examples. Figure 2 shows the distribution of valid examples and nodes for the other transforms.

Table 3: Negative transforms

Transform	Description
WrongArithmeticOperator	Replaces an operator `["+", "-", "*", "/", "%", "//", "**"]` with a wrong operator, chosen at random.
WrongComparisonOperator	Replaces an operator `[">", ">=", "<", "<=", "==", "!=", "is", "is not", "in", "not in"]` with a wrong operator, chosen at random from the subset of operators most likely to produce a different effect.
WrongBooleanValue	Swaps True and False constants.
WrongBooleanOperator	Swaps `and` and `or` operators.
WrongAugAssignOperator	Similar replacement as WrongArithmeticOperator but for augmented assignments `["+=", "-=", "*=", "/="]`.
RangeOffByOne	Introduces off-by-one errors in for-loops with range().
NumberWrongSign	Flips signs of floats and ints.
NumberWrongValue	Changes values of floats and ints (same order-of-magnitude).
RemoveNegation	Removes `not` operators before expressions.
DeletedStatement	Deletes a statement. Can either target statements where a variable is being assigned for the first time or any statement.
TypoInName	Randomly introduces typos in variable or parameter usages (not definitions) by either deleting a character or swapping adjacent characters.

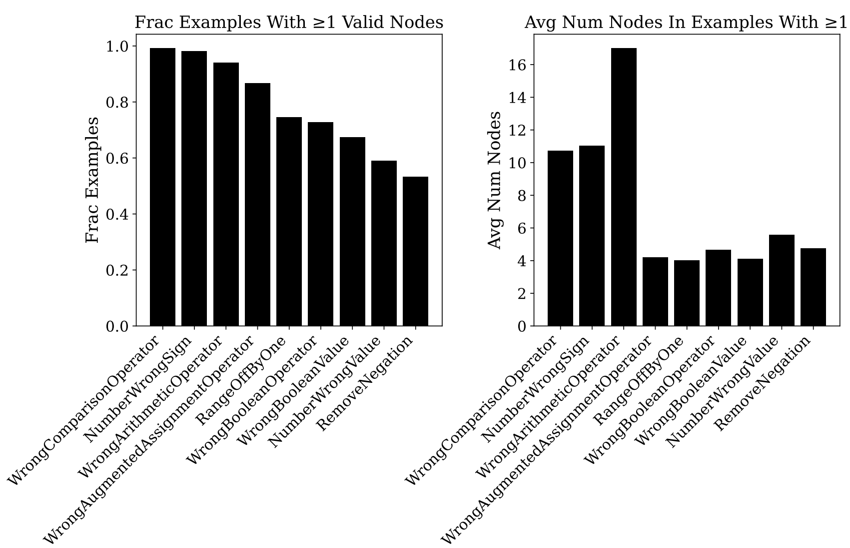


Figure 2: Fraction of examples with at least one valid node for each negative transform (left) and average number of nodes (right). `DeletedStatement` and `TypoInName` are excluded as all examples have many valid nodes.

4 Experimental setup

4.1 Models

We experiment on a wide set of state-of-the-art code embedding models, spanning a range of sizes, architectures, and pooling mechanisms. These models were chosen because they are top-performing across benchmarks in code clone detection (BigCloneBench [**?**]), fault detection (Devign [12]), and code-code search (CSN-CCR [13]). Recent work has focused around contrastive training on in-the-wild text-code pairs, and of our set of baselines, only CodeT5+ and StarEncoder were not fine-tuned in this manner. We exclude CodeBERT from our analysis due to its context window of 512 tokens falling below the median length of examples in our dataset (see Figure 9).

Table 4: Baseline embedding models

Embed Model	Base Model	Type	Size	Max Seq	D_{emb}	Pooling
Nomic Embed Code [14]	Qwen2.5-Coder-7B-Instruct	Decoder	7B	8192	768	Last
CodeXEmbed-7B [15]	Mistral-7B-Instruct-v0.3	Decoder	7B	4096	4096	Last
SFR-Embedding-Code-2 [15]	Google-Gemma-2-2B	Decoder	2B	32k	2048	Last
CodeSage-large [16]	-	Encoder	1.3B	2048	1024	Mean
CodeT5+ [17]	T5	Encoder-Decoder	770M	$512^{\dagger}$	1024	Mean
Jina-Code-v2 [18]	-	Encoder	161M	8192	768	Mean
StarEncoder [19]	-	Encoder	125M	1024	768	Mean

†Extended to 4096 with relative positional encodings

4.2 Augmentation settings

For our experiments, we apply all possible positive transforms maximally, in order to create the most adversarial pairs. These settings are shown in Listing 5a. We further filter to examples with at least 5 transforms applied, since this still leaves over 60% of our dataset, as shown in Figure 4. For negative transforms, we choose a reasonably aggressive set of transforms and filter to examples with at least six transforms applied (see Figure 4 and Listing 5b); in contrast to the positive pairs, we are deliberately not making negative pairs maximally adversarial. In our analysis, we chose to generate and analyze one positive and one negative pair for each example, though we note that our augmentation pipeline allows us to generate many pairs, which we do for the construction of `pydra`.

5 Experimental results

5.1 Similarity of random pairs

The average cosine similarity between random code pairs in our dataset, modulated by model and token bin, are shown in Figure 6. We begin by noting that different embedding models exhibit different baseline similarities for random pairs. These baseline values are not standardized or inherently interpretable (e.g., they do not necessarily cluster around 0 or 0.5), which is an artifact of the model training, in which there is no explicit enforcement of a specific similarity distribution for unrelated examples. Critically, we also find that all models are, to varying degrees, sensitive to code length, with most models predicting higher similarity between random pairs of longer code. For some, such as Nomic Embed Code, this increase is dramatic.

Other models, such as CodeT5+ 770M, do not show much dependence on code length as long as the two codes are of similar length, but we find that they are very sensitive to *relative* length differences. In Figure 7, we calculate the average similarity between pairs from potentially different token bins for

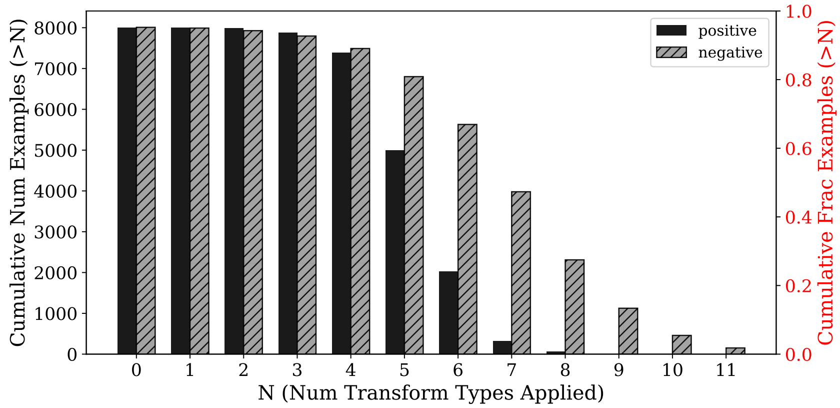


Figure 3: Positive transforms

Figure 4: Cumulative number and fraction of examples with at least N types of transforms applied.

Listing (1) PositivePipeline

```
CommentDeletion(),
ArithmeticTransform(),
SwapCondition(),
ForInRangeToWhile(),
ListCompToForLoop(),
ConditionalExprToIfElse(),
ChainedComparisonToAnd(),
BooleanSimplify(),
FStringToFormat(),
ChangeNames(rename_strategy="funky")
```

(a) Positive pipeline settings

Listing (2) NegativePipeline

```
WrongArithmeticOperator(sample_p=0.05) with p=0.5,
WrongComparisonOperator(sample_p=0.1) with p=0.5,
WrongBooleanValue(sample_p=0.5),
WrongBooleanOperator(sample_p=0.5),
WrongAugmentedAssignOperator(sample_p=0.25) with p=0.5,
RangeOffByOne(sample_p=0.5),
NumberWrongSign(sample_p=0.1) with p=0.5,
NumberWrongValue(sample_p=0.5) with p=0.01,
RemoveNegation(sample_p=0.5),
DeletedStatement(statement_type="simple_def", sample_p=0.08),
TypoInName(typo_type="missing_char", sample_p=0.08) with p=0.5
```

(b) Negative pipeline settings

Figure 5: The experimental settings chosen for the positive and negative augmentation pipelines.

Nomic Embed Code and CodeT5+ 770M. While Nomic Embed's length sensitivity appears along the vertical and main diagonal, there is only minor variance along the horizontal direction, implying that the model does not rely on length mismatches as a signal that two codes are dissimilar. In contrast, CodeT5+ similarities do not show much length sensitivity as long as the codes are similar length, but vary with the relative difference, becoming more dissimilar the more mismatched the lengths are.

5.2 Similarity of positive and negative pairs

We investigate how models score the similarity of positive pairs $\text{PP} = \{(x_{\text{orig},i}, x_{\text{pos},i})\}$ and negative pairs $\text{NP} = \{(x_{\text{orig},i}, x_{\text{neg},i})\}$ relative to random pairs $\text{RP} = \{(x_{\text{orig},i}, x_{\text{orig},j})\}_{i \neq j}$. The mean and standard deviation for each model are given in Table 5 and also visualized in Figure 8.

In addition to $\langle\text{RP}\rangle$, we calculate $\langle\text{RP}\rangle_\ell$, a weighted average of the 2D token-binned $\langle\text{RP}\rangle_{k,m}$ values that we derived in the analysis of the previous section:

$$\langle\text{RP}\rangle_\ell = \frac{1}{N}\sum_i^N \langle\text{RP}\rangle_{k,m} \text{ s.t. } \begin{cases} x_{\text{orig},i} \in \text{token bin } k, \\ x_{\text{pos},i} \in \text{token bin } m \end{cases} \tag{1}$$

The reason we use the positive pairs to set the bin weighting is that the positive augmentations tend to shift and broaden the token distribution (see Figure 9 in Appendix **??**) due to the extra characters added by the default renaming strategy, whereas the negative augmentations do not. The average values of $\langle\text{RP}\rangle$ and $\langle\text{RP}\rangle_\ell$ are only marginally different, since the positive augmentations tend to only shift by one token bin, but we nevertheless use $\langle\text{RP}\rangle_\ell$ when appropriate to disentangle the influence of our augmentations from the effects of simply comparing two codes of differing lengths.

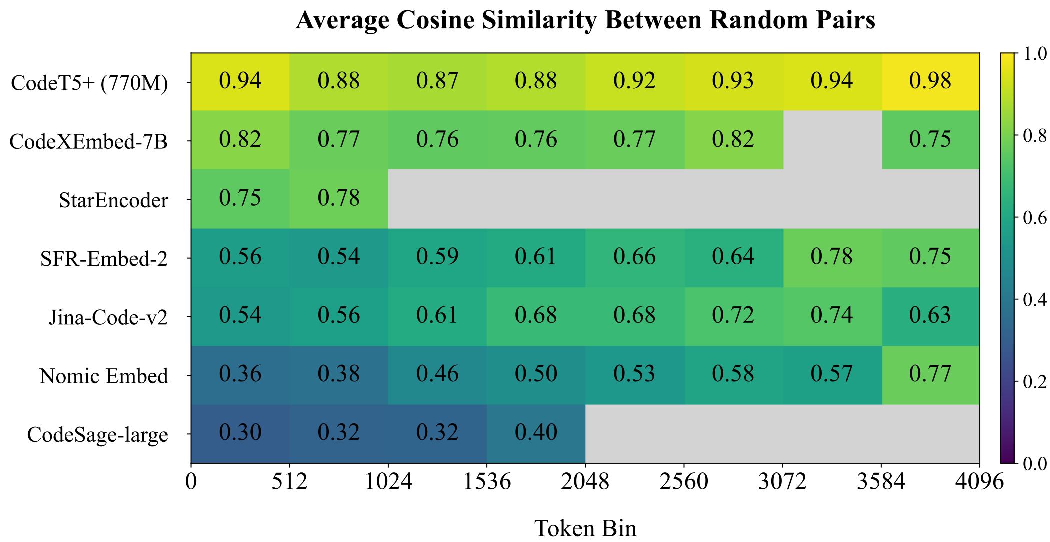


Figure 6: The cosine similarity of code embeddings generated by each model, averaged across random pairs of original code within token bins.

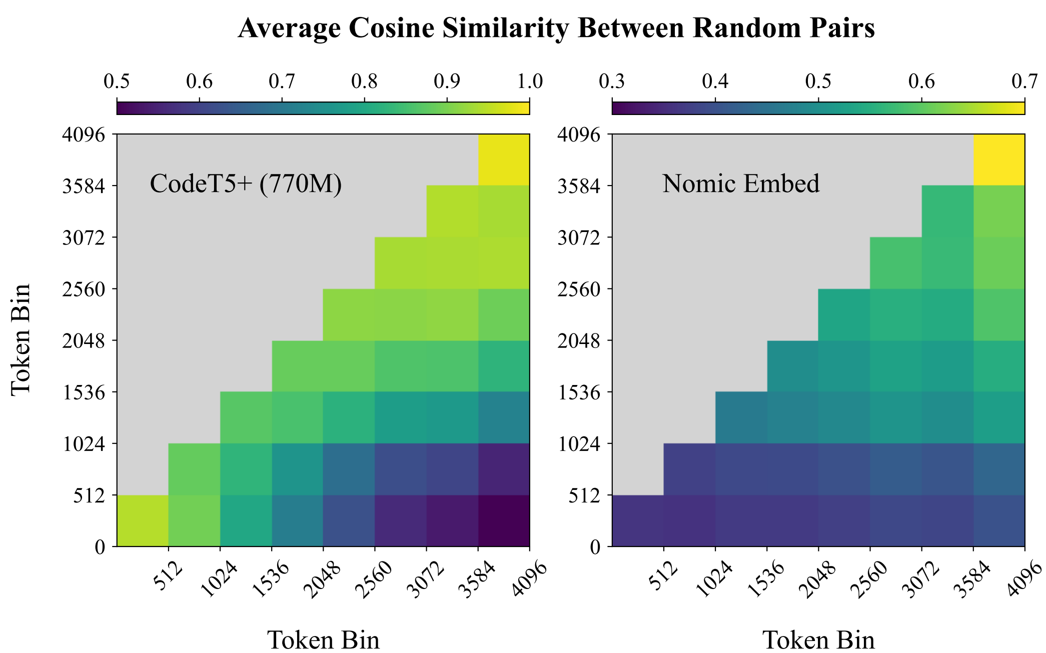


Figure 7: The cosine similarity of code embeddings from CodeT5+ 770M (left) and Nomic Embed Code (right), with the (i, j)th bin averaged across all random pairs of original code where one example's length falls within token bin i and the other's within token bin j.

A separate benefit to our length normalization is that it effectively up-weights the token bins where most of the examples lie and down-weights the noisier bins with fewer examples, hence why $\langle \mathrm{RP} \rangle$ has significantly larger standard deviation than $\langle \mathrm{RP} \rangle_\ell$.

Table 5: Cosine similarities of embedding pairs

Model	Cosine Similarity (Mean ± Std Dev)			
	⟨PP⟩	**⟨RP⟩**	**$\langle$RP$\rangle_\ell$**	**⟨NP⟩**
CodeSage-large	0.40 ± 0.10	0.30 ± 0.13	0.29 ± 0.02	0.90 ± 0.08
Nomic Embed Code	0.61 ± 0.08	0.37 ± 0.13	0.41 ± 0.06	0.97 ± 0.02
SFR-Embedding-Code-2	0.83 ± 0.07	0.54 ± 0.07	0.57 ± 0.04	0.93 ± 0.05
Jina-Code-v2	0.65 ± 0.08	0.55 ± 0.13	0.58 ± 0.04	0.98 ± 0.02
CodeXEmbed-7B	0.83 ± 0.06	0.77 ± 0.06	0.76 ± 0.01	0.98 ± 0.02
StarEncoder	0.82 ± 0.13	0.77 ± 0.13	0.77 ± 0.01	0.99 ± 0.02
CodeT5+ (770M)	0.83 ± 0.05	0.84 ± 0.08	0.83 ± 0.04	0.99 ± 0.01

PP = Positive Pairs, RP = Random Pairs, NP = Negative Pairs, $\langle\rangle_\ell$ = Length Normalized

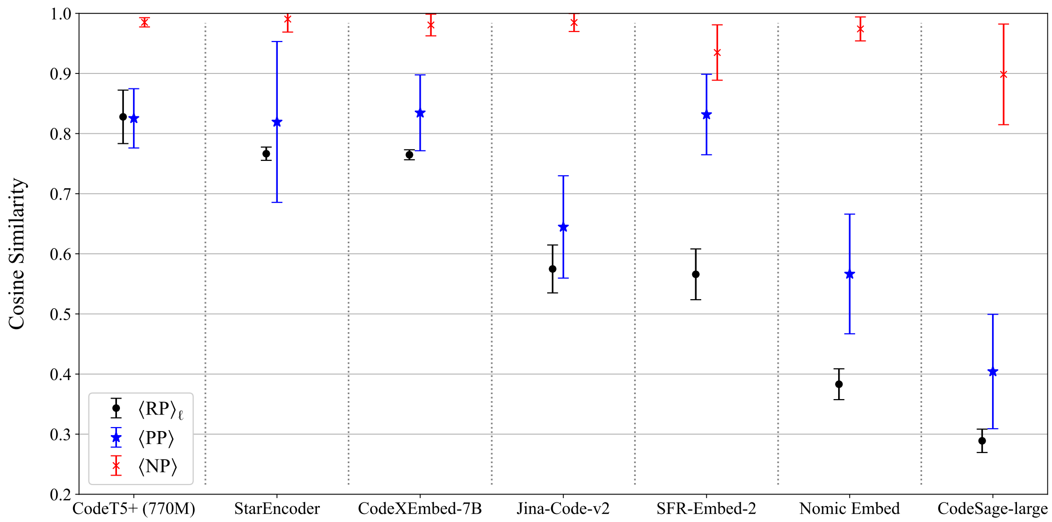


Figure 8: For each embedding model, we plot the average similarity of the embeddings of all positive pairs, negative pairs, and random pairs, with error bars representing the standard deviation.

5.3 Classification of positive vs. negative pairs

In order to perform binary classification on the positive and negative pairs using their respective pair-wise cosine similarities, we must carefully choose the classification threshold, since cosine similarities are model- and scale-dependent, as demonstrated in Section 5.1. The most principled threshold is each model's $\langle\text{RP}\rangle_\ell$, the length-normalized average cosine similarity between random pairs defined in the previous section. Alternatively, we can calculate the ROC-AUC, which measures how well positives and negatives can be distinguished by the model across all thresholds. Both sets of results are given in Table 6.

Recall, defined as the ratio of positives that are correctly classified, is a key evaluation metric for clone detection benchmarks such as BigCloneBench. In those benchmarks, there are a set of clone pairs (analogous to our positive pairs) and a larger set of non-clone pairs (analogous to our random pairs), but no matching "hard negative" pair for each clone pair (our negative pairs). As such, it is generally discouraged to report accuracy or precision, as those metrics depend on the count of true and false negatives. Using random non-clones for the negative class is both noisy and often misleading because there are typically many more non-clones than clones. We note that our classes are perfectly balanced, with every example belonging to exactly one positive pair PP and one negative pair NP. Our setup is unique in that it allows us to report accuracy and precision, which contain considerably more information about performance than recall alone.

Table 6: Classification of positive vs. negative

Model	Threshold: $\langle \text{RP} \rangle_\ell$			ROC-AUC (%)
	Acc (%)	**Precision (%)**	**Recall (%)**	
CodeSage-large	44.0	46.8	87.9	0.79
Nomic Embed Code	**49.1**	**49.5**	**98.2**	0.16
SFR-Embedding-Code-2	**49.4**	**49.7**	**98.8**	**11.5**
Jina-Code-v2	40.2	44.6	80.4	0.51
CodeXEmbed-7B	40.9	45.0	81.8	0.80
StarEncoder	29.4	37.0	48.7	**5.33**
CodeT5+ (770M)	9.3	15.6	18.4	0.30

PP = Positive Pairs, RP = Random Pairs, NP = Negative Pairs, $\langle\rangle_\ell$ = Length Normalized Mean

Most of the models we consider[4] achieve 80% or above on recall in the $\langle \text{RP} \rangle_\ell$-thresholded classification setting. SFR-Embedding-Code-2 and Nomic Embed Code score the highest on recall by a significant margin. These two models also have the clearest visual separation between PP and $\langle \text{RP} \rangle_\ell$ in Figure 8. On the other hand, all models score at or below random performance when it comes to precision and accuracy, where the misclassification of the negative pairs as false positives dominates. Since nearly all negative pairs are misclassified, we see 50% accuracy/precision when recall is close to 100% and lower than 50% when some positive pairs are also misclassified.

This behavior is even more apparent in the ROC-AUC values, where only SFR-Embedding-Code-v2 and StarEncoder achieve greater than 1%. Looking at Figure 8, we see that the other models not only fail to rank negative pairs below positive pairs, but they in fact cleanly separate the two groups *in the wrong direction*, consistently predicting cosine similarities close to 1 for negative pairs. Here, AOC-ROC captures the fact that, at certain thresholds $> \langle \text{RP} \rangle_\ell$, those models misclassify all samples, both positive and negative. SFR-Embedding-Code-v2 and StarEncoder are the only models that show any overlap between positive and negative pairs, though still in the wrong direction.

We posit in Section 5.5 that this may be due to negatives having substantially higher token overlap with the original examples than positives because of the renaming augmentation. This suggests that name-preserving, semantics-changing transforms have significantly less effect on similarity than semantics-preserving, name-changing transforms.

5.4 Ranking of positive vs. random pairs

We also evaluate the models on a retrieval-style formulation of the task, with results given in Table 7. Top-1 accuracy is the fraction of queries for which the true clone c_i is the top-ranked candidate, while Mean Reciprocal Rank (MRR) is the average over the reciprocal of the rank of the true clone:

$$\text{MRR (\%)} = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{\text{rank}(c_i)} \times 100 \tag{2}$$

Mean Average Precision (MAP), another common retrieval metric, is equivalent to MRR in our setting since there is only one clone for each query. Since MRR explicitly depends on the size of the non-clone pool, we set it to a fixed value of $n = 100$; for each positive pair, i.e. an original example and its clone, we take the 100 random examples with the greatest similarity to the original as the rest of our candidate pool. Since we already know from Section 5.3 that the models almost always rank negative pairs above positive pairs, we exclude them.

Instead, we wish to probe how the models rank positive pairs relative individual random pairs. The recall statistic in Section 5.3 is related but subtly different, as it measures how well the models capture positive pairs relative to a threshold set by the average similarity of random pairs. Interestingly, we

[4] Except StarEncoder and CodeT5+, notably the only two of our baselines not contrastively fine-tuned.

Table 7: Ranking of positive vs. 100 randoms

Model	Top-1 Acc (%)	MRR
CodeSage-large	10.3	17.2
Nomic Embed Code	19.9	28.0
SFR-Embedding-Code-2	**78.3**	**84.1**
Jina-Code-v2	3.75	7.32
CodeXEmbed-7B	**32.0**	**42.8**
StarEncoder	19.2	22.6
CodeT5+ (770M)	1.46	2.96

PP = Positive Pairs, RP = Random Pairs

find that these two metrics do not perfectly correlate. While SFR-Embedding-Code-v2 outperforms all other models on both, CodeXEmbed-7B achieves higher top-1 accuracy than Nomic Embed Code despite achieving lower recall, and the same is observed for StarEncoder vs. both Codesage-large and Jina-Code-v2. This highlights that overall performance is related not only on the relative averages of similarities across PP, NP, and RP, but also their relative spreads.

5.5 Digging deeper into the impact of renaming

For positive augmentations, we find that `ChangeNames` has by far the largest effect. With all transforms except `ChangeNames` applied, the similarity of positive pairs is significantly closer to 1.0 across models. In Table 8, we examine this setting for the three models with the largest positive-negative gap from Figure 8 (Jina-Code-v2, Nomic Embed Code, and CodeSage-large), and show that the gap is mostly closed. We then examine two other settings in which only `ChangeNames` applied: the first uses our default strategy `FunkyRenamer` (which creates long Docker-style "adjective_noun" names), while the other uses a `RandomRenamer` strategy that replaces names with random strings *of the same length* as the original names. The latter strategy, which by definition keeps the code length fixed, scores higher similarity than the FunkyRenamer strategy, which introduces characters.

Table 8: Cosine similarity between PP under three ablations of the settings

Model	Only `FunkyRename`	Only `RandomRename`	Exclude `ChangeNames`
Jina-Code-v2	0.71	0.81	0.98
Nomic Embed Code	0.65	0.73	0.96
CodeSage-large	0.49	0.55	0.94

6 Conclusions

We presented our AST-based pipeline for generating controlled positive and negative code variants, and used it to build `pydra`, an open-source dataset of clones and bugs in Python. Our empirical evaluation of leading code embedding models suggest that they remain dominated by surface-level heuristics such as token overlap and relative length. We find that renaming-based augmentations have the strongest influence on similarity scores, especially when renamings increase code length, whereas models are insensitive to small but behavior-critical edits. Our negative pairs present a challenging and subtle type of "hard negative" and we hope that `pydra` will serve the research community.

References

[1] Neha Saini, Sukhdip Singh, and Suman. Code clones: Detection and management. *Procedia Computer Science*, 132:718–727, 01 2018.

[2] Yutao Xie, Jiayi Lin, Hande Dong, Lei Zhang, and Zhonghai Wu. Survey of code search based on deep learning. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–42, 2023.

[3] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2(1), 2023.

[4] Jimmy Lin, Rodrigo Nogueira, and Andrew Yates. *Pretrained transformers for text ranking: Bert and beyond*. Springer Nature, 2022.

[5] Xinyu Shi, Zhenhao Li, and An Ran Chen. Enhancing llm-based fault localization with a functionality-aware retrieval-augmented generation framework. *arXiv preprint arXiv:2509.20552*, 2025.

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

[7] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

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

[9] Hao Yu, Xing Hu, Ge Li, Ying Li, Qianxiang Wang, and Tao Xie. Assessing and improving an evaluation dataset for detecting semantic code clones via deep learning. *ACM Trans. Softw. Eng. Methodol.*, 31(4), July 2022.

[10] Jeffrey Svajlenko and Chanchal K. Roy. Evaluating clone detection tools with bigclonebench. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 131–140, 2015.

[11] Sabaat Haroon, Ahmad Faraz Khan, Ahmad Humayun, Waris Gill, Abdul Haddi Amjad, Ali Raza Butt, Mohammad Taha Khan, and Muhammad Ali Gulzar. How accurately do large language models understand code? *ArXiv*, abs/2504.04372, 2025.

[12] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *CoRR*, abs/1909.03496, 2019.

[13] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

[14] Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. Cornstack: High-quality contrastive data for better code retrieval and reranking, 2025.

[15] Ye Liu, Rui Meng, Shafiq Joty, Silvio Savarese, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. Codexembed: A generalist embedding model family for multiligual and multi-task code retrieval, 2025.

[16] Dejiao Zhang, Wasi Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. Code representation learning at scale, 2024.

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

[18] Michael Günther, Jackmin Ong, Isabelle Mohr, Alaeddine Abdessalem, Tanguy Abel, Mohammad Kalim Akram, Susana Guzman, Georgios Mastrapas, Saba Sturua, Bo Wang, Maximilian Werk, Nan Wang, and Han Xiao. Jina embeddings 2: 8192-token general-purpose text embeddings for long documents, 2024.

[19] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023.

[20] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics.

[21] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In Pierre Isabelle, Eugene Charniak, and Dekang Lin, editors, *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics.

[22] Aryaz Eghbali and Michael Pradel. Crystalbleu: Precisely and efficiently measuring the similarity of code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[23] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020.

[24] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert, 2020.

[25] Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. Codebertscore: Evaluating code generation with pretrained models of code. 2023.

[26] Nickil Maveli, Antonio Vergari, and Shay B Cohen. What can large language models capture about code functional equivalence? *arXiv preprint arXiv:2408.11081*, 2024.

[27] Atharva Naik. On the limitations of embedding based methods for measuring functional correctness for code generation, 2024.

[28] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. What do they capture? a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th international conference on software engineering*, pages 2377–2388, 2022.

[29] Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. ReCode: Robustness evaluation of code generation models. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13818–13843, Toronto, Canada, July 2023. Association for Computational Linguistics.

[30] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

[31] Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. Reasoning runtime behavior of a program with llm: How far are we? *arXiv preprint arXiv:2403.16437*, 2024.

[32] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. Can large language models reason about program invariants? In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 27496–27520. PMLR, 23–29 Jul 2023.

[33] Thanh Le-Cong, Bach Le, and Toby Murray. Can llms reason about program semantics? a comprehensive evaluation of llms on formal specification inference, 2025.

[34] Ziyu Li and Donghwan Shin. Mutation-based consistency testing for evaluating the code understanding capability of llms. *2024 IEEE/ACM 3rd International Conference on AI Engineering – Software Engineering for AI (CAIN)*, pages 150–159, 2024.

[35] Ashish Hooda, Mihai Christodorescu, Miltiadis Allamanis, Aaron Wilson, Kassem Fawaz, and Somesh Jha. Do large code models understand programming concepts? counterfactual analysis for code predicates. *arXiv preprint arXiv:2402.05980*, 2024.

[36] Wei Ma, Shangqing Liu, Zhihao Lin, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, Li Li, and Yang Liu. Lms: Understanding code syntax and semantics for code analysis, 2024.

[37] Thu-Trang Nguyen, Thanh Trong Vu, Hieu Dinh Vo, and Son Nguyen. An empirical study on capability of large language models in understanding code semantics, 2024.

[38] Sergey Troshin and Nadezhda Chirkova. Probing pretrained models of source code. *arXiv preprint arXiv:2202.08975*, 2022.

[39] Anjan Karmakar and Romain Robbes. Inspect: Intrinsic and systematic probing evaluation for code transformers. *IEEE Transactions on Software Engineering*, 50(2):220–238, 2023.

[40] Saiteja Utpala, Alex Gu, and Pin-Yu Chen. Language agnostic code embeddings. In Kevin Duh, Helena Gomez, and Steven Bethard, editors, *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 678–691, Mexico City, Mexico, June 2024. Association for Computational Linguistics.

[41] Shounak Naik, Rajaswa Patil, Swati Agarwal, and Veeky Baths. Probing semantic grounding in language models of code with representational similarity analysis. In *International Conference on Advanced Data Mining and Applications*, pages 395–406. Springer, 2022.

[42] Zhuohao Li, Wenqing Chen, Jianxing Yu, and Zhichao Lu. Functional consistency of llm code embeddings: A self-evolving data synthesis framework for benchmarking. *Expert Systems with Applications*, page 129523, 2025.

[43] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 1287–1293, 2016.

[44] Jeffrey Svajlenko and Chanchal K. Roy. Bigclonebench: A retrospective and roadmap. In *2022 IEEE 16th International Workshop on Software Clones (IWSC)*, pages 8–9, 2022.

[45] Jens Krinke and Chaiyong Ragkhitwetsagul. Bigclonebench considered harmful for machine learning. In *2022 IEEE 16th International Workshop on Software Clones (IWSC)*, pages 1–7, 2022.

[46] Jens Krinke and Chaiyong Ragkhitwetsagul. How the misuse of a dataset harmed semantic clone detection. *arXiv preprint arXiv:2505.04311*, 2025.

[47] Farouq Al-Omari, Chanchal K Roy, and Tonghao Chen. Semanticclonebench: A semantic code clone benchmark using crowd-source knowledge. In *2020 IEEE 14th International Workshop on Software Clones (IWSC)*, pages 57–63. IEEE, 2020.

[48] Ajmain Inqiad Alam, Palash Ranjan Roy, Farouq Al-omari, Chanchal Kumar Roy, Banani Roy, and Kevin Schneider. Gptclonebench: A comprehensive benchmark of semantic clones and cross-language clones using gpt-3 model and semanticclonebench. In *Proceedings of the 39th International Conference in Software Maintenance and Evolution (ICSME 2023)*. October 2023, Bogota, Colombia (to appear), 2023.

[49] Chanchal K. Roy and James R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th IEEE International Conference on Program Comprehension*, pages 172–181, 2008.

[50] Md Nahidul Islam Opu, Shaowei Wang, and Shaiful Chowdhury. Llm-based detection of tangled code changes for higher-quality method-level bug datasets. *arXiv preprint arXiv:2505.08263*, 2025.

[51] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 1556–1560, 2020.

[52] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

[53] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks, 2021.

[54] AIZU Online Judge. Aizu online judge. `https://judge.u-aizu.ac.jp/`. Accessed: 2025-10-28.

[55] AtCoder Inc. Atcoder: Programming contest site. `https://atcoder.jp/`. Accessed: 2025-10-28.

[56] Ethan Caballero, OpenAI, and Ilya Sutskever. Description2code dataset. `https://github.com/ethancaballero/description2code`, 2016. Accessed: 2025-10-28.

[57] CodeChef. Codechef: Learn and practice coding with problems. `https://www.codechef.com/`. Accessed: 2025-10-28.

[58] HackerEarth. Hackerearth: Validity and reliability of assessments. https://marketplace.jazzhr.com/wp-content/uploads/HackerEarth-Validity-and-reliability-of-assessments.pdf. Accessed: 2025-10-28.

[59] Codeforces. Codeforces: Programming contests and programming challenges. https://codeforces.com/. Accessed: 2025-10-28.

[60] Yunhui Xia, Wei Shen, Yan Wang, Jason Klein Liu, Huifeng Sun, Siyue Wu, Jian Hu, and Xiaolong Xu. Leetcodedataset: A temporal dataset for robust evaluation and efficient training of code llms, 2025.

[61] GeeksforGeeks. Geeksforgeeks: "https://www.geeksforgeeks.org/". https://www.geeksforgeeks.org/. Accessed: 2025-10-28.

[62] Google Inc. Google code jam: Programming contest site. https://codingcompetitions.withgoogle.com/codejam. Accessed: 2025-10-28.

[63] zibada.guru. Google coding competitions archive. https://zibada.guru/gcj/, 2023. Unofficial archive of Google Code Jam, Kick Start, and Hash Code competitions (data up to 2022; updated May 2023).

A Related work

Measuring code similarity. Traditional similarity metrics such as ROUGE [20], BLEU [21], and CrystalBLEU [22] primarily rely on n-gram overlap, rewarding lexical closeness rather than semantic agreement. CodeBLEU [23] incorporates data-flow graph (DFG) and abstract syntax tree (AST) matching, but code clones usually have different AST and often different DFG as well (due to code refactoring, intermediate variables, order of independent computations, etc), even when considering clones in the same programming language. More recent methods such as BERTScore [24] and CodeBERTScore [25] compare contextual embeddings at the token level. These metrics relax the exact token match requirement but still hinge on semantic similarity between individual identifiers and keywords, and have been shown to be poor predictors of functional equivalence or correctness [26, 27]. Another approach is to use pooled embeddings of entire code fragments or units. Ideally, these representations encode high-level abstractions and are invariant under semantics-preserving perturbations. In practice, however, they may be dominated by surface-level features. Pooling also introduces the potential risk of information loss on longer structured programs. Evaluating the quality of pooled code embeddings is an open research challenge and the focus of our empirical study.

Probing semantic code understanding. A growing body of work investigates the ability of LLMs to reason abstractly about code. Most of this analysis is focused on the generative setting [11, 26, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37], though a handful of works investigate how well embedding models capture code semantics, either by training linear, task-specific classifiers to probe the frozen embeddings [38, 39, 40] or by inspecting the embeddings directly [41, 42].

Generative models. Multiple complementary approaches to evaluating LLMs for code understanding have been proposed, including novel generative tasks based on predicting runtime behavior [30, 31, 36] or inferring semantic [32, 33] or structural [28, 36] artifacts from input code. Several recent works probe code understanding via controlled code perturbations that may be either semantics-preserving [33], semantics-altering [11], or both [11, 26, 34, 35, 36, 37]. Tasks include predicting the equivalence of a pair of codes [26] or a code and its description [34], or localizing faulty lines [11]. Others evaluate the LLM's ability to correctly complete code after perturbing the docstrings [29] or partial code prefixes [29, 35] in the prompt. In [37], the authors include a code summarization task that uses the cosine similarity (under a text embedding model) of LLM-generated code summaries of a code pair as a proxy for the semantic similarity of the code pair itself under the LLM. Across a swathe of tasks and models, these works generally support the finding that LLMs are oversensitive to semantics-preserving augmentations and under-sensitive to subtle semantics-altering augmentations.

Embedding models. Both [38] and [39] include downstream tasks classifying correct vs. incorrect code wherein the incorrect code was produced by injecting various types of errors into the correct code, and both report poor performance across models for those tasks. [40] focuses on cross-lingual embeddings, suggesting that language-specific syntactic features and language-agnostic semantic features can be decoupled, and that eliminating the former improves performance on code retrieval. In [41], the authors measure "semantic grounding" by comparing two dissimilarity matrices, the first across a set of source codes from CodeNet, the second across the corresponding set of their natural language descriptions. In [42], embedding models are evaluated on an evolved version of POJ-104 [43] constructed by using ChatGPT to generate multiple augmented versions of each example.

Clone datasets. BigCloneBench [**?**] has long served as the canonical code clone dataset in the clone detection literature. However, BigCloneBench is Java only, and significant concerns about its use in AI research have been raised [44, 45, 46], citing issues of noise and ambiguity, high rates of false positives, and pervasive data leakage. Additionally, [9] showed that many Type III/IV clones in the dataset share identifier names, leading to inflated performance numbers. More recently, SemanticCloneBench [47] provides 1000 function-level pairs each for Java, C, C#, and Python, collected from StackOverflow. Hoping to scale this to a size more suitable for deep learning applications, GPTCloneBench [48] prompts GPT-3 to generate additional semantic clones, including cross-lingual clones, for the examples in SemanticCloneBench. Since the goal of SemanticCloneBench/GPTCloneBench is to focus on semantic (Type IV) clones, both attempt to filter out syntactic clones using a combination of tools such as NiCAD [49] and manual review by human judges. Nevertheless, there are no guarantees that all clones are true semantic clones or even clones at all. Aside from potentially introducing noise and false positives, the data collection process inherently restricts the controllability and diversity of the resulting clones, and the focus on semantic clones, while useful in many respects, inhibits analyses aiming to disentangle semantic and syntactic factors in model performance.

Bug/defect/fault datasets. Most datasets and benchmarks related to debugging and code repair are execution-based and rely on in-the-wild code, revision histories, and patches, often with "tangled changes" (see e.g.[50] and references therein). Classic examples include Defects4J [**?**], which aggregates real-world Java bugs and developer fixes, and its newer Python equivalent, BugsInPy [51] and PyTraceBugs. While these datasets benefit from capturing realistic bug scenarios from real libraries, they miss the advantage of paired "ground-truth" variants that differ by controlled, fine-grained, and self-contained transforms.

B Dataset details

B.1 Base dataset by data source

The largest subset is sourced from DeepMind Code Contests [52], which is itself composed of problems from Project CodeNet [53] (scraped from AIZU [54] and AtCoder [55]) and Description2Code [56] (scraped from CodeChef [57], HackerEarth [58], and Codeforces [59]), along with additional problems scraped from Codeforces. The second largest part of our dataset is the LeetCode Dataset [60]. Finally, we add extra problems from Project CodeNet and from scraping the GeeksForGeeks [61], Google Code Jam [62], and Project Euler [63] websites.

Table 9: Detailed composition of `unified_code_contests_python` by source

Source	**# Examples**	**# Tests / Example**		**# Solutions* / Example**	
		Min	**Mean**	**Min**	**Max**
DeepMind Code Contests	5,459	1	2.2	1	6
– AIZU	1215	1	1.4	1	3
– AtCoder	1114	1	2.9	1	3
– Codeforces	3130	1	2.2	1	6
Project CodeNet	1,536	1	2.5	1	6
– AIZU	676	1	1.7	1	3
– AtCoder	860	1	3.0	1	6
LeetCode Dataset	1,379	3	100	1	1
GeeksForGeeks	293	3	10	1	1
Google Code Jam	82	1	1	1	4
Project Euler	19	1	1	1	1
Total	**8,768**				

* after filtering for solution quality (Appendix B.2) and diversity (Appendix B.3)

B.2 Normalization, de-duplication, and filtering

Normalization. Submissions are first normalized by applying NFC (Normalization Form C) Unicode normalization to ensure consistent character representation. Line endings are standardized by converting carriage return (CR, `\r`) and carriage return + line feed (CRLF, `\r\n`) sequences to a single line feed (LF, `\n`). Byte Order Marks (BOMs) and other non-standard whitespace characters are removed. Shebang lines (e.g., `#!/usr/bin/env python`) and encoding cookies (e.g., `# -*- coding: utf-8 -*-`) are stripped. Tabs are converted to four spaces. All comments are removed, and module/class/function docstrings are pruned. Trailing whitespace is trimmed, consecutive blank lines are collapsed into one, and a single trailing newline is enforced. This standardized form is then used for de-duplication.

De-duplication. An AST-level SHA-256 hash is first computed over a version of the code stripped of docstrings and attribute names. This captures semantically identical submissions that differ only in formatting or comments. A second SHA-256 hash is computed over the problem statement text to eliminate any residual textual duplicates.

Validation and basic filtering. Safety and testability gates are then applied. Only code that parses successfully is retained. Submissions that import or call sensitive APIs—such as `os`, `subprocess`, `socket`, `requests`, or direct calls to `open`, `eval`, `exec`, `os.system`, `subprocess.*`, `Path.open`, and `sys.exit`—are excluded. Finally, only solutions with paired, non-empty input/output examples are kept to ensure they can be validated.

Validation is execution-based and occurs in an isolated Python environment with memory and time limits. The program's output is compared against the expected output, and only submissions that pass all provided tests are retained.

Additional quality filtering. Additional filters are applied to weed out solutions that are trivially constructed to "hack" the competition environment; along with programs that simply print the expected test outputs (i.e. `print("a\nb\nc\nd")`), the cleaning process filters out "lookup-table" solutions that bypass computation by relying on precomputed answers. We also remove sources dominated by oversized literals, and highly repetitive code containing few unique lines.

B.3 Choosing dissimilar natural solutions

For problems with multiple solutions, the solutions are aggressively de-duplicated in two stages.

First, one solution is randomly selected as the "gold" reference. The other solutions are embedded with all available models (as long as the given solution does not exceed that model's maximum sequence length) and their cosine similarity with the gold solution is calculated. Solutions whose similarity exceeds the model-specific threshold for *any* model are removed as duplicates.

Then, the surviving alternate solutions (those judged dissimilar from the gold) then undergo pairwise de-duplication to detect near-duplicates amongst themselves:

- **Similarity graph construction**: Within each problem, we build an undirected graph where nodes represent surviving submissions. Two submissions are connected by an edge if their cosine similarity exceeds the model-specific threshold for *any* model. The union of edges from all models forms a single combined similarity graph per problem. Edge creation is transitive: if Model 1 links $A \leftrightarrow B$ and Model 2 links $B \leftrightarrow C$, all three solutions belong to the same component.
- **Single-linkage clustering**: Depth-first search identifies connected components in the combined graph. Each component represents a cluster of near-duplicate solutions. Within each cluster, we retain a single solution as the canonical representative and prune all others.

Model-specific thresholds are calibrated by starting with each model's baseline similarity $\langle RP \rangle$ (see Table 5) and then systematically relaxing each threshold until the pipeline admitted at most one or two alternate solutions per problem. The calibrates thresholds are 0.8, 0.6, 0.5, 0.9, 0.9, 0.9, and 0.9 for Salesforce/SFR-Embedding-Code-2B_R, nomic-ai/nomic-embed-code, codesage/codesage-large-v2, Salesforce/SFR-Embedding-Mistral, bigcode/starencoder, jinaai/jina-embeddings-v2-base-code, and Salesforce/codet5p-220m, respectively.

B.4 Different strategies for `ChangeNames` augmentation

Three main renaming strategies are implemented:

- **`random_chars`**: Generates new names using random letters. Name length can either be fixed to a user-specified positive integer value, sampled uniformly between 1 and 10, forced to be equal to the original name length, or restricted to the shortest possible unique name.
- **`shuffle_original`**: Shuffles the characters of the original name (single-character names remain unchanged).
- **`funky`**: Generates "Docker-style" adjective-noun names using the `funkybob`[5] package.

All generated names are by default lower-case. For strategies that preserve name length (i.e. `shuffle_original` or `random_chars` with the original length setting), additional user options include matching the casing pattern and/or underscore index locations of the original name.

[5] https://github.com/andreacorbellini/funkybob

All strategies ensure generated names are unique within the relevant scope. If a valid new name cannot be generated, the original name is retained.

B.5 Other dataset statistics

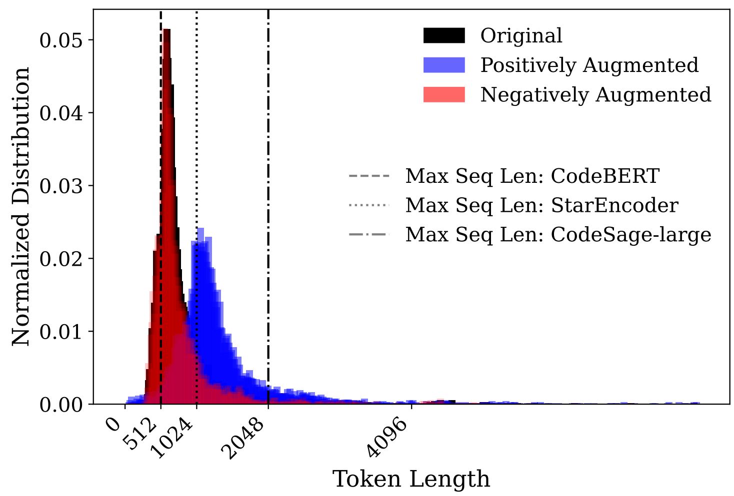


Figure 9: Distribution of token lengths of original, positively augmented, and negatively augmented code, under experimental settings described in Section 4.2 and using Jina-Code-v2 tokenizer.

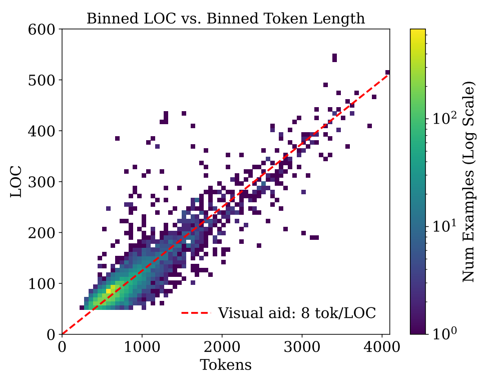
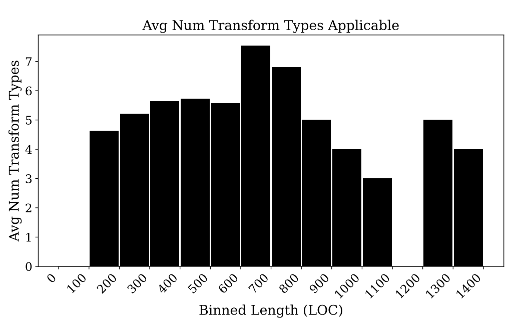
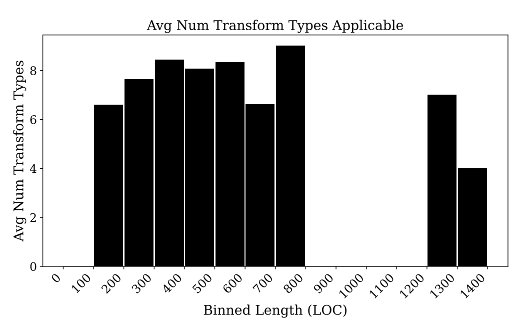


Figure 10: Scaling of tokens with LOC from original examples using Jina-Code-v2 tokenizer.



(a) Positive transforms.



(b) Negative transforms.

Figure 11: Lines of code statistics for positive and negative transforms.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: **[TODO]**

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: **[TODO]**

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

Justification: **[TODO]**

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: **[TODO]**

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: **[TODO]**

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyper-parameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

Answer: [Yes]

Justification: **[TODO]**

Guidelines:

- The answer NA means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: **[TODO]**

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.

- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: **[TODO]**

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: **[TODO]**

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: **[TODO]**

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to

generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.

- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: **[TODO]**

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: **[TODO]**

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: **[TODO]**

Guidelines:

- The answer NA means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: **[TODO]**

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: **[TODO]**

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [NA]

Justification: **[TODO]**

Guidelines:

- The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.