COODetect at SemEval 2026 Task 13: Unsupervised Latent Domain Adaptation for Out-of-Distribution AI-Generated Code Detection

Aldan Creo[1], Atharv Nair[2], Mohana Kannan Ravikumar[2], Vaishak Menon[1], Dario Wisznewer[1], Vaibhav Jain[2]

[1]School of Computing, Information and Data Sciences,
[2]Jacobs School of Engineering
University of California, San Diego
Correspondence: research@acmc.fyi

Abstract

The widespread use of AI-generated code raises questions about software maintenance and academic integrity. However, tools to detect it are still in their infancy. In this article, we explore the issue of out-of-distribution (OOD) detection; while embedder models like CodeBERT can easily achieve high accuracies in the context of their training data, they are unable to properly generalize to unseen contexts or programming languages. We argue that this is caused by an overfitting of such models to the training distribution, e.g. memorizing a language's "AI syntax" instead of the true generative artifacts, and develop a approach that is able to naturally generalize to completely unseen languages and domains. Our system is also considerably more interpretable than the deep neural alternatives. In particular, we propose three orthogonal views (lexical, structural, and symbolic) to capture the AI-generated code's indicators. To deal with OOD shift, we normalize the scores per language with Z-scoring and a Gaussian Mixture Model to remove the language bias automatically. We test our approach on the SemEval-2026 Task 13 dataset, where our experiments reached a macro F1 of 0.602 compared to the task baseline of 0.305, demonstrating the generalization capabilities of our system. We make our source code and data available at https://github.com/ACMCMC/COODetect.

1 Introduction

Large language models are now part of the daily toolkit for developers, researchers, and students (Adamenko et al., 2025; Zadenoori et al., 2025; Bistarelli et al., 2025). They can write correct programs and act as coding assistants; however, they also elicit the question: can we tell if a code sample is human-written or AI-generated?

AI-generated code detection can be reliably executed with embedder models when the testing data is similar to the training data. However, these models tend to suffer from a lack of ability to generalize beyond their training context. We observe that they fall into a syntax trap where they confuse the language syntax with the generative artifacts. For instance, when training an embedder model in Python and Java to detect AI code, the model may learn that indentation variance is a feature of Python, and not necessarily a feature of human code. The attention mechanisms of these deep models focus in the specific token sequences of the training languages. This greatly impedes generalization; if we later apply this model to out-of-distribution languages like C# or Go, the model sees unfamiliar syntax and fails to generalize. The embeddings collapse, and the system predicts the labels with extreme but incorrect confidence. We call this a calibration collapse.

We develop a main strategy that fixes this problem. We build COODetect, an ensemble that decomposes the code into three orthogonal signals (lexical, structural, and symbolic). We do not predict the classification labels directly from the raw scores. Instead, we use a Bayesian Gaussian Mixture Model to find latent domains in the test set without needing any labels, and we normalize our predictions inside those specific domains. This means we compare a piece of code only to other similar pieces of code.

By participating in this task, on the one hand, we show that deep neural models are not the only avenue in state-of-the-art for this problem. A well-engineered feature pipeline can be more interpretable and generalizable than the other mechanisms and thus be preferrable in certain contexts. In fact, we beat the official embedder baselines by a large margin in the OOD regimes. On the other hand, we also observed how our system struggles heavily with very short code snippets because there is not enough text to extract statistical anomalies (though we expect that a neural architecture would

Proceedings of the The 20th International Workshop on Semantic Evaluation (2026), pages 1443–1450

struggle in similar ways). Our baseline experiments reached a macro F1 of 0.602 on the OOD test set, and subsequent feature expansions reached 0.785[1].

2 Background and Related Work

Our experiments use the data and the evaluation protocol of SemEval-2026 Task 13 (Orel et al., 2026b), specifically Subtask A ("Binary Machine-Generated Code Detection"). The task organizers released a training and validation set with algorithmic code in Python, Java, and C++. Each example is labeled as human-written (238K examples) or AI-generated (262K examples). The test set contains snippets in all these languages, and it also includes five out-of-distribution languages (C, C#, Go, PHP, and JavaScript). The test set also introduces unseen domains like production code and research scripts.

To deal with the challenge of the detection of AI-generated code, some avenues have already been proposed. Previous work on the creation of large datasets to evaluate detectors across different programming languages (Orel et al., 2026a; Demirok et al., 2025) has already shown that simple classification fails when the test data comes from a different distribution. Hu et al. (2022) explore this distribution shift in source code; they explain that the detectors from other fields do not transfer well for code.

Currently, the most popular approach is to use supervised embedder models like CodeBERT or to apply large language models directly. For example, Yang et al. (2023) propose a zero-shot method that adapts probability curvature to source code; Bulla et al. (2024) use token probabilities from language models to explain the classifications. However, these two families of models have different flaws. The supervised embedders work impressively well in distribution, but they do not generalize adequately to unseen languages. On the other hand, the zero-shot neural methods do not require training data, but they require a massive amount of computation to extract the token probabilities. This makes them too slow and expensive for practical use.

Because of this, we take ideas from code stylometry and feature extraction. Gurioli et al. (2024) use stylometry to find AI code across ten languages, while Idialu et al. (2024) extract similar features to classify submissions in programming contests. Suh et al. (2025) report that (1) natural language AI-generated text detectors don't transfer properly to AI-generated code detection, and (2) machine learning with static code metrics can outperform complex models. We follow this feature-based philosophy because it naturally generalizes to completely unseen languages, it is fast, and we do not need expensive GPUs.

[1]We do not report on those since we achieved them after the submission deadline.

3 System Overview

The official baseline, a CodeBERT model fine-tuned on the training data, can score near perfect on that setting. In our tests, we generated training and testing subsets by splitting the official shared task training set (all examples are thus in-distribution), which allowed us to achieve a macro F1 of 0.97 with early stopping. On a similar dataset, Orel et al. (2025) achieve an even higher score of 0.99 with an equivalent method. However, this approach collapses on the test set (0.305); it overfits to the seen languages and the short algorithmic style. This is a general phenomenon that also applies to families of models typically used in AI-generated text detection, which can achieve a decent 0.63 in-distribution, but degrade to 0.21 when testing with unseen domains and languages (Orel et al., 2026a). In contrast, we propose a system that avoids deep neural networks in favor of three interpretable "analysts", which we describe next, and is able to generalize much better to the OOD test set (0.602 macro F1).

3.1 The COODetect Architecture

We hypothesize that if one view fails because of a specific language syntax, the other views can compensate for the mistake, so we design three views to capture complementary signals. To prove the orthogonality of our views experimentally, we calculate the pairwise agreement percentage between the three models on the test set (Figure 2). If the three models agreed completely, the ensemble would be useless, but instead, we observe that there is a certain level of disagreement (e.g. structural–symbolic 78.2%) that can capture different signals and thus improve the overall performance. The three views, which we call "analysts" for intuition, are the following:

Lexical Analyst. Our hypothesis is that LLMs exhibit a "training accent", i.e., they have a specific distribution of vocabulary that persists regardless of the programming language. For example, they

```
for i in range(int(input())):
input()
a = input().split(' ')
for j in range(len(a)):
a[j] = int(a[j])
b = min(a)
c = []
d = []
for j in range(len(a)):
if a[j] % b == 0:
c.append(a[j])
d.append(j)
c = sorted(c)
d = enumerate(d)
for (j, k) in d:
a[k] = c[j]
d = 1
for j in range(len(a) - 1):
d = a[j] <= a[j + 1] and d
if d == 1:
print('YES')
else:
print('NO')
```

(a) Human: short lines and overall length, obscure variable names, no comments.

```
def min_cards_to_flip(s):
    vowels = {'a', 'e', 'i', 'o', 'u'}
    odd_digits = {'1', '3', '5', '7', '9'}

    # Collect all cards with vowels and all cards
     with odd digits
    cards_with_vowels = [i for i, char in enumerate
     (s) if char in vowels]
    cards_with_odd_digits = [i for i, char in
     enumerate(s) if char in odd_digits]

    # Calculate the minimum number of cards to flip
    if not cards_with_vowels:
    return 0

    cards_to_flip = set(cards_with_vowels) | set(
     cards_with_odd_digits)
    return len(cards_to_flip)

# Example usage:
print(min_cards_to_flip("ee")) # Output: 2
print(min_cards_to_flip("z")) # Output: 0
print(min_cards_to_flip("0ay1")) # Output: 2
```

(b) AI: more verbose and descriptive, predictable, includes comments and appropriate variable names.

Figure 1: Examples of algorithmic Python snippets from the training set.

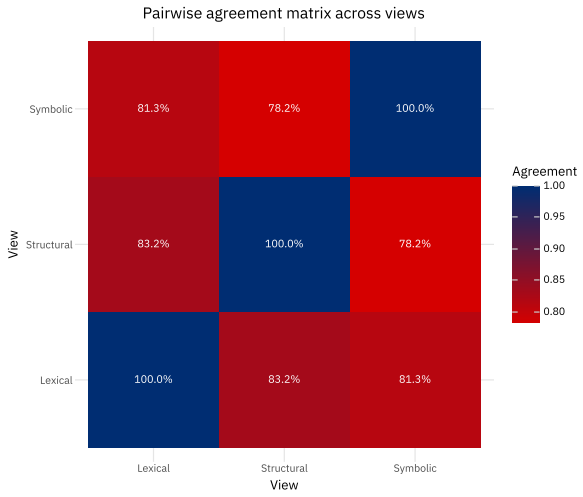


Figure 2: View disagreement matrix showing the pairwise agreement percentage between the three analysts.

might overuse words like “utilize” or “paramount” in the comments, or use very generic variable naming conventions that human programmers might be less prone to use. For instance, in non-production contexts, we expect humans to be “lazier” and use shorter or obscure names (“`res_1`”), inconsistent casing (“`myVar1`” and “`my_var_2`”), among others. We implement this with a Term Frequency-Inverse Document Frequency (TF-IDF) vectorization of word unigrams and bigrams. We use a minimum document frequency to remove rare human typos and then we train a logistic regression model. This analyst thus aims to capture the semantic intent and the word choice.

Structural Analyst. AI-generated code is blockier and more statistically regular than human code. Human code tends to be jagged and organic. We extract several invariant features to capture this visual rhythm. For the information density, we use the LZMA and BZ2 compression ratios (we expect AI code to be more compressible due to a more redundant style). For the entropy dynamics, we calculate the Shannon entropy over sliding windows of the file and measure the variance. For the visual rhythm, we measure the standard deviation of the indentation, the standard deviation of the line length, and the whitespace ratio. We pass these features to an XGBoost classifier that, generally speaking, aims to capture the “shape of the code”.

Symbolic Analyst. We hypothesize that the logical flow of the control structures follows distinct patterns in AI-generated code. For example, a sequence like `IF` followed by `CHECK` and `RETURN` is likely different in humans and LLMs, independently of variable names and similar features. We also expect that AI models have a tendency to write more boilerplate logic in a very predictable order. We thus use a preprocessor of regular expressions to replace all the identifiers with a generic token,

but we preserve the keywords and the operators. Then, we vectorize this abstracted string using a character-level hashing vectorizer with large n-grams. Finally, we apply a stochastic gradient descent classifier. Overall, this third analyst has the goal of capturing the abstract logic sequences.

3.2 Unsupervised Latent Domain Calibration

The core innovation of our system is how we combine these three views to generalize to new domains. Naïve averaging fails because the models are uncalibrated on the OOD data (e.g., the embedders output 0.99 probability for new languages just because the syntax is different).

To mitigate this "confidently wrong" problem, we discard the absolute probabilities entirely and we favor relative ranks in their place. We calculate the ensemble score by averaging the relative ranks of the three views to ensure that no single overconfident view dominates the decision.

However, it is clear that we cannot, for instance, compare the rank of a short Python script with the rank of a large C# enterprise class. To deal with this, we reject hard-coded rules for domain discovery because they are fragile and instead, we treat the domain discovery as a clustering problem. More specifically, we cluster the samples using the invariant structural features like the file length and the entropy. We use a Bayesian Gaussian Mixture Model (BGMM) with a Dirichlet Process prior that places a probability distribution over the partitions; this allows the model to infer the number of active components automatically and "turn off" the clusters that do not contain data.

Upon inference, the data evaluated (the OOD test set in our setup) is partitioned into latent domains automatically. Within each latent domain, we normalize the ensemble scores using a standard Z-score so that, for example, a high-entropy file is compared only to other high-entropy files (if it has a Z-score greater than 2.0, it is an anomaly relative to its structural peers, not relative to the entire dataset). Finally, to determine the decision boundary without labels, we fit a 2-component Gaussian Mixture Model to the global Z-scores using Expectation-Maximization. The optimal threshold is the intersection point between the human curve and the AI curve.

Also, from a practical perspective, we prioritize computational efficiency to save resources. Using massive embedder models for inference is expensive and slow, so we also choose to focus exclusively on the three domains described above because they are fast to compute and do not require expensive GPUs (which can be key for real-time integration in e.g. university grading portals or large-scale repository scans).

4 Experimental Setup

We use the SemEval-2026 Task 13 Subtask A dataset (Orel et al., 2026b), from which we extract the features using standard Python libraries and we abstain from using GPUs for the inference (as described above).

To validate our models, we use a "leave-one-language-out" strategy; we train on two languages and test on the third to simulate the OOD shift. We extract the features and train the models using standard libraries (`scikit-learn`, `xgboost`, and `gpytorch`). For the hyperparameters, the BGMM uses a weight concentration prior of $1e^{-2}$ with 30 maximum components. The hashing vectorizer uses 2^{18} features to prevent collisions, and the TF-IDF vectorizer is limited to 15,000 features. The XGBoost model uses 100 estimators and a learning rate of 0.1. We evaluate the performance using the Macro F1 score (the harmonic mean of the precision and the recall for both classes), also the official metric of the shared task.

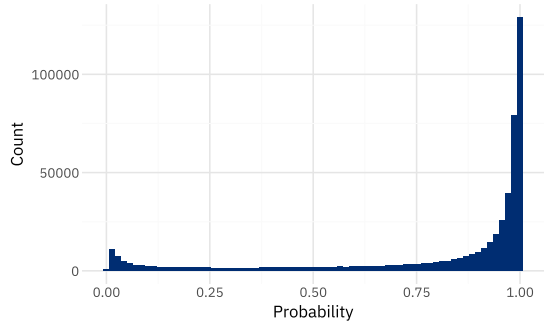
5 Results and Interpretation

Our official submission to the shared task reached a macro F1 of 0.602 on the OOD test set, which is a large improvement over the official CodeBERT baseline (0.305) and other methods described in Section 3 (0.21). We dedicate this section to analyzing and interpreting our results and how we generalize. We show that our approach is much more interpretable and stable than the embedder models. A detailed analysis of the data distribution is provided in Appendix A.

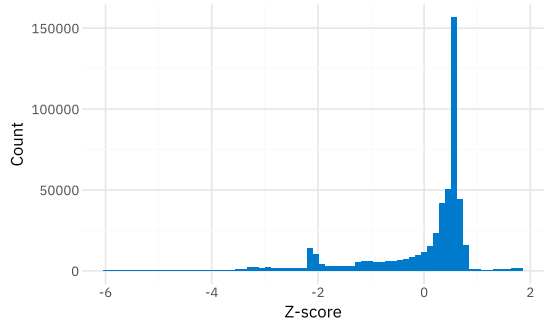
5.1 Generalization via Latent Domains

We measured the structural drift between the languages to provide numerical proof that models trained on specific syntax fail out of distribution. We observed that many structural features drift significantly between the training languages and the testing languages. For instance, the Kolmogorov-Smirnov shift statistic for the symbolic operator density is highly unstable ($KS \approx 0.46$). Conversely, features like the ratio of long lines remain stable across distributions ($KS \approx 0.04$).

In the raw structural probabilities, the score distribution is very dependent on the specific language we consider: the mean scores range from 0.432 (Go) to 0.965 (PHP), and several languages are heavily saturated in the high-confidence region (overall, 64.4% of test samples have score > 0.90). For instance, for C#, the raw mean is 0.798, with 55.3% of samples above 0.90, which indicates there is substantial, but not universal, overconfidence. After we normalize Z-scores per language, the first-order language bias is less prominent (per-language z-mean ≈ 0 and z-std ≈ 1), which helps in making the scores more comparable across languages before we apply a global thresholding step. Figure 3 illustrates this effect.



(a) Raw probabilities showing the calibration collapse: the model is confidently wrong (the probability mass is concentrated at 1).



(b) Z-score distributions after normalization. We standardize within each language, which enables us to have a global set of calibrated scores. We still see a mass concentration at the high end, but now the scores are less language-dependent and more comparable across languages.

Figure 3: Calibration collapse and the effect of Z-score normalization. Normalizing the scores per language is effective in making the distributions more comparable across languages, which is crucial for the global thresholding step afterwards.

The BGMM discovered approximately 10 active domains in the test set automatically. We analyzed these clusters qualitatively to understand what the model learns (see Table 2 in Appendix B). For example, one cluster is characterized by a low line count (14.6 average lines, algorithmic solutions), while another has deep indentation and high boilerplate (65.2 average lines, enterprise Java classes). By normalizing the scores inside these semantic clusters, we effectively condition the model on the drift and thus try to neutralize it.

5.2 Ablation of the Orthogonal Views

Table 1 shows the ablation study using the leave-one-language-out validation. We put in evidence that the individual views are weak under shift, but the ensemble recovers the performance.

Method	Python	Java	C++	F1
Lexical	0.346	0.493	0.456	0.425
Structural	0.356	0.749	0.728	0.613
COODetect	0.581	0.812	0.730	0.691

Table 1: Performance by strategy in the ablation study. The columns show macro F1 when that language is held out during training.

The structural view works well for compiled languages like Java and C++, but it fails completely in Python because of the syntax differences. COODetect uses the lexical and symbolic views to rescue the Python performance, so we believe that combining these orthogonal views is strictly better than optimizing a single view.

5.3 Interpreting the Features

Our approach is much more interpretable than the neural networks; for example, we can inspect the Logistic Regression weights and the XGBoost tree nodes directly. Figure 4 shows the feature importance for the structural view. The BZ2 compression ratio and the line length standard deviation are very important signals, which points to our hypothesis that AI code tends to be repetitive and statistically regular. In the symbolic view, tri-grams representing sequences of operators and blank spaces (like `OP-OP-BLK`) offer the highest information gain.

5.4 Qualitative Error Analysis

We inspected the false positives and the false negatives to understand how the system generalizes and where it fails.

The false positives are often human files with very rigid formatting: examples include auto-generated code or files passed through aggressive linters. These files have low entropy variance and

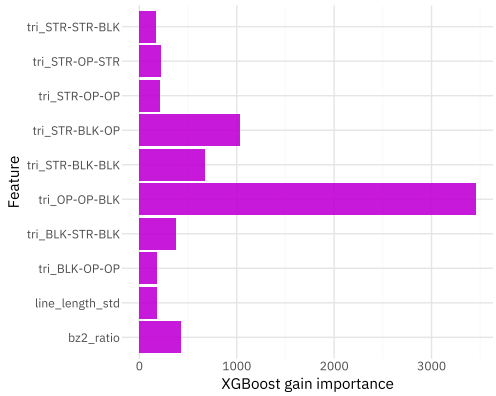


Figure 4: XGBoost feature importance for the structural view.

seem AI-generated to the structural analyst. However, the lexical view often corrects this if the variable names appear to be human.

To provide an accurate error analysis, we bin the files into length categories (Table 3 in Appendix C). The false negatives are usually AI snippets that are very short. For snippets with 1 to 10 lines, the macro F1 is only 0.317, with a low false negative rate indicating extreme bias. For files with more than 50 lines, the F1 rises to 0.570. Thus, the model struggles with short snippets because there is not enough text to extract the statistical anomalies; detecting AI in extremely short snippets without the help of embedder models remains an open problem.

We also think it to be worth mentioning that students or malicious users may try "to game the system" by altering the code slightly, but we expect COODetect to be naturally protected against these adversarial attacks. For instance, if an adversary modifies the variable names to look human, the structural view still detects the unusual entropy. To bypass the detector entirely, the adversary would need to rewrite the code, which defeats the purpose of using an AI generator in the first place.

6 Conclusion

In this article, we present an unsupervised framework for out-of-distribution AI-generated code detection. While code detection works well with embedder models like CodeBERT in-distribution, their performance drops drastically when facing unseen languages. We solve this by decomposing the signal into three orthogonal views (lexical, structural, and symbolic) and normalizing the predictions within automatically discovered latent domains. We achieve high stability without requiring target domain labels. Our approach is much more interpretable and generalizable than the deep neural mechanisms. We confirm that relative anomaly detection within local clusters is superior to global classification for cross-language generalization.

Acknowledgments

Aldan Creo gratefully acknowledges financial support for this publication by the Fulbright Foreign Student Program, which is sponsored by the U.S. Department of State and the Spanish Fulbright Commission. Its contents are solely the responsibility of the author and do not necessarily represent the official views of the Fulbright Program, the Government of the United States, or the Spanish Fulbright Commission.

References

Pavel Adamenko, Mikhail Ivanov, Aidar Valeev, Rodion Levichev, Pavel Zadorozhny, Ivan Lopatin, Dmitry Babayev, Alena Fenogenova, and Valentin Malykh. 2025. SWE-MERA: A Dynamic Benchmark for Agenticly Evaluating Large Language Models on Software Engineering Tasks.

Stefano Bistarelli, Marco Fiore, Ivan Mercanti, and Marina Mongiello. 2025. Usage of large language model for code generation tasks: A review. *SN Comput. Sci.*, 6(6).

L. Bulla, Alessandro Midolo, M. Mongiovì, and E. Tramontana. 2024. Ex-code: A robust and explainable model to detect ai-generated code. *Inf.*, 15:819.

Basak Demirok, Mucahid Kutlu, and Selin Mergen. 2025. Multiaigcd: A comprehensive dataset for ai generated code detection covering multiple languages, models,prompts, and scenarios. *ArXiv*, abs/2507.21693.

Andrea Gurioli, Maurizio Gabbrielli, and Stefano Zacchiroli. 2024. Is this you, llm? recognizing ai-written programs with multilingual code stylometry. *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 394–405.

Qiang Hu, Yuejun Guo, Xiaofei Xie, Maxime Cordy, Mike Papadakis, and Y. L. Traon. 2022. Codes: Towards code model generalization under distribution shift. *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 1–6.

Oseremen Joy Idialu, N. Mathews, Rungroj Maipradit, J. Atlee, and M. Nagappan. 2024. Whodunit: Classifying code as human authored or gpt-4 generated- a case study on codechef problems. *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 394–406.

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. 2025. Droid: A resource suite for ai-generated code detection. In *Conference on Empirical Methods in Natural Language Processing*.

Hyunjae Suh, Mahan Tafreshipour, Jiawei Li, Adithya Bhattiprolu, and Iftekhar Ahmed. 2025. An empirical study on automatically detecting ai-generated source code: How far are we? *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 859–871.

Xianjun Yang, Kexun Zhang, Haifeng Chen, L. Petzold, William Yang Wang, and Wei Cheng. 2023. Zero-shot detection of machine-generated codes. *ArXiv*, abs/2310.05103.

Mohammad Amin Zadenoori, Jacek Dąbrowski, Waad Alhoshan, Liping Zhao, and Alessio Ferrari. 2025. Large Language Models (LLMs) for Requirements Engineering (RE): A Systematic Literature Review.

A Data Distribution Analysis

The training data provided by the organizers contains short algorithmic snippets. When we analyze the data distribution, we observe that the average file length in the training set is less than 50 lines. The tokens are dense, and the vocabulary is limited to standard computer science problems (e.g., sorting, searching, LeetCode-style solutions).

However, the test set contains mixed unknown domains. We do not know if a file is an algorithmic script, a research notebook, or a production class. Figure 5 illustrates the massive difference in the length distributions between the splits. The test set has files with thousands of lines and complex enterprise boilerplate. Embedder models like CodeBERT memorize the length of the tokens and the specific syntax of the training languages. They fall into the syntax trap. When they see a long C# file in the test set, their attention mechanisms fail to find the familiar patterns, and they predict the labels with extreme but incorrect confidence. Our system avoids this by using features that are normalized by the file length automatically, like the compression ratio and the entropy variance.

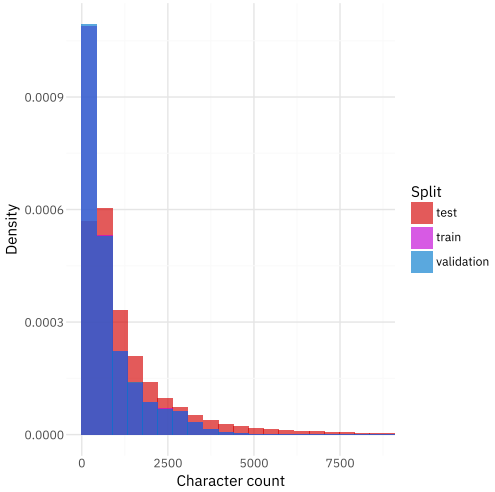


Figure 5: Distribution of snippet lengths in the train, validation, and test sets. While the character counts for the train and validation sets are nearly coincident, the test set includes some snippets that are considerably longer (production and research code).

B Latent Domain Details

C Error Analysis by Code Length

Cluster	Size	Avg Lines	Top Language	Length Bin
17	123366	28.7	Unknown	Medium
12	84418	43.2	Python	Medium
4	40162	14.6	Unknown	Medium
1	38876	20.5	Python	Medium
29	34802	65.2	Java	Long

Table 2: Top latent domains discovered by BGMM clustering on test structural features. Here, "Unknown" means that inside that cluster, `guesslang` (the language detection library) found the code to be too ambiguous to assign one of the 8 testing languages with high enough confidence. For example, this may happen when the code is malformed (some examples contain chat messages instead of proper code) or when the code is very short.

Length Bin	Macro F1	FNR	N
1 to 10 lines	0.317	0.041	78,287
11 to 50 lines	0.409	0.137	331,162
50+ lines	0.570	0.235	90,551

Table 3: Error analysis by code length, showing the macro F1 and the false negative rate (FNR) for different line counts.