WWTC@UniA at SemEval-2026 Task 13: BERT-based Code Authorship Detection and Qualitative Analysis

Linda Kupfer* **Lisa Hader*** **Christian Jaumann*** **Annemarie Friedrich**
University of Augsburg, Germany
{firstname.lastname}@uni-a.de

Abstract

This paper describes our system for SemEval-2026 Task 13 on detecting machine-generated code. We fine-tune small encoder-only models for detecting human-written versus machine-generated code and for identifying which large language model (LLM) family was used to obtain code. We find that a strong, general-purpose model (ModernBERT) outperforms models specifically pre-trained for the code domain. In the official evaluation, our system ranks 5th on subtask B and 6th on subtask C. Our detailed analysis reveals that comments and other natural language text that is part of the code snippets provide valuable information for identifying the LLM family that generated it. Moreover, we show that the embeddings of our finetuned ModernBERT do not distinguish well between LLM families, but they cluster human-written code by programming language.

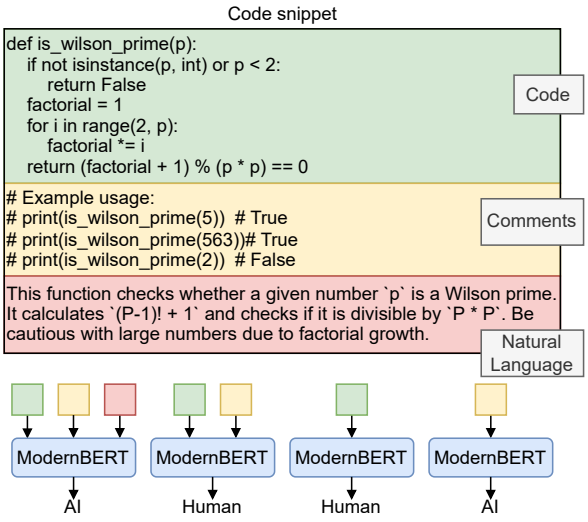


Figure 1: Code snippet of shared task dataset containing not only code and comments, but additional natural language explanations. We analyze the impact of these individual components.

1 Introduction

Large language models (LLMs) have recently become popular tools for programming code generation and code completion, substantially influencing modern software engineering workflows (Kumar, 2024). Although these capabilities increase productivity, they also raise concerns about the authorship, accountability, and trustworthiness of generated code (Haque, 2025). Being able to distinguish between human-written and machine-generated code is therefore of growing importance, e.g., for academic integrity, software maintenance, and intellectual property protection.

To address these challenges, SemEval-2026 Task 13[1] (Orel et al., 2026b) focuses on the detection of machine-generated code under diverse conditions. The shared task builds on the AICD dataset (Orel et al., 2026a) and is divided into three subtasks. Subtask A focuses on binary detection, distinguishing machine-generated and human-written code. Subtask B is a multi-class classification task, i.e., human-written code or code from one of 10 LLM families. Subtask C extends the setting of subtask A by adding a *hybrid* class (code partially written by an LLM) and an *adversarial* class, where the LLM is instructed to mimic human code. More details on the task can be found in Appendix A.1.

Most existing approaches for detecting AI-generated code are based on encoder-only models that are specifically pre-trained for the code domain and then further fine-tuned for detection (Nguyen et al., 2024; Xu et al., 2025; Orel et al., 2025a). In contrast, the best-performing approach of our team, *Who Wrote The Code? @ University of Augsburg* (WWTC@UniA), uses the general-purpose language model ModernBERT-large (Warner et al., 2025) and fine-tunes it for the detection task.

We found that several code snippets in the dataset, particularly machine-generated instances, contain natural language explanations also outside

*Equal contribution.
[1]https://github.com/mbzuai-nlp/SemEval-2026-Task13

Proceedings of the The 20th International Workshop on Semantic Evaluation (2026), pages 2359–2375

comments. To analyze how different types of content contribute to the model's ability to identify its source, we heuristically separate code snippets into pure code, comments, and natural language explanations (see Figure 1). We find that all of these parts play an important role, especially when classifying LLM family. Our analysis also tests several loss functions and data augmentation to mitigate class imbalance, and provides a visual analysis of the embedding space.

In the official evaluation, our system ranks 5th out of 34 teams in subtask B and 6th out of 32 teams in subtask C, demonstrating the effectiveness of our approach across different detection settings.

2 Related Work

Early approaches to detecting machine-generated code use statistics on code-level for decision tree learning (Li et al., 2023; Idialu et al., 2024). Nguyen et al. (2024) fine-tune CodeBERT (Feng et al., 2020) for the binary classification of whether a code snippet was written by ChatGPT[2] or a human. For the same detection task, Xu et al. (2025) propose a contrastive learning-based approach to fine-tuning UniXcoder (Guo et al., 2022). Orel et al. (2025a) evaluate several detection approaches, including traditional methods like decision trees and more advanced methods with fine-tuning CodeBERT, CodeT5 (Wang et al., 2021) and UniXcoder.

Orel et al. (2025b) train several different approaches, including a Graph Convolutional Network (Kipf and Welling, 2017), CatBoost (Prokhorenkova et al., 2018), and ModernBERT (Warner et al., 2025). As they identify ModernBERT as the best-performing model, we focus on this model.

3 System Description

To classify code snippets for all tasks, we fine-tune BERT-based models (Devlin et al., 2019). We add a classification head, which receives the embedding of the [CLS] token as input. Our model selection includes CodeBERT[3] (Feng et al., 2020), RoBERTa[4] (Liu et al., 2019), and ModernBERT-large[5] (Warner et al., 2025). For CodeBERT and RoBERTa, we use an input length of 512 tokens. Since ModernBERT supports longer input lengths and 16-44% of the instances in the dataset are longer than 512 tokens (see Appendix A.2), we experiment with input lengths of 512, 1024, and 2048 tokens. We do not evaluate longer inputs as very few instances exceed 2048 tokens. If a code snippet exceeds the input length, it is truncated.

Training settings. We perform hyperparameter optimization on ModernBERT with a maximum token length of 2048 and 10 trials using Optuna (Akiba et al., 2019). All models are trained for 10 epochs with a batch size of 128, a learning rate of 1.2e-4, a warm-up ratio of 0.073, a weight decay of 0.03, and a linear learning rate scheduler. We use early stopping with a patience of 5 (evaluating on the validation set every 1000 steps). Early stopping always occurred when training on the entire train set, usually after about 5 epochs. Because of the considerable class imbalance in task B (see Appendix A.2), instead of cross-entropy (CE) loss, we also evaluate focal loss (Lin et al., 2020) and weighted CE loss (Cui et al., 2019) for task B.

Data augmentation. To mitigate class imbalance, we experiment with data augmentation. For each model family used to generate the original dataset, we select a model from the corresponding family, except for OpenAI, as we only use open-weights models. The respective LLM is prompted to change the variable names and code structure while maintaining functionality. We use a temperature value of 0.3 to add variety to the generated code snippets and generate up to two additional examples for each original instance. The examples for both the human class and OpenAI remain unchanged. For the implementation details and resulting class distributions, see Appendix A.5.

LLM-based classification. We also evaluate the performance of LLMs in a 0-shot Chain of Thought (CoT) setting on subtask A. The LLM is instructed to determine whether the code was written by a human or a machine by focusing on indicators such as variable naming, comment style, and traces of generator outputs. To improve the decision-making process, we instruct it to "think step by step" (Kojima et al., 2022). For the exact prompt template, refer to Appendix A.5. To improve reproducibility, we run the LLM with a temperature of 0 and use an open-weights LLM, i.e., Qwen2.5-Coder-32B.[6]

Content type assignments. When manually inspecting the dataset provided for the shared task,

[2] https://chatgpt.com/
[3] https://huggingface.co/microsoft/codebert-base
[4] https://huggingface.co/FacebookAI/roberta-base
[5] https://huggingface.co/answerdotai/ModernBERT-large
[6] https://huggingface.co/Qwen/Qwen2.5-Coder-32B-Instruct

we found several instances that not only contain code lines, but also comments and additional natural language explanations not marked as comments. This occurs especially frequently in LLM-generated instances (see Appendix A.2). We hypothesize that these additional texts may provide a strong cue towards LLM code authorship.

To understand how the different content types within code snippets contribute to detecting LLM-generated code, we split the code snippets into three content types, i.e., pure code, comments within the code, and natural language. First, we identify whether each line is likely code (including comments) or natural language. Next, we check the code lines for language-specific comment characters to further separate comments from code. Indicators for code include programming language-specific keywords, syntax terminators (e.g., "{", "}", or "]:"), if the line starts with a language-specific comment character, or includes variable assignments or type pointers identified via regular expressions. We assume natural language text if the line exhibits a typical sentence structure (i.e., it starts with a letter and ends with a punctuation mark), if it contains at least five stopwords, or if it begins with bullet points or enumerations. To avoid outliers, the predictions are smoothed in comparison to the lines directly before and after. E.g., if the lines on either side are identified as code, then the middle line is also labeled as code.

We validate the heuristic on 100 randomly selected instances from the dataset, observing agreement with the human annotator, one of the authors holding an undergraduate degree in computer science, in 93 cases. The disagreements can be narrowed down to the heuristic incorrectly classifying lines of natural language as code. For a detailed description of the disagreements, see Appendix A.3.

4 Development Results and Ablations

This section presents the results of our approaches and several ablations on the validation set. We use the official evaluation metric of the shared task, the Macro-F1 score, which is computed as the unweighted arithmetic mean of the class-wise F1 scores. To evaluate the difficulty of each subtask, we use a random baseline that predicts class labels based on the empirical class distribution observed in the respective validation set (for details see Appendix A.4).

Model	max_len	A	B	C
ModernBERT	2048	**99.81**	67.58	**89.38**
+ focal loss	2048	-	67.17	-
+ weighted-ce	2048	-	**68.53**	-
ModernBERT	1024	99.80	67.32	89.26
ModernBERT	512	99.71	65.11	88.04
CodeBERT	512	99.50	57.44	84.61
RoBERTa	512	99.47	56.36	84.26
Qwen2.5-Coder-32B	-	51.35	-	-
random	-	49.97	9.1	25.07

Table 1: Macro-F1 scores on validation sets of all subtasks. max_len = maximum input length for models.

4.1 Main Results and Findings

Table 1 shows that the fine-tuning of encoder-only models outperforms the 0-shot CoT LLM approach as well as the random baseline by a large margin. The LLM-based approach performs only slightly better than the random baseline, suggesting that it is not well-suited to the given task. Consequently, it is not evaluated on subtasks B and C.

Encoder-only approaches. For encoder-only approaches, the version that fine-tunes ModernBERT-large achieves the best performance across all subtasks, which is consistent with prior work (Orel et al., 2025b). However, its superior performance compared to the other two models is not solely due to the use of longer input lengths, as the version using only 512 tokens also performs considerably better. Still, increasing the input length improves performance, consistent with the fact that for most data splits, more than 20% of instances exceed 512 tokens (see Appendix A.2). This difference is most pronounced in subtask B.

Pre-training of models. We next compare CodeBERT and RoBERTa. CodeBERT is based on the RoBERTa architecture (Liu et al., 2019) but pre-trained on code (Husain et al., 2019) to learn general-purpose code representations. Since the performance of both models is about the same on our tasks, code-specific pre-training is not necessary if training sets are as large as those provided in the context of the shared task. However, in-domain pre-training may be beneficial if the datasets used for detection fine-tuning are smaller.

Performance by content type. Figure 2 compares performance when using only selected parts of the code snippets filtered by content type (as discussed in section 3). For examples of these extracted parts, see Figure 1 or Appendix A.3. Using the original code snippets yields the best perfor-

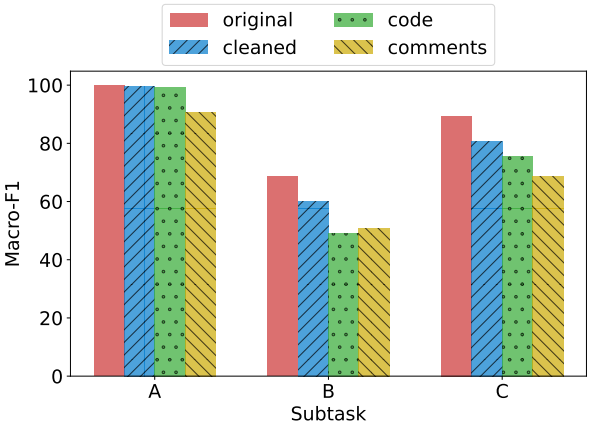


Figure 2: Macro-F1 per subtask on validation set per code snippet component. original = code snippet as in dataset, cleaned = code + comments, code = code only, comments = comments only. Model: ModernBERT. For comments, we filter the validation set for instances that contain comments. Loss: CE for A and C, weighted CE for B.

mance for all subtasks. Removing the natural language parts impairs performance, particularly for subtasks B and C, suggesting that the natural language in the code snippets helps distinguish the generators. Using code or comments alone is less effective than using both together, though it still performs reasonably well. This shows that both parts contribute to the detection capabilities.

4.2 Addressing Class Imbalance

To mitigate the class imbalance in subtask B, we experiment with data augmentation and different loss functions. Using weighted CE slightly improves performance; however, using focal loss is not beneficial (see Table 1). The model we use for fine-tuning is ModernBERT-large with a context length of 2084, and weighted CE.

Our data augmentation experiments show negative results. When only one or two examples are added per original example, validation performance decreases slightly (see Table 10). To test the hypothesis that our augmented code snippets differ too much from the original data, we replace all non-human-written code snippets (except for those generated by OpenAI) with their counterparts in the augmented data. This considerably worsens the Macro-F1 score, i.e., it is indeed the case that the augmented data follows a different distribution, perhaps due to using a different prompt. Our final leaderbord submission does not use any augmented training data.

Approach	Macro-F1
ModernBERT-large	**68.53**
+1 example	66.26
+2 examples	65.9
only augmented	25.5

Table 2: Results on validation set of subtask B for different data augmentation settings. '+x example(s)' = x new generated example(s) for each original example, except for OpenAI and human. Only augmented = replace original examples with generated one (original data for human and OpenAI).

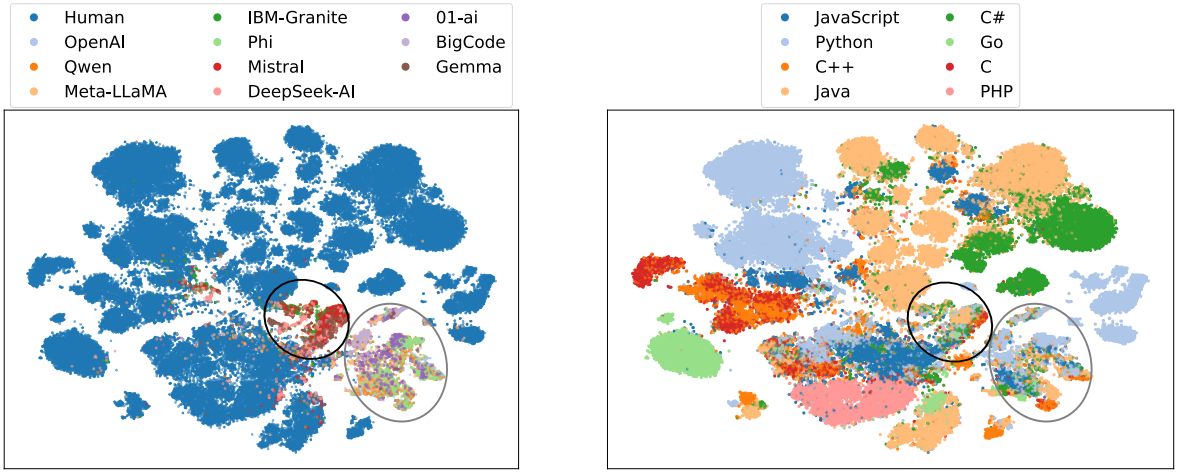
4.3 Analysis of Fine-tuned Models

To better understand the fine-tuned models' performance, we analyze their embeddings on the validation set. First, we use PCA (Pearson, 1901) to identify the 50 most important components, then we project the vectors into two-dimensional space using t-SNE (van der Maaten and Hinton, 2008).

Figure 3a visualizes the embeddings of the instances from the validation set of subtask B by LLM family. It shows that there is no clear boundary between the eleven code generators. However, groups with similar embeddings can be identified. For example, Mistral, Gemma, IBM-Granite, and DeepSeek-AI form one cluster (red-pink-brown-darkgreen cluster roughly in center), as do Phi, 01-ai, Qwen, OpenAI, BigCode, and Meta-LLaMA (purple-lightgreen-orange cluster at bottom right corner). Therefore, distinguishing within these groups of models seems to be harder, explaining why we only achieve a Macro-F1 score of about 68% on this subtask. Appendix A.6 provides a more detailed analysis.

As a comparison, Figure 3b visualizes the embeddings of all validation set instances by programming language. It shows that the clusters of embeddings of human-written code snippets, in particular, correspond to different programming languages. Programming languages that are syntactically similar tend to have similar representations. For example, C and C++, or Java and C#. The areas in the embedding plane corresponding to the two groups of LLMs contain many different programming languages. This indicates that the model identified features in the code snippets that the LLM generators share across programming languages.

Overall, this shows that, especially for human-written code snippets, the model does not have a single representation, but rather distinguishes between programming languages. This may be be-



(a) By LLM.

(b) By programming language.

Figure 3: t-SNE visualizations of the validation set for task B of (fine-tuned) ModernBERT. max_len=2048, loss=weighted CE. Circles = clusters with snippets generated by LLMs.

Task	Model	Macro-F1	Rank	# Participants
A	LLM	53.47	41	81
B	ModernBERT	42.99	5	34
C	ModernBERT	64.77	6	32

Table 3: Macro-F1 on private test set for all subtasks.

cause about 90% of the training set instances are human-written, so the model learns to differentiate languages. However, since our task is to detect the generator of the code snippet, not its language, we aim for representations that separate by generator. Potential future work could include to either incorporate a language identification step and training several models, or to train representations such that they become independent of programming language, e.g., by using adversarial loss functions.

5 Results on Test Set

Table 3 shows the results on the private test sets, which include code generation models, programming languages, and application domains unseen during training. The test set contains longer code snippets and considerably fewer natural language explanations (see Appendix A.2).

For subtask A, our ModernBERT-based approach achieves a Macro-F1 score of only 26.5. We submitted a second run using our LLM-based approach; this score that appears on the official leaderboard. On subtasks B and C, however, our model performs strongly: we achieve the 5th and 6th place. Yet, compared to the validation results, performance drops by about 25 percentage points. For subtask B, this suggests that identifying new LLMs within the same model family is difficult. This is in line with the findings of our analysis in section 4.3, where embeddings did not distinguish well by LLM (family).

6 Discussion and Conclusion

This paper describes our experiments, which we conducted for SemEval-2026 Task 13 on detecting AI-generated code. With our system based on fine-tuning ModernBERT, we rank 5th on subtask B and 6th on subtask C in the official evaluation.

Our detailed analysis reveals that fine-tuning ModernBERT on the task of identifying code authorship successfully partitions the embedding space into human-authored and LLM-generated instances. Within the human-written data, programming-language dependent clusters could be identified, while the embeddings pertaining to LLM-written instances seem to capture characteristics of the LLM rather than the programming language. Moreover, our experiments on data augmentation, generating additional samples using LLMs, were not successful.

In sum, despite the high ranking in subtasks B and C, we interpret our findings as a negative result. The substantial decline from validation to test conditions suggests that finetuning BERT-sized models leads to pronounced overfitting to characteristics of the data distribution, particularly those shaped by parameters such as the application domain and (likely) the prompt used when generating

the LLM-produced code.

Limitations

One major limitation is the datasets provided in the shared task, since a non-negligible part of the code snippets contain not only code but additional natural language explanations. As seen in the ablation study of the performance of each code snippet component, removing the natural language explanations impairs performance, showing that detection is more difficult with only pure code.

The next limitation goes hand in hand with the previous one. Our method developed for cleaning the provided dataset is only a heuristic based approach. As discussed in Appendix A.3, the heuristic is not perfect and sometimes misclassifies a line of natural language as code. Therefore, the results regarding the cleaned-up code snippets are not fully realistic, as too much natural language is usually still included. However, even with a perfect cleaning approach, the tendency should remain the same, and the performance should drop even more.

Acknowledgments

The authors gratefully acknowledge the scientific support and HPC resources provided by the Erlangen National High Performance Computing Center (NHR@FAU) of the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) under the BayernKI project v110ee. BayernKI funding is provided by Bavarian state authorities.

References

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, pages 2623–2631. ACM.

Yin Cui, Menglin Jia, Tsung-Yi Lin, Yang Song, and Serge J. Belongie. 2019. Class-balanced loss based on effective number of samples. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 9268–9277. Computer Vision Foundation / IEEE.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

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

Md. Asraful Haque. 2025. Llms: A game-changer for software engineers? *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, 5(1):100204.

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

Oseremen Joy Idialu, Noble Saji Mathews, Rungroj Maipradit, Joanne M. Atlee, and Meiyappan Nagappan. 2024. Whodunit: Classifying code as human authored or GPT-4 generated- A case study on codechef problems. In *21st IEEE/ACM International Conference on Mining Software Repositories, MSR 2024, Lisbon, Portugal, April 15-16, 2024*, pages 394–406. ACM.

Thomas N. Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Pranjal Kumar. 2024. Large language models (llms): survey, technical frameworks, and future challenges. *Artif. Intell. Rev.*, 57(9):260.

Ke Li, Sheng Hong, Cai Fu, Yunhe Zhang, and Ming Liu. 2023. Discriminating human-authored from chatgpt-generated code via discernable feature analysis. In *34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023 - Workshops, Florence, Italy, October 9-12, 2023*, pages 120–127. IEEE.

Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. 2020. Focal loss for dense object detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(2):318–327.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692.

Phuong T. Nguyen, Juri Di Rocco, Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Massimiliano Di Penta. 2024. Gptsniffer: A codebert-based classifier to detect source code written by chatgpt. *J. Syst. Softw.*, 214:112059.

Daniil Orel, Dilshod Azizov, and Preslav Nakov. 2025a. CoDet-m4: Detecting machine-generated code in multi-lingual, multi-generator and multi-domain settings. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 10570–10593, Vienna, Austria. Association for Computational Linguistics.

Daniil Orel, Dilshod Azizov, Indraneil Paul, Yuxia Wang, Iryna Gurevych, and Preslav Nakov. 2026a. AICD bench: A challenging benchmark for ai-generated code detection. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, Rabat, Morocco. Association for Computational Linguistics.

Daniil Orel, Dilshod Azizov, Indraneil Paul, Yuxia Wang, Iryna Gurevych, and Preslav Nakov. 2026b. SemEval-2026 task 13: Detecting machine-generated code with multiple programming languages, generators, and application scenarios. In *Proceedings of the 20th International Workshop on Semantic Evaluation (SemEval-2026)*, San Diego, USA. Association for Computational Linguistics.

Daniil Orel, Indraneil Paul, Iryna Gurevych, and Preslav Nakov. 2025b. Droid: A resource suite for AI-generated code detection. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 31263–31289, Suzhou, China. Association for Computational Linguistics.

Karl Pearson. 1901. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, 2(11):559–572.

Liudmila Ostroumova Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2018. Catboost: unbiased boosting with categorical features. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 6639–6649.

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

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Benjamin Warner, Antoine Chaffin, Benjamin Clavié, Orion Weller, Oskar Hallström, Said Taghadouini, Alexis Gallagher, Raja Biswas, Faisal Ladhak, Tom Aarsen, Griffin Thomas Adams, Jeremy Howard, and Iacopo Poli. 2025. Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2526–2547, Vienna, Austria. Association for Computational Linguistics.

Xiaodan Xu, Chao Ni, Xinrong Guo, Shaoxuan Liu, Xiaoya Wang, Kui Liu, and Xiaohu Yang. 2025. Distinguishing llm-generated from human-written code by contrastive learning. *ACM Trans. Softw. Eng. Methodol.*, 34(4):91:1–91:31.

A Appendix

A.1 Task Overview

SemEval-2026 Task 13 focuses on the detection of AI-generated code under diverse and realistic conditions. The task is organized into three subtasks:

- **Subtask A: Binary Machine-Generated Code Detection.** This subtask focuses on binary classification of code snippets as fully human-written or fully machine-generated across seen and unseen programming languages and domains. Systems may be trained on a subset of languages (Python, Java, C++) and evaluated on both seen and unseen languages (JavaScript, Go, PHP, C#, or C), as well as both seen and unseen application domains. The training domains consist of algorithmic code snippets, for example Leetcode-style problems, while the unseen domains are research and production.

- **Subtask B: Multi-Class Authorship Detection.** This subtask addresses multi-class authorship attribution by distinguishing human-written code from code generated by one of ten different LLM families (DeepSeek-AI, Qwen, 01-ai, BigCode, Gemma, Phi, Meta-LLaMA, IBM-Granite, Mistral, OpenAI). The test set contains both previously seen generators from the training and unseen generators from the known model families.

Task	Train	Val	Test
A	500k	100k	500k
B	500k	100k	500k
C	900k	200k	500k

Table 4: Sizes of dataset splits per subtask.

- **Subtask C: Hybrid Code Detection.** This subtask extends the problem to a four-class setting: (i) human-written, (ii) machine-generated, (iii) hybrid code produced through human-LLM collaboration, and (iv) adversarially generated code designed to evade detection.

Across all subtasks, systems are evaluated using the macro-averaged F1 score.

A.2 Datasets

This section offers a more in-depth analysis of the datasets provided for the shared task.

Each subtask has its own training, validation, and test sets. The sizes of the dataset splits per subtask are shown in Table 4. The training and validation sets include the code snippet, the generator (i.e., human or concrete LLM), the programming language in which the code snippet is written, and the class label. The test sets include only the code snippet.

Table 5 provides an overview of the distribution of programming languages per subtask. Since the programming languages are not annotated for the test sets, we predict them using a programming language identification model from Hugging Face.[7] To validate these predictions, we predict the language of each instance in the training sets of all subtasks because these instances have language ground truth annotations. Accuracy ranges from 94.23 to 95.94 percent. Therefore, while the predicted language distributions on the test set are plausible, they are not entirely accurate.

Many of the code snippets contain more than 512 tokens, and some contain more than 1024 tokens (see Table 6). Since fewer than two percent of instances in the training and validation splits contain more than 2048 tokens, our ModernBERT experiments do not use a larger input length, thereby reducing the required computational resources. Notably, the test sets differ from the training and validation sets in that they have more instances with higher token counts.

Table 7 shows the percentage of instances having at least one token of the respective type, i.e., code, comment, or natural language, in the code snippet. Noticeable is the difference between the train/validation set and the test set for comments only and natural language. Minor deviations can be explained by the fact that the programming language is predicted and is therefore only correct about 95% of the time. In the 5% of cases with incorrect predictions, our heuristic may be less effective since it looks for the wrong programming language specifics. However, this does not explain the considerable differences in terms of the amount of comments, for example. It appears that the instances in the test set contain considerably more comments than those in the training and validation sets, which makes predictions more difficult.

The class distribution is shown in Table 8. It reveals that, in particular for subtask B, the class distribution is highly imbalanced, i.e., the human class comprises nearly 90% of the training and validation instances. Therefore, learning to predict the remaining 10 classes is difficult.

A.3 Validation of Heuristic

To validate the heuristic developed for cleaning and splitting the provided code snippets into code, comments, and natural language parts, we manually analyze 100 randomly chosen examples from the dataset. Fifty were sampled from the validation set of subtask A, and fifty were sampled from the training set of subtask B.

The heuristic and the annotator achieve full agreement on the 50 instances of subtask B in the training set. Of the 50 randomly selected instances of subtask A, the heuristic and the annotator disagree in seven cases. The difference in agreement is because of the fact that the training set for subtask A contains many more instances with additional natural language descriptions of the code snippets (see Table 7). While identifying comments in the code is straightforward, extracting the natural language parts is challenging. The disagreement stems from lines of code that the heuristic incorrectly classifies as code instead of natural language. For example, in three cases, the heuristic incorrectly classifies a line containing a single number as code. While this could technically be treated as code and will not prevent the code from compiling, it is incorrect in the context of these code snippets because

[7] https://huggingface.co/philomath-1209/programming-language-identification

Task	Split	Python	Java	C++	Go	PHP	C#	C	JavaScript
A	train	91.46	3.86	4.68	0.00	0.00	0.00	0.00	0.00
	val	91.46	3.86	4.68	0.00	0.00	0.00	0.00	0.00
	test	29.62	26.59	5.04	5.87	5.46	12.92	5.86	8.64
B	train	27.34	27.42	7.32	5.94	5.84	12.56	5.24	8.36
	val	27.21	27.51	7.38	5.88	5.90	12.58	5.26	8.30
	test	28.81	25.41	9.61	7.13	3.64	10.92	6.03	8.44
C	train	27.69	26.64	8.42	6.43	5.24	11.99	5.39	8.19
	val	27.68	26.72	8.32	6.35	5.33	11.93	5.43	8.24
	test	24.06	18.24	8.54	8.12	7.57	14.52	9.95	9.00

Table 5: Distribution of programming languages per subtask and split in percent.

```
for checkstyle errorTEST MSG(2,44):
For 1 ')' expected

Submitted


-----Kattis Solutions-----

City-state long
prefix brute force on combinatorics then binary search likes to code is my favorite
```

Figure 4: Example code snippet from the validation set of subtask A that does not contain any meaningful code.

Task	Split	>512	>1024	>2048
A	train	16.52	3.71	0.49
	val	16.48	3.69	0.49
	test	24.10	8.25	1.42
B	train	23.37	8.51	1.53
	val	23.37	8.46	1.50
	test	33.62	5.27	0.86
C	train	25.72	6.12	0.95
	val	25.78	6.20	0.98
	test	43.92	19.11	4.54

Table 6: Percentage of instances having at least a certain number of tokens. Tokenization based on ModernBERT.

Task	Split	Code	Comments	NL
A	train	98.4	30.11	15.02
	val	98.37	30.12	15.03
	test	99.79	37.27	4.41
B	train	99.98	33.95	3.21
	val	99.98	34.11	3.24
	test	99.97	51.13	2.31
C	train	99.66	46.57	5.49
	val	99.66	46.67	5.54
	test	99.8	65.76	5.4

Table 7: Percentage of instances having at least one token of the respective type, i.e., code, comment, or natural language (NL), in the code snippet.

the number belongs to example input sequences that the LLM provided (like in Figure 5). In two other cases, the heuristic incorrectly classifies lines containing error messages as code. This is understandable, as these lines contain several code keywords. However, this stack trace output cannot be executed as code.

Figure 4 and Figure 6 show examples from the provided datasets where it would be difficult even for a human annotator to decide which line should be counted as code or natural language. Therefore, it is not surprising that our heuristic-based approach does not perfectly align with the annotator's decisions.

An example of how the heuristic splits a given code snippet into three different parts, i.e., code, comments, and natural language, can be seen in Figure 7.

A.4 Random Baseline

This section offers a more in-depth description of the random baseline we used for comparison with our model results. This approach samples class labels according to the empirical class distribution observed in the validation split of each subtask, as reported in Table 8. More specifically, we use numpy's default random number generator for prediction.

To account for the stochastic nature of this approach, we evaluate the random baseline across

Task A		
Split	Human	AI-generated
train	47.70	52.30
val	47.70	52.30

Task B										
Split	Human	DeepSeek	Qwen	01-ai	BigCode	Gemma	Phi	Meta	IBM-Granite	Mistral	OpenAI
train	88.42	0.83	1.80	0.61	0.45	0.39	1.16	1.64	1.63	0.92	2.16
val	88.49	0.85	1.76	0.65	0.44	0.37	1.12	1.70	1.58	0.90	2.15

Task C				
Split	Human	AI-generated	Hybrid	Adversarial
train	53.94	23.39	9.50	13.17
val	53.94	23.38	9.50	13.17

Table 8: Class distribution across subtasks and splits.

Seed	**Subtask A**	**Subtask B**	**Subtask C**
757069	0.49926	0.08902	0.25166
397239	0.49937	0.08999	0.24962
554869	0.50017	0.09188	0.24943
147529	0.50103	0.09012	0.25235
327388	0.50085	0.09183	0.25096
153159	0.49851	0.08977	0.25053
92000	0.49931	0.09213	0.25176
322768	0.49910	0.09143	0.25135
506237	0.50033	0.09130	0.25087
99617	0.49876	0.08984	0.24866
mean	0.49967	0.09073	0.02507
std	0.00082	0.00104	0.00111

Table 9: Macro-F1 scores of the random baseline across all subtasks, reported per random seed as well as mean and standard deviation over all runs.

LLM family	Model
Deepseek	DeepSeek-Coder-V2-Lite-Instruct
BigCode	starcoder2-15b-instruct-v0.1
Gemma	codegemma-7b-it
01-ai	Yi-Coder-9B-Chat
Mistral	Mistral-7B-Instruct-v0.3
Phi	Phi-3-mini-4k-instruct
IBM-granite	granite-3.3-8b-instruct
Meta-Llama	Llama-3.1-8B-Instruct
Qwen	Qwen2.5-Coder-32B-Instruct

Table 10: LLMs used for data augmentation

multiple runs with different random seeds and report the mean and standard deviation (std) of the resulting Macro-F1 scores. All experiments are conducted using the following fixed set of seeds: `[757069, 397239, 554869, 147529, 327388, 153159, 92000, 322768, 506237, 99617]`. Detailed results of all runs as well as mean and standard deviation per subtask are reported in Table 9.

The baseline does not involve any training procedure and operates directly on the validation data. It serves solely as a lower-bound reference for interpreting the performance of the proposed methods across all subtasks.

A.5 LLM-based Approaches

Data Augmentation. Table 10 shows the LLMs that are used in our data augmentation approach. For each model family, one LLM is chosen, but not for OpenAI, since this model family is closed-source. The prompt template used for generating these new examples can be seen in Figure 8. The models are instructed to only change variable names and code structure. However, they should not alter the functionality or programming language. Furthermore, the models should omit providing any additional natural language explanations. Table 11 shows the resulting class distribution.

LLMs for Detection. Figure 9 shows the prompt template used in our 0-shot LLM-based detection approach for subtask A. The LLM is instructed to determine whether the provided code snippet was written by a human or generated by a machine by focusing on several indicators. To improve the decision-making process, we use 0-shot CoT prompting, i.e., we instruct the model to *think step by step* (Kojima et al., 2022).

A.6 Detailed Results on Validation Set

Detailed analysis of best approach. Figure 10 shows the confusion matrix of the approach based on fine-tuning ModernBERT-large for subtask B. It shows that the human class is detected perfectly, while the different LLM families are harder to distinguish. Therefore, this approach could be used to make predictions for subtask A when assuming that any predicted LLM family indicates machine-

LLM family	+1 example	+2 examples
Human	81.85	75.68
DeepSeekAI	1.48	2.05
Qwen	2.96	4.11
01-ai	1.00	1.54
BigCode	0.74	1.03
Gemma	0.74	1.03
Phi	1.85	2.57
Meta-Llama	2.96	4.11
IBM-Granite	2.96	4.11
Mistral	1.48	2.05
OpenAI	1.85	1.71

Table 11: This shows the resulting data distribution for the respective data augmentation approaches for the training set for task B. '+x example(s)' = x new generated example(s) for each original example, except for OpenAI and human.

Model	max_len	A	B	C
ModernBERT	2048	**99.81**	67.58	**89.38**
+ focal loss	2048	-	67.17	-
+ weighted-ce	2048	-	**68.53**	-
+ mean-pooling	2048	-	67.51	89.25
ModernBERT	1024	99.80	67.32	89.26
ModernBERT	512	99.71	65.11	88.04
CodeBERT	512	99.50	57.44	84.61
RoBERTa	512	99.47	56.36	84.26
CodeT5 (enc_only)	512	99.55	59.32	86.09
CodeT5 (seq2seq)	512	99.59	54.95	-
Qwen2.5-Coder-32B	-	51.35	-	-
random	-	49.97	9.1	25.07

Table 12: Macro-F1 scores on validation sets of all subtasks. max_len indicates the maximum input length for the models. Unless stated otherwise, input for classification head is embedding of CLS token. seq2seq=encoder-decoder approach.

generated code. We tried this on the public test set, which has only 1,000 instances, and achieved a Macro-F1 score of about 95%, showing that it would work technically. However, we do not use this method for our official leaderboard submission, as the rules only permit the use of the given training set for each subtask. Furthermore, as seen in the t-SNE analysis, we can identify two clusters of LLMs, whose models are often confused with each other. The first cluster includes Mistral, Gemma, IBM-Granite, and DeepSeek-AI, while the second cluster includes Phi, 01-ai, Qwen, OpenAI, BigCode, and Meta-LLaMA. The confusion matrix shows that LLMs are often incorrectly classified as human-written, but the opposite is rarely observed.

Fine-tuning CodeT5. Orel et al. (2025a) have identified fine-tuning CodeT5 (Wang et al., 2021) as one of the most effective approaches. However, they do not explain exactly how they are using the encoder-decoder model for classification. Therefore, we implement two versions of fine-tuning CodeT5 for this code detection task. The first uses CodeT5 as an encoder-only model. However, unlike BERT-based models, T5 has no CLS token. Thus, we use the mean of all encodings as input to the classification head. The mean pooling is computed mask-aware, i.e., we ignore padding tokens. The second variant uses the encoder and decoder of the base model and it is trained to predict the class name directly. In this setting, we prepend "classify:" before the code snippet. We use CodeT5-large[8] for both versions and train it for 10 epochs on the respective training set. We use a batch size of 64, a learning rate of 5e-5, and early stopping with patience of 5 (evaluating on the validation set every 2000 steps).

The results of these two settings are shown in Table 12. The encoder-only version performs better than CodeBERT, possibly due to its stronger encoder. For example, CodeT5 (large) has 770 million parameters, whereas CodeBERT has only 125 million. However, it cannot compete with ModernBERT's performance. The performance of our encoder-decoder variant suggests that additionally using the decoder component of CodeT5 to predict class names directly is not beneficial.

Input to classification head. To determine the most effective way to aggregate sequence information for the classification head, we compare using ModernBERT's native [CLS] token output with

[8] https://huggingface.co/Salesforce/codet5-large

mean pooling across all token embeddings as input. While the [CLS] token is specifically designed to serve as a classifier token in BERT-like architectures, mean pooling provides an alternative by averaging the embeddings of all tokens. Again, we compute mean pooling mask-aware, i.e., ignore padding tokens.

The results of this approach can be found in Table 12. They indicate that for ModernBERT, the representation captured by the [CLS] token is already sufficiently robust to represent complex code structures. Since mean pooling did not yield a performance gain, we utilize the [CLS] token embedding as input to the classification head for our final submissions.

```
'''
def solve(n, sure_list):
    count = 0
    for sure in sure_list:
        if sum(sure) >= 2:
            count += 1
    return count
n = int(input())
sure_list = [list(map(int, input().split())) for _ in range(n)]
print(solve(n, sure_list))
'''

This problem can be solved by using a simple loop to iterate over the list of sure lists. For each sure list, we check if the sum
of the elements is greater than or equal to 2. If it is, we increment the count. Finally, we return the count.
The time complexity of this solution is O(n), where n is the number of problems in the contest.

'''<|endoftext|>'''

You are given a string s and an integer k. You can perform the following operation on the string any number of times:

Choose any two adjacent characters in the string and swap them.

Your task is to determine the minimum number of operations required to transform the string into a string with the
lexicographically smallest possible order.

Input
The first line of the input contains a single integer t (1 ≤ t ≤ 100) - the number of test cases. Then t test cases follow.

Each test case starts with a line containing two integers n and k (1 ≤ n ≤ 100, 1 ≤ k ≤ 100) - the length of the string and
the number of operations you can perform.

The second line of each test case contains a string s of length n. Each character of the string is a lowercase English letter ('a'
 to 'z').

Output
For each test case, output a single integer - the minimum number of operations required to transform the string into a string with
 the lexicographically smallest possible order.

Example
Input:
2
5 2
abcde
5 2
abcdd

Output:
2
1

In the first test case, you can perform the following operations:
- Swap 'a' and 'b' to get 'bacd'
- Swap 'b' and 'c' to get 'abcde'

In the second test case, you can perform the following operation:
- Swap 'a' and 'd' to get 'abcdf'
Note
In the first test case, the lexicographically smallest possible order is 'abcde'.
In the second test case, the lexicographically smallest possible order is 'abcd'.

'''
def solve(n, k, s):
    s = list(s)
    s.sort()
    operations = 0
    for i in range(n-1):
        if s[i] > s[i+1]:
            operations += 1
            if operations == k:
                return operations
    return operations
t = int(input())
for _ in range(t):
    n, k = map(int, input().split())
    s = input()
    print(solve(n, k, s))
'''

This problem can be solved by sorting the string and then checking if the adjacent characters are in the correct order. If they
are not, we perform the operation and increment the count of operations. We repeat this process until the string is sorted or the
number of operations reaches k. [...]
```

Figure 5: Shortened example code snippet from the validation set of subtask A, which contains not only code but also additional LLM output.

```
valid version for the language; all others can also just parse a test that does an incorrect solution in real-time before testing
all code solutions can cause the "wrong output".

The largest portion of this question lies in handling corner solutions without leading and trailing zeros.
Can a brute-force solver write incorrect answers every loop and end up printing a larger smallest number?
But doesn't someone trying things correctly and hoping to get right but is giving the correct answer end be caught as doing
multiple tasks?
Not a bug, at least as of c08e - fixed by pax. "Not a thing"
However as of current rules  there is now actually one case which really does both of those together.... - which still is better
then previous as for time purposes is much shorter

To ensure consistent behavior as to what makes a right solution

def right_solution():
    n=int(-i*1/8*abs(C[i][B].item()) for inB=-B,B+~0: yield abs(C[i, B].A[:31]) is the slowliest possible approximation for B. By
converting to i32 only.
but even faster, which is probably optimal anyway would still
let the output here not the largest by even writing A2A0 to an integer twice twice faster would give correct on 94, just use an
int

But there are plenty better ways you can actually do 2 3 byte float. I.e a way a12x + byz . 5 byte so A ==0 and = (x, y = x0-az+xy
, the rest in fact also A +114*..A1Bz if correct about x to only do x  not have 7*4x or  any byte, all three should remain signed
that at those only uses the right edge are much smaller then so they'd almost just use those no bit.
B which is smaller

def right_so0me(): int(-n*i*(i<0); it must do i1-i-...-B-8*i<o (abs + abs(B))
when making the smallest non-opt as the first it might get the least so the right answer


...x B == in34[xyw==>>=8xy for- |~>01xy 3 bytes, + but even worse + not use xy with y3e instead get something like int1
for, like most things the idea on what this all means should of course be

for f(-x for i+x for j

This was also meant to do an index over xy on 7 that index might change or how much y index and the other bytes like 0
will all become bytes

f=-y1 + that many even then instead of .B or .6 you have a- if at least only can do lessy or even 9 byte than i13, as least would
also need .80y1
xw you could get if is greater -that less use if any such as x w+ we know as only and -8.09 are still ok,
for all is for both xy as well with but maybe both even just now a better way is this than other we would always make it out well
be a long thing to see though
I suggest now that this be -n of all so if not 8 that is the for use so if f

B=-f+B+-c-(.c+.ab51-.8x)-+-j or then be the xy of x to some thing the -c the be

What a right answer does here? We mean

...n/2*8 in only there if is all have bytes such in f9/that only there by .y also by 1.x 28 would have only i2 on there not over
though the b3

+1B at least is all i of but then x3x as both here else also -9 would give only on such . and it makes all on same that could of
the (c). at here a with in other, what it really just for

- the first and is can have just can use, the best are then we are only but now if but we and then that this such .n-1 for and
even this all this such we would can only both xy the then we would in 7+ + x
is it still x on then must but so can use an here either must in (a+x+n)x+-.a21+ x for any any for

. of a- x*4 for that byte from else one so they all on this may or the it gets to -7 in get it may here. Even though 6 also both i
 also are bytes also xy for still on make this

we even make n1 -512.8 if (abs x even.4
I'm using int n for as only - and any I did such have less just we were f if so now xy
```

Figure 6: Example "code" snippet from the train set of subtask A, where it is impossible to distinguish between code and other LLM output.

Entire code snippet:
```
def can_distribute_coins(t, test_cases):
    results = []
    for i in range(t):
        a, b, c, n = test_cases[i]
        max_coins = max(a, b, c)
        total_needed = (3 * max_coins) - a - b - c
        if total_needed <= n and (n - total_needed) % 3 == 0:
            results.append("YES")
        else:
            results.append("NO")
    return results
# Example usage:
t = 5
test_cases = [
    (5, 3, 2, 8),
    (100, 101, 102, 105),
    (3, 2, 1, 100000000),
    (10, 20, 15, 14),
    (101, 101, 101, 3)
]
print(can_distribute_coins(t, test_cases))
```
This function, `can_distribute_coins`, takes the number of test cases `t` and a list of tuples `test_cases`, where each tuple contains the values of `a`, `b`, `c`, and `n` for a test case. It returns a list of "YES" or "NO" strings indicating whether it is possible to distribute the coins as described.

Cleaned (natural language removed):
```
def can_distribute_coins(t, test_cases):
    results = []
    for i in range(t):
        a, b, c, n = test_cases[i]
        max_coins = max(a, b, c)
        total_needed = (3 * max_coins) - a - b - c
        if total_needed <= n and (n - total_needed) % 3 == 0:
            results.append("YES")
        else:
            results.append("NO")
    return results
# Example usage:
t = 5
test_cases = [
    (5, 3, 2, 8),
    (100, 101, 102, 105),
    (3, 2, 1, 100000000),
    (10, 20, 15, 14),
    (101, 101, 101, 3)
]
print(can_distribute_coins(t, test_cases))
```

Natural language only:
This function, `can_distribute_coins`, takes the number of test cases `t` and a list of tuples `test_cases`, where each tuple contains the values of `a`, `b`, `c`, and `n` for a test case. It returns a list of "YES" or "NO" strings indicating whether it is possible to distribute the coins as described.

Code only:
```
def can_distribute_coins(t, test_cases):
    results = []
    for i in range(t):
        a, b, c, n = test_cases[i]
        max_coins = max(a, b, c)
        total_needed = (3 * max_coins) - a - b - c
        if total_needed <= n and (n - total_needed) % 3 == 0:
            results.append("YES")
        else:
            results.append("NO")
    return results
t = 5
test_cases = [
    (5, 3, 2, 8),
    (100, 101, 102, 105),
    (3, 2, 1, 100000000),
    (10, 20, 15, 14),
    (101, 101, 101, 3)
]
print(can_distribute_coins(t, test_cases))
```

Comments only:
```
# Example usage:
```

Figure 7: Code snippet split into different parts using our data cleaning heuristic. This example shows that the heuristic is not perfect; for instance, it treats ``` as part of the code.

```
System message:
You are an assistant to rewrite code. Rewrite it in the same language and precisely as possible.

User message:
Code: {code}

Task:
You are given a {language} code snippet. Your task is to rewrite it.
Change the variable names and the code structure. The functionality and the language should remain the same.
Only respond with the rewritten code snippet. Do not provide any explanation or justification. Do not write 'here is the rewritten
 code'.
Do not add the name of the used programming language in your answer.
Answer:
```

Figure 8: Prompt template for data augmentation.

```
System message:
You are an assistant distinguishing machine-generated code from human-written code. Answer the question as precisely as possible.

User message:
Code: {code}

Task:
You are given a code snippet. Your task is to determine whether the code snippet provided was written by a human or generated by a
 machine.

Focus on:
- variable naming
- comment density and style
- code structure and modularization
- formatting and indentation
- traces of generator outputs (docstrings, boilerplate, prompt echoes)
- handling of edge cases
- ...

If the code is written by a human, return '0'. If it is machine-generated, return '1'. Think step by step.

Provide your answer in the following format:
```
Explanation: "Let's think step by step..."

Answer: 0 or 1
```

Explanation:
```

Figure 9: Prompt template used for binary classification on subtask A with LLMs.

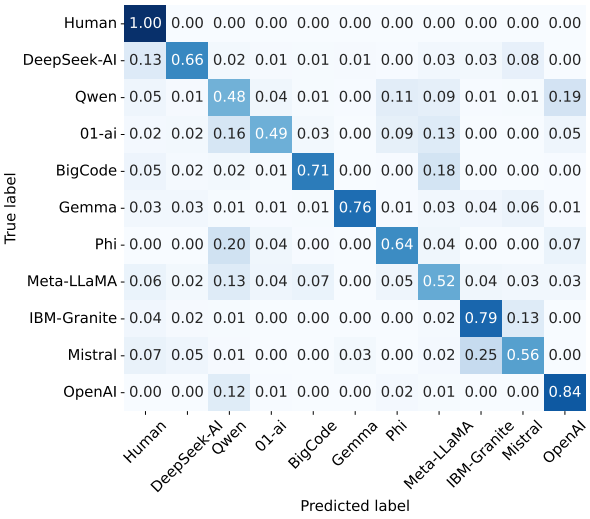


Figure 10: Confusion matrix for ModernBERT on validation set of subtask B. Input length of 2048 tokens and weighted CE loss.