Journal of Measurement and Evidence
Volume 1 · Number 1 · 2026

Likelihood Normalization Changes Which Code Candidate Wins

Cameron Marshall[1] · David Harper[2] · Paul Lewis[3]

[1] Greece · [2] Estonia · [3] Latvia

Original publication: October 2, 2026 | Article 612

Abstract

The same model chose the test-supported reference in sixteen of twenty-six program pairs when candidates were scored by mean token loss, but in only nine pairs when total continuation loss was used. The two conventions disagreed on nine programs. Scores came from actual Pythia-14M forward passes after removing comments and inert explanatory strings from captured QuixBugs sources. Reference eligibility was established separately through bounded execution of the supplied JSON cases. Mean reference-minus-buggy token loss was -0.013648 nats, with an interval spanning zero; total loss was 4.097314 nats higher for references on average. These findings concern score construction for fixed candidates, rather than repair generation or calibrated program-correctness probabilities.

A small-model continuation audit of finite-test-supported QuixBugs source pairs, with no generated repairs or preference fine-tuning.

Likelihood Normalization Changes Which Code Candidate Wins

A small-model continuation audit of finite-test-supported QuixBugs source pairs, with no generated repairs or preference fine-tuning.

Abstract

The same model chose the test-supported reference in sixteen of twenty-six program pairs when candidates were scored by mean token loss, but in only nine pairs when total continuation loss was used. The two conventions disagreed on nine programs. Scores came from actual Pythia-14M forward passes after removing comments and inert explanatory strings from captured QuixBugs sources. Reference eligibility was established separately through bounded execution of the supplied JSON cases. Mean reference-minus-buggy token loss was -0.013648 nats, with an interval spanning zero; total loss was 4.097314 nats higher for references on average. These findings concern score construction for fixed candidates, rather than repair generation or calibrated program-correctness probabilities.

1. A code preference needs an observable definition

When two source programs are available, a researcher can ask which one a language model prefers. That question remains incomplete until the score and its normalization are stated. Multiplying continuation probabilities favors a different quantity from averaging token log probabilities. Programs also differ in length, so the choice of convention can alter a ranking even when both scores come from exactly the same forward pass. A reference label supplied by tests is a third object, with its own limited evidence.

Our research question is whether lower continuation loss selects the finite-test-supported reference in a captured buggy/reference pair, and whether that selection changes with length normalization. The primary description is the number of reference selections under mean-per-token and candidate-total loss. We also estimate average paired score differences. Neither score is assumed in advance to be a calibrated probability of functional correctness, and no new code is sampled from the model.

The preference-learning problem in Tao and Kamata, 2026 makes this construct distinction relevant. Our model receives no preference fine-tuning, and the recorded comparisons are not a replication of that work. The experiment instead inspects two ways to manufacture a candidate preference from ordinary continuation likelihood. Keeping that operation explicit helps distinguish a model score from the behavioral evidence that supports a benchmark label or a human judgment about which edit is useful.

2. The source pairs and the available test interface

The public benchmark in Lin et al., 2017 supplies buggy and corrected Python source directories. We use exact archive paths rather than selecting files by a loosely matching name. Thirty-one programs have the captured line-delimited JSON test interface used in this analysis. The benchmark's other graph-oriented tasks are outside that interface and are not silently counted as evaluated. Every available test record supplies positional input arguments and an expected result, allowing a reproducible finite compatibility gate.

We execute the reference and buggy versions under the previously reviewed pure-function restrictions, supplying only reviewed standard-library helpers. File access, dynamic evaluation, private-attribute access, and other disallowed operations are rejected by source inspection. A source-line tracing limit stops

a call after 100,000 executed lines. Copied JSON arguments prevent one version's input mutation from changing another version's case. No repository test runner or model-generated program is executed.

Generator results are materialized, tuple and list representations are normalized, and numerical comparisons use relative and absolute tolerances of one part in ten billion. There are 242 captured cases across the thirty-one initial programs. A program enters the likelihood study only if its reference passes every case within this gate and its buggy implementation has at least one nonpassing case. This produces twenty-six eligible source pairs and 198 corresponding cases. The label is supported by these finite tests and operational bounds, not by a universal proof about all inputs.

3. Exclusions define a restricted benchmark population

Five initial programs do not meet the reference gate: `knapsack`, `levenshtein`, `longest_common_subsequence`, `possible_change`, and `sqrt`. Four encounter at least one reference call exceeding the line budget; `sqrt` has two strict numerical mismatches. A line-budget event is not proof that a program never terminates, and a stringent oracle mismatch is not proof that its mathematics is generally incorrect. Excluding such programs keeps the measured label tied to a declared operational criterion.

Benchmarking concerns in Liu et al., 2023, Austin et al., 2021, and Jain et al., 2024 give context for the importance of tests and evaluation conditions. None supplies extra cases or generated solutions here. Software-defect classification in Herzig et al., 2013 also motivates distinguishing wrong outputs, exceptions, and operational limits rather than treating every nonpass as the same defect mechanism. We do not estimate the prevalence of defects in real software from this curated algorithmic collection.

The empirical-software guidance represented by Kitchenham et al., 2002 and repository-analysis perspectives in Bird et al., 2009 reinforce the need to preserve the source population. Twenty-six selected algorithmic pairs are not a random sample of coding-assistant requests, repository patches, or professional review decisions. The code-review setting in Bacchelli and Bird, 2013 contains contextual judgments absent from a short function benchmark. We keep the finite reference-selection result separate from those broader notions of edit quality.

4. Scoring executable syntax without attached explanation text

Captured source files contain comments and bare-string explanations, including some explanatory strings after definitions rather than in formal leading docstring positions. Leaving those notes in one candidate would make the comparison partly about attached prose. We remove inert bare-string expression statements at all positions, including leading docstrings and trailing module explanations. String constants used as values remain intact. The remaining syntax is rendered through the same AST unparser and receives one terminal newline.

The normalized text is reparsed and checked for exact AST equality with the transformed tree. Thus formatting is standardized without replacing the executable algorithm or renaming identifiers. Eligibility tests refer to the separately captured source implementations. Removing explanatory strings is not offered as a proof of general semantic equivalence under every external introspection operation; it defines the code text whose continuation likelihood is measured within this restricted source gate.

For each pair the identical prefix is `Python function NAME:` followed by a newline. It supplies the task name but no complete natural-language specification, demonstrations, or correctness feedback. We score the full normalized module continuation, including imports when present. The tokenized common prefix is verified to match the beginning of each full sequence, and decoding reproduces the scored text exactly.

No target token is dropped or truncated, and all candidate contexts fit within the model's 2,048-position capacity.

5. Actual scores from an untuned small language model

The scorer is the final standard-Pile Pythia-14M checkpoint associated with Biderman et al., 2023. We use numeric safetensors values promoted from float16 to float32 and load the entire state strictly. Forward passes run locally on the CPU through Paszke et al., 2019 version 2.6.0+cpu and Wolf et al., 2019 version 4.57.1, with evaluation mode, eager attention, caching disabled, and four threads. Hugging Face contributors, 2025 version 0.22.1 supplies the captured official segmentation. These are actual conditional next-token scores, not estimates inferred from source length or syntax counts.

The general code-model setting in Chen et al., 2021, Li et al., 2023, and Guo et al., 2024 differs substantially from this small untuned scorer. Code-oriented representations in Guo et al., 2020 and completion settings in Fried et al., 2022 supply further context, without providing models or embeddings to our calculation. We do not report competitive code-generation accuracy or infer the behavior of those systems from Pythia. The checkpoint's possible exposure to benchmark code during pretraining is not independently ruled out.

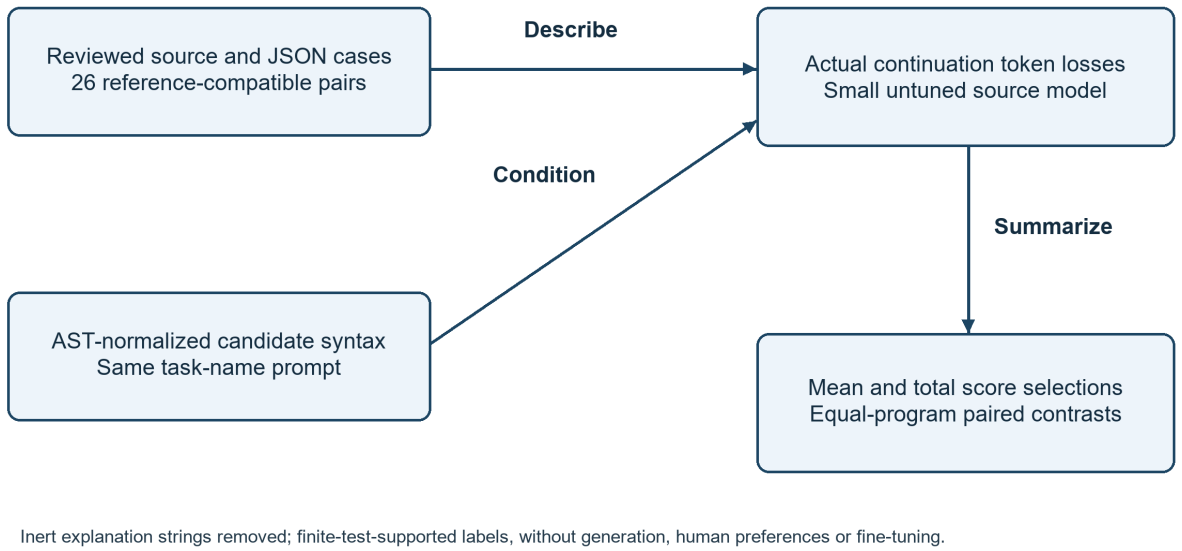

Figure 1. Bounded tests establish a restricted reference label; normalized source pairs receive actual model scores before two length conventions select a candidate.

Preference optimization in Rafailov et al., 2023 and feedback-based learning in Ouyang et al., 2022 use additional training information and objectives. The reinforcement-learning code setting in Le et al., 2022 also differs from ranking fixed source strings without adaptation. We cite these distinctions because an ordinary likelihood convention should not be renamed a learned human preference, reward model, or trained edit-selection policy. No gradients, training examples, or preference updates are applied in the executed experiment.

6. Mean and total loss select different quantities

For each candidate we compute next-token negative log likelihood only over continuation tokens, excluding the shared prompt. Mean loss divides their sum by the candidate token count. Total loss retains the sum. Lower values define the preferred candidate under the respective convention, and an exact tie would be recorded separately. The totals describe the likelihood of the scored finite continuation prefix; we append no special completion marker that would define a probability of a terminated program.

Buggy candidates contain between thirty-four and 182 scored tokens, while references contain between thirty-three and 186. References are longer by 2.076923 tokens on average, but individual length differences have both signs. Mean and total losses therefore need not rank the pair alike. The predictive-scoring perspective in Gneiting and Raftery, 2007 supports naming the quantity directly. Neither raw total nor length-normalized mean is automatically a functional-correctness probability merely because it comes from a language model.

We fit separate intercept-only ordinary least-squares regressions to reference-minus-buggy mean loss and reference-minus-buggy total loss over the twenty-six programs. Covariance is HC3 with normal-reference 95% intervals, following the heteroskedasticity-consistent framework of White, 1980. The two outcomes have different units: nats per continuation token and nats per full scored continuation. Their coefficients should not be compared numerically as though they were the same effect scale.

Table 1. Average paired source-score contrasts, reference minus buggy, over twenty-six eligible programs.

Model	Term	Estimate (SE)	95% CI	N
M1	Mean outcome/contrast	-0.013648 (0.015070)	[-0.043186, 0.015889]	26
M1: fit	R-squared = 0.0000	-	-	26
M2	Mean outcome/contrast	4.097314 (1.359297)	[1.433140, 6.761487]	26
M2: fit	R-squared = 0.0000	-	-	26

Model labels: M1 = Mean-per-token reference minus buggy NLL; M2 = Total-candidate reference minus buggy NLL.

The mean-only fits have effectively zero R-squared. Programs, rather than their tokens or test cases, supply the inferential rows. Dependence considerations in Cameron and Miller, 2015 remain a limit because related algorithmic tasks and one jointly trained model do not establish an independent task population. The HC3 intervals describe a hypothetical task-draw interpretation of this fixed collection. They do not quantify uncertainty in a deterministic preference count or validate transfer to another checkpoint.

7. The selected reference count depends on normalization

Under mean token loss, sixteen of twenty-six programs select the test-supported reference. Under total continuation loss, only nine do. No exact ties occur. The conventions disagree on nine pairs, approximately 34.6 percent of the eligible collection. This disagreement is an empirical ranking result from the same candidate scores. It cannot be explained by a different test population or a different model checkpoint, because those objects are held fixed within each pair.

Average mean loss favors references by 0.013648419 nats per token: the reference-minus-buggy coefficient is -0.013648419, with standard error 0.015070355 and interval -0.043185771 to 0.015888934. The conditional interval includes zero. Total loss instead has a positive average difference of 4.097313765

nats, standard error 1.359297129, and interval 1.433140347 to 6.761487182. That sign favors buggy candidates under the total-loss convention on average.

The aggregate score differences and reference-choice counts are related but not identical. A few large total differences can influence the mean, while the count gives every pair one binary choice. Neither result is a repair success rate: both candidate implementations were provided before scoring, and the model produces no edit. We also avoid calling every finite-test-supported reference universally correct or treating a model preference for the buggy version as evidence about a human reviewer's preference.

8. Two concrete reversals show the length issue

For `bucketsort`, the buggy continuation contains seventy-four tokens and the reference seventy-five. The reference's mean loss is lower by 0.031350 nats per token, yet its total loss is higher by 0.165951 nats. Mean scoring therefore chooses the reference while total scoring chooses the buggy candidate. A one-token length difference can be enough to reverse a close pair. The example does not require an extreme difference in program size or a different prompt.

The reversal runs in the opposite direction for `flatten`. Its reference has forty-five tokens compared with forty-nine for the buggy version. Reference mean loss is higher by 0.045766 nats per token, but total loss is lower by 7.453919 nats. Thus total scoring chooses that reference while mean scoring does not. These examples demonstrate that the disagreement is not a universal bias in favor of one directory label; it reflects how the two conventions combine per-token scores and varying lengths.

Influence reasoning associated with Cook, 1977 would motivate examining the pairs with the largest score differences. We retain every eligible program rather than trimming those that contradict a preferred convention. No causal decomposition estimates how much of each ranking change is due to length, particular identifiers, or edited syntax. The mathematical relationship between sum and mean is known, while its interaction with these specific source pairs is the observed result.

9. A score-construction audit has a narrower claim than assistant evaluation

Validation issues in Roberts et al., 2017 and selection bias in Varma and Simon, 2006 limit reuse of these outcomes to choose a scoring convention and then claim independent validation on the same programs. A full assessment would require additional tasks and a declared role for each set. It could also provide complete task specifications, evaluate a code-specialized model, or compare learned preference scores. Those are prospective experiments, rather than capabilities established by the present small-model likelihood calculation.

Interpretation following Wasserstein and Lazar, 2016 emphasizes the effect units and uncertainty, particularly the mean-loss interval that spans zero. The nine ranking disagreements remain directly observable even when an aggregate mean contrast is uncertain. Conversely a total-loss interval above zero does not identify an ideal replacement preference rule. The appropriate score depends on what selection is intended to accomplish and what evidence supports that objective. A convenient string-probability convention does not settle the construct by itself.

The benchmark includes curated algorithm defects, finite tests, and an operational runtime gate. Real edit selection also involves maintainability, compatibility, and task-specific constraints. The name-only prefix provides little of that information. We do not infer a practical assistant's willingness to make a requested change, its generated-patch correctness, or the outcome of pairwise fine-tuning. The completed study concerns how two explicit likelihood summaries rank fixed benchmark candidates under limited context.

10. State the preference rule before counting successes

This audit shows that a fixed model can yield substantially different reference-selection counts when the same continuation losses are totaled or averaged. Both scores are reproducible, while neither is a self-defining measure of code correctness. The result supports reporting candidate length, normalization, prompt content, label support, and transition disagreements whenever likelihood is used to create a code preference. It does not establish that one convention is generally best for editing or generation.

Documentation under Gebru et al., 2021 records source directories, exclusions, bounded oracle outcomes, explanatory-string removal, AST checks, prompt boundaries, and token counts. A related distinction arises in production-failure diagnosis through distributed tracing, the subject of Yan, 2026: evidence must retain the context of the event being measured. Our likelihood scores are not runtime traces, and the present fixed-candidate experiment identifies neither service failures nor their causes. Preserving the scored text, checkpoint and per-program contrasts makes the narrower comparison recoverable without suggesting that a candidate-scoring convention supplies operational fault diagnosis.

References

1. Ouyang, Long; Wu, Jeff; Jiang, Xu; Almeida, Diogo; Wainwright, Carroll L.; Mishkin, Pamela; Zhang, Chong; Agarwal, Sandhini; Slama, Katarina; Ray, Alex; Schulman, John; Hilton, Jacob; Kelton, Fraser; Miller, Luke; Simens, Maddie; Askell, Amanda; Welinder, Peter; Christiano, Paul; Leike, Jan; Lowe, Ryan. (2022). Training language models to follow instructions with human feedback. arXiv:2203.02155. https://arxiv.org/abs/2203.02155.
2. Jain, Naman; Han, King; Gu, Alex; Li, Wen-Ding; Yan, Fanjia; Zhang, Tianjun; Wang, Sida; Solar-Lezama, Armando; Sen, Koushik; Stoica, Ion. (2024). LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. arXiv:2403.07974. https://arxiv.org/abs/2403.07974.
3. Hugging Face contributors, . (2025). Tokenizers v0.22.1 [Software]. https://github.com/huggingface/tokenizers/releases/tag/v0.22.1.
4. Gneiting, Tilmann; Raftery, Adrian E. (2007). Strictly Proper Scoring Rules, Prediction, and Estimation. Journal of the American Statistical Association. https://doi.org/10.1198/016214506000001437.
5. Herzig, Kim; Just, Sascha; Zeller, Andreas. (2013). It’s not a bug, it’s a feature: How misclassification impacts bug prediction. https://doi.org/10.1109/ICSE.2013.6606585.
6. Liu, Jiawei; Xia, Chunqiu Steven; Wang, Yuyao; Zhang, Lingming. (2023). Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. arXiv:2305.01210. https://arxiv.org/abs/2305.01210.
7. Gebru, Timnit, et al. "Datasheets for datasets" *Communications of the ACM*, 2021. https://doi.org/10.1145/3458723.
8. Paszke, Adam; Gross, Sam; Massa, Francisco; Lerer, Adam; Bradbury, James; Chanan, Gregory; Killeen, Trevor; Lin, Zeming; Gimelshein, Natalia; Antiga, Luca; Desmaison, Alban; Köpf, Andreas; Yang, Edward; DeVito, Zach; Raison, Martin; Tejani, Alykhan; Chilamkurthy, Sasank; Steiner, Benoit; Fang, Lu; Bai, Junjie; Chintala, Soumith. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. https://arxiv.org/abs/1912.01703.
9. Bacchelli, Alberto; Bird, Christian. (2013). Expectations, outcomes, and challenges of modern code review. 2013 35th International Conference on Software Engineering (ICSE). https://doi.org/10.1109/icse.2013.6606617.
10. Kitchenham, Barbara A.; Pfleeger, Shari Lawrence; Pickard, Lesley M.; Jones, Peter W.; Hoaglin, David C.; El Emam, Khaled; Rosenberg, Jarrett. (2002). Preliminary guidelines for empirical research in software engineering. https://doi.org/10.1109/TSE.2002.1027796.
11. Chen, Mark; Tworek, Jerry; Jun, Heewoo; Yuan, Qiming; Pinto, Henrique Ponde de Oliveira; Kaplan, Jared; Edwards, Harri; Burda, Yuri; Joseph, Nicholas; Brockman, Greg; Ray, Alex; Puri, Raul; Krueger, Gretchen; Petrov, Michael; Khlaaf, Heidy; Sastry, Girish; Mishkin, Pamela; Chan, Brooke; Gray, Scott; Ryder, Nick; Pavlov, Mikhail; Power, Alethea; Kaiser, Lukasz; Bavarian, Mohammad; Winter, Clemens; Tillet, Philippe; Such, Felipe Petroski; Cummings, Dave; Plappert, Matthias; Chantzis, Fotios; Barnes, Elizabeth; Herbert-Voss, Ariel; Guss, William Hebgen; Nichol, Alex; Paino, Alex; Tezak, Nikolas; Tang, Jie; Babuschkin, Igor; Balaji, Suchir; Jain, Shantanu; Saunders, William; Hesse, Christopher; Carr, Andrew N.; Leike, Jan; Achiam, Josh; Misra, Vedant; Morikawa, Evan; Radford, Alec; Knight, Matthew; Brundage, Miles; Murati, Mira; Mayer, Katie; Welinder, Peter; McGrew, Bob; Amodei, Dario; McCandlish, Sam; Sutskever, Ilya; Zaremba, Wojciech. (2021). Evaluating Large Language Models Trained on Code. arXiv:2107.03374. https://arxiv.org/abs/2107.03374.

12. Lin, Derrick; Koppel, James; Chen, Angela; Solar-Lezama, Armando. (2017). QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge. Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity. https://doi.org/10.1145/3135932.3135941.

13. Austin, Jacob; Odena, Augustus; Nye, Maxwell; Bosma, Maarten; Michalewski, Henryk; Dohan, David; Jiang, Ellen; Cai, Carrie; Terry, Michael; Le, Quoc; Sutton, Charles. (2021). Program Synthesis with Large Language Models. arXiv:2108.07732. https://arxiv.org/abs/2108.07732.

14. White, Halbert. "A Heteroskedasticity-Consistent Covariance Matrix Estimator and a Direct Test for Heteroskedasticity." *Econometrica*, vol. 48, no. 4, 1980, pp. 817-838. https://doi.org/10.2307/1912934.

15. Wolf, Thomas; Debut, Lysandre; Sanh, Victor; Chaumond, Julien; Delangue, Clement; Moi, Anthony; Cistac, Pierric; Rault, Tim; Louf, Rémi; Funtowicz, Morgan; Davison, Joe; Shleifer, Sam; von Platen, Patrick; Ma, Clara; Jernite, Yacine; Plu, Julien; Xu, Canwen; Scao, Teven Le; Gugger, Sylvain; Drame, Mariama; Lhoest, Quentin; Rush, Alexander M.. (2019). HuggingFace's Transformers: State-of-the-art Natural Language Processing. https://arxiv.org/abs/1910.03771.

16. Varma, Sudhir and Richard Simon. "Bias in error estimation when using cross-validation for model selection." *BMC Bioinformatics*, vol. 7, no. 1, 2006. https://doi.org/10.1186/1471-2105-7-91.

17. Cook, R. Dennis. (1977). Detection of Influential Observation in Linear Regression. Technometrics. https://doi.org/10.1080/00401706.1977.10489493.

18. Tao, Hongmo; Kamata, Hiroyuki. (2026). Teaching Coding Assistants What to Change: Fine-Tuning from Pairwise Preferences. Proceedings of the Second Workshop for REsearch on Agent Language Models (REALM 2026), 1363-1374. https://doi.org/10.18653/2026.realm-1.95.

19. Fried, Daniel; Aghajanyan, Armen; Lin, Jessy; Wang, Sida; Wallace, Eric; Shi, Freda; Zhong, Ruiqi; Yih, Wen-tau; Zettlemoyer, Luke; Lewis, Mike. (2022). InCoder: A Generative Model for Code Infilling and Synthesis. arXiv:2204.05999. https://arxiv.org/abs/2204.05999.

20. Yan, Biyun. (2026). From Observability to Causality: Detecting Production Failures in Microservices with Distributed Tracing. 2026 9th International Conference on Computer Information Science and Application Technology (CISAT), 1522-1525. https://doi.org/10.1109/CISAT71030.2026.11682113.

21. Roberts, David R., et al. "Cross-validation strategies for data with temporal, spatial, hierarchical, or phylogenetic structure." *Ecography*, vol. 40, no. 8, 2017, pp. 913-929. https://doi.org/10.1111/ecog.02881.

22. Rafailov, Rafael; Sharma, Archit; Mitchell, Eric; Ermon, Stefano; Manning, Christopher D.; Finn, Chelsea. (2023). Direct Preference Optimization: Your Language Model is Secretly a Reward Model. arXiv:2305.18290. https://arxiv.org/abs/2305.18290.

23. Guo, Daya; Ren, Shuo; Lu, Shuai; Feng, Zhangyin; Tang, Duyu; Liu, Shujie; Zhou, Long; Duan, Nan; Svyatkovskiy, Alexey; Fu, Shengyu; Tufano, Michele; Deng, Shao Kun; Clement, Colin; Drain, Dawn; Sundaresan, Neel; Yin, Jian; Jiang, Daxin; Zhou, Ming. (2020). GraphCodeBERT: Pre-training Code Representations with Data Flow. arXiv:2009.08366. https://arxiv.org/abs/2009.08366.

24. Biderman, Stella; Schoelkopf, Hailey; Anthony, Quentin; Bradley, Herbie; O'Brien, Kyle; Hallahan, Eric; Khan, Mohammad Aflah; Purohit, Shivanshu; Prashanth, USVSN Sai; Raff, Edward; Skowron, Aviya; Sutawika, Lintang; van der Wal, Oskar. (2023). Pythia: A Suite for Analyzing Large Language Models Across Training and Scaling. https://arxiv.org/abs/2304.01373.

25. Cameron, A. Colin, and Douglas L. Miller. "A Practitioner's Guide to Cluster-Robust Inference." *Journal of Human Resources*, vol. 50, no. 2, 2015, pp. 317-372. https://doi.org/10.3368/jhr.50.2.317.

26. Le, Hung; Wang, Yue; Gotmare, Akhilesh Deepak; Savarese, Silvio; Hoi, Steven C. H.. (2022). CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. arXiv:2207.01780. https://arxiv.org/abs/2207.01780.

27. Guo, Daya; Zhu, Qihao; Yang, Dejian; Xie, Zhenda; Dong, Kai; Zhang, Wentao; Chen, Guanting; Bi, Xiao; Wu, Y.; Li, Y. K.; Luo, Fuli; Xiong, Yingfei; Liang, Wenfeng. (2024). DeepSeek-Coder: When the Large Language Model Meets Programming -- The Rise of Code Intelligence. arXiv:2401.14196. https://arxiv.org/abs/2401.14196.

28. Bird, Christian; Rigby, Peter C.; Barr, Earl T.; Hamilton, David J.; German, Daniel M.; Devanbu, Premkumar. (2009). The promises and perils of mining git. https://doi.org/10.1109/MSR.2009.5069475.

29. Li, Raymond; Allal, Loubna Ben; Zi, Yangtian; Muennighoff, Niklas; Kocetkov, Denis; Mou, Chenghao; Marone, Marc; Akiki, Christopher; Li, Jia; Chim, Jenny; Liu, Qian; Zheltonozhskii, Evgenii; Zhuo, Terry Yue; Wang, Thomas; Dehaene, Olivier; Davaadorj, Mishig; Lamy-Poirier, Joel; Monteiro, João; Shliazhko, Oleh; Gontier, Nicolas; Meade, Nicholas; Zebaze, Armel; Yee, Ming-Ho; Umapathi, Logesh Kumar; Zhu, Jian; Lipkin, Benjamin; Oblokulov, Muhtasham; Wang, Zhiruo; Murthy, Rudra; Stillerman, Jason; Patel, Siva Sankalp; Abulkhanov, Dmitry; Zocca, Marco; Dey, Manan; Zhang, Zhihan; Fahmy, Nour; Bhattacharyya, Urvashi; Yu, Wenhao; Singh, Swayam; Luccioni, Sasha; Villegas, Paulo; Kunakov, Maxim; Zhdanov, Fedor; Romero, Manuel; Lee, Tony; Timor, Nadav; Ding, Jennifer; Schlesinger, Claire; Schoelkopf, Hailey; Ebert, Jan; Dao, Tri; Mishra, Mayank; Gu, Alex; Robinson, Jennifer; Anderson, Carolyn Jane; Dolan-Gavitt, Brendan; Contractor, Danish; Reddy, Siva; Fried, Daniel; Bahdanau, Dzmitry; Jernite, Yacine; Ferrandis, Carlos Muñoz; Hughes, Sean; Wolf, Thomas; Guha, Arjun; von Werra, Leandro; de Vries, Harm. (2023). StarCoder: may the source be with you!. arXiv:2305.06161. https://arxiv.org/abs/2305.06161.

30. Wasserstein, Ronald L.; Lazar, Nicole A.. (2016). The ASA Statement on p -Values: Context, Process, and Purpose. The American Statistician. https://doi.org/10.1080/00031305.2016.1154108.