CS 445 – Group 18
Project Final Report

Project Title:
SemEval-2026 Task 13

Group Members:
Barbaros Yahya 31143
Enes Çağlar 31109
Filiz Ilgaz Sönmez 32073
İsa Utku Dursunoğlu 30881
Mehmet Barış Baştuğ 30617

1. Introduction

The widespread adoption of large language models in the software industry has increased the importance of detecting machine-generated source code. This paper addresses SemEval-2026 Task 13 (Subtask A), which formulates the problem as a binary classification task distinguishing human-written code from machine-generated code. Our approach combines traditional machine learning and neural modeling techniques, including a CatBoost baseline built on handcrafted statistical features and multiple encoder-based transformer models fine-tuned for code semantics, namely CodeBERT, UniXcoder, and ModernBERT. In addition, we apply GPTSniffer-inspired preprocessing techniques to mask surface-level artifacts and encourage models to focus on deeper semantic patterns. All models are trained and evaluated using the official SemEval-2026 Task 13 Subtask A dataset and optimized based on Macro F1 score. Our results indicate comparable performance across models, underscoring the challenge of detecting machine-generated code and achieving robust generalization across diverse programming languages.

2. Related Work

Recent work on detecting machine-generated source code has explored both feature-oriented and neural approaches. GPTSniffer (Nguyen et al., 2024), for example, combines a CodeBERT-based classifier with a preprocessing stage that masks surface-level artifacts such as imports, comments, and indentation, encouraging models to rely on deeper semantic and structural information rather than characteristics specific to particular generators. Droid (Orel et al., 2025) presents a large-scale resource suite for AI-generated code detection, evaluating multiple models across diverse programming languages, domains, and generation strategies, and demonstrates that encoder-based transformer models perform strongly in human–AI binary classification. Continuing along this direction, CoDet-M4 (Orel et al., 2025) focuses on multilingual and multi-generator detection and shows that both CatBoost and transformer encoder models can achieve strong performance when properly tuned. At the same time, UniXcoder (Guo et al., 2022) offers a unified representation model based on an encoder that is pre-trained using source code, abstract syntax trees, and natural language comments. This solution facilitates the expression of complex syntactic and semantic features on source code, which are relevant within the discrimination task. Then, ModernBERT (Warner et al., 2025) introduces a modernized encoder-only architecture inspired by BERT- and DeBERTa-style models, incorporating rotary positional embeddings, efficient attention mechanisms, and large-scale pretraining that includes source code data, resulting in strong performance on classification and code-related tasks.

3. Methodology

We adopt the official SemEval-2026 Task 13 Subtask A dataset provided by the organizers. This dataset is mandatory for submission and evaluation under the data and model restrictions of the task, and no additional datasets were permitted. Therefore, we used the official SemEval dataset with fixed training, validation, and test splits to comply with competition rules.

Our approach is informed by recent literature indicating the value of both feature-based and encoder-based methods for this challenge. We therefore implement a CatBoost baseline with carefully tuned statistical features to capture surface-level code characteristics, and fine-tune several encoder-based transformer models, namely CodeBERT, UniXcoder, and ModernBERT, to learn deeper semantic and structural patterns. Furthermore, we adopt preprocessing inspired by GPTSniffer, which masks superficial artifacts such as imports and comments to prevent overfitting to generator-specific stylistic traits. Specifically, our methodology draws on and reimplements key modeling and evaluation choices from prior work on machine-generated code detection, including the feature-based and encoder-based frameworks proposed by Orel, Paul, et

al. (2025) in Droid, the detection setup introduced by Orel, Azizov, and Nakov (2025) in CoDet-M4, the preprocessing strategy of GPTSniffer (Nguyen et al., 2024), and the ModernBERT architecture described by Warner et al. (2025). Together, these choices allow us to systematically compare complementary modeling approaches within a unified, SemEval-compliant experimental setup.

3.1 CatBoost

We employed a CatBoost classifier as a feature-based baseline model, following procedures similar to those described in Droid (Orel, Paul, et al., 2025) and CoDet-M4 (Orel, Azizov, & Nakov, 2025). Both Droid and CoDet-M4 rely on statistical and structural characteristics of source code when employing a CatBoost classifier. Since neither paper released an official feature extraction pipeline, we reimplemented the feature set by aggregating all features explicitly mentioned across both papers. This resulted in a total of 13 handcrafted features, including average line length, whitespace ratio, number of lines, count of empty lines, comment density, average identifier length, maintainability index, average function length, function density, abstract syntax tree (AST) depth, assignment statement density, literal density, and function call density. The Maintainability Index was computed following the definition provided in Microsoft's official documentation (Microsoft, n.d.). AST-based features were extracted using the Tree-sitter parsing framework with language-specific grammars provided by libraries such as tree_sitter_python, tree_sitter_java, tree_sitter_c, among others, enabling consistent structural feature extraction across languages and improving generalization to previously unseen programming languages. As both Droid and CoDet-M4 report the use of hundreds of handcrafted features, not all were computationally feasible in our setting; therefore, the final feature set was limited to a manageable number of manually calculated features while preserving the most influential indicators highlighted in prior work. All features were extracted using a custom FeatureExtractor.

We trained the CatBoostClassifier with 2000 trees, a learning rate of 0.1, and a tree depth of 6, and a fixed random seed of 42. The choice of 2000 trees and a learning rate of 0.1 follows the configuration reported in CoDet-M4, where these values were shown to be optimal in terms of balancing convergence speed and generalization. While CatBoost supports extensive hyperparameter tuning, we did not perform exhaustive hyperparameter optimization. This decision was motivated by two factors. First, the inclusion of computationally expensive features, particularly AST-based features, substantially increased preprocessing time. Second, limited computational resources constrained the feasibility of large-scale grid searches, as CatBoost was not supported under the Tinker API.

3.2 ModernBERT

We used ModernBERT-base, a state-of-the-art encoder-only Transformer developed for efficient long-context classification, aligning with the design guide described in the ModernBERT paper. For this model, our objective is binary classification of human-written versus machine-generated code. The reason we used ModernBERT is primarily because of its efficiency in discriminating code, as well as its rotary positional embedding and efficient design, making it appropriate for sequence classification.

The training was conducted based on the Hugging Face Transformers library, initiated from the answerdotai/ModernBERT-base pretrained model and a sequence classification head designed for a two-way classification task. The code snippets to be predicted were tokenized appropriately, limited to a maximum length of 512 tokens, and included truncation. The handling of the dynamic padding was performed on a batch basis using a DataCollatorWithPadding. The fine-tuning process was conducted for three epochs with a batch size, learning rate, and weight decay of 128, 2e-5, and 0.01 respectively. For training and evaluation, we used the Trainer API and performed stepwise evaluation and checkpointing at intervals of 1000 steps. Metrics for

model selection were based on macro F1 score and early stopping was applied when two consecutive evaluations failed to improve performance. Enhancements for increased efficiency on modern GPUs included training with bfloat16 and TensorFloat32.

3.3 UniXcoder

We fine-tuned UniXcoder (microsoft/unixcoder-base) as an encoder-based baseline for SemEval-2026 Task 13 Subtask A. UniXcoder is a RoBERTa-style encoder pretrained on code and natural language, designed to learn semantic representations that transfer well across programming languages. We framed the task as binary sequence classification, where each input is a raw code snippet and the output label indicates whether the snippet is human-written or machine-generated.

We used the official SemEval dataset splits provided by the organizers in Parquet format, and trained only on the released training set while selecting models based on validation macro F1. Code snippets were tokenized using the UniXcoder tokenizer with truncation and a maximum sequence length of 512 tokens. To avoid excessive padding and improve efficiency, we employed dynamic padding at batch time using a DataCollatorWithPadding.

We evaluated UniXcoder with two preprocessing configurations to test its cross-language generalization behavior. First, we applied the variable masking approach used with CodeBERT, hypothesizing that UniXcoder's cross-lingual pretraining would further improve generalization. Second, we evaluated the C4 configuration (comments and imports removed only) to preserve code structure. UniXcoder was pretrained using Abstract Syntax Tree (AST) information alongside code text, which resulted in relatively worse compared to CodeBERT that would be explained in the results section. For modeling, we initialized AutoModelForSequenceClassification from the pretrained UniXcoder checkpoint with a two-class classification head. Training was conducted for 3 epochs using the Hugging Face Trainer API with AdamW optimization, a learning rate of 2e−5, and weight decay of 0.01. We evaluated every 500 steps and used early stopping (patience = 3 evaluations) while retaining the checkpoint with the best validation macro F1. When supported by hardware, we enabled mixed-precision training (FP16) and TensorFloat32 acceleration to improve throughput on modern GPUs. Performance was measured using accuracy and macro-averaged precision, recall, and F1, consistent with the SemEval evaluation protocol. For test-time inference, we applied the same tokenizer configuration and generated predictions in batches, exporting submissions in the required CSV format with an uppercase “ID” column.

3.4 TF-IDF-Based Lexical Baselines

In addition to neural and feature-based approaches, we implemented classical lexical baselines based on term frequency–inverse document frequency (TF-IDF) representations combined with linear classifiers. These models serve as lightweight, interpretable baselines that rely exclusively on surface-level lexical patterns rather than learned semantic representations. We constructed TF-IDF features over the raw code text using word n-grams, motivated by prior findings that AI-generated code often exhibits characteristic token and phrase distributions. Two configurations were evaluated. First, we trained a Multinomial Naive Bayes classifier using TF-IDF features with unigrams and bigrams (n = 1–2), a minimum document frequency of 5, and a capped vocabulary size of 100,000 features. Laplace smoothing ($\alpha = 0.1$) was applied to handle unseen tokens. Second, we trained a linear support vector machine using SGD optimization with hinge loss, using TF-IDF features with unigrams, bigrams, and trigrams (n = 1–3) and a vocabulary size of up to 150,000 features.

All models were trained on the official SemEval-2026 Task 13 Subtask A training set and evaluated on the provided validation split. As these methods are CPU-based and require no pretraining or fine-tuning epochs, they are computationally efficient and provide a strong

contrast to encoder-based transformer models. For test-time inference, predictions were generated directly from the trained pipelines and exported in the required submission format.

3.5 GraphCodeBERT

We additionally evaluated GraphCodeBERT (microsoft/graphcodebert-base), an encoder model pretrained to incorporate structural information from code via data-flow-aware pretraining objectives. We fine-tuned GraphCodeBERT for SemEval-2026 Task 13 Subtask A as a binary sequence classification problem, using raw code snippets as input and predicting whether each snippet was human-written or machine-generated.

We trained exclusively on the official SemEval training split and monitored performance on the provided validation split. Inputs were tokenized with the GraphCodeBERT tokenizer using truncation and a maximum sequence length of 512 tokens. To improve preprocessing throughput, tokenization was parallelized across available CPU cores using the datasets multiprocessing interface. Samples with missing code fields were removed to ensure stable tokenization.

For fine-tuning, we used the Hugging Face Trainer API for 3 epochs with AdamW optimization, a learning rate of 2e−5, and weight decay of 0.01. We used a per-device training batch size of 16 (and 32 for evaluation), and enabled mixed-precision training (FP16) to accelerate training on GPU. Evaluation and checkpointing were performed at the end of each epoch, and we retained the best checkpoint based on validation accuracy. Model performance was reported using accuracy and macro-averaged precision, recall, and F1.

3.6 CodeBERT

We fine-tuned CodeBERT (microsoft/codebert-base) with multiple preprocessing configurations to investigate cross-language generalization. CodeBERT is a bimodal pretrained model trained on the CodeSearchNet corpus, selected due to its proven effectiveness in code classification tasks. We have tried three preprocessing strategies:

C8 Configuration (from GPTSniffer): Following Nguyen et al. (2024), we removed all comments and import statements while normalizing whitespace.

C4 Configuration (from GPTSniffer): A lighter approach removing only comments and imports while preserving variables, strings, and numeric literals.

Variable Masking (Our contribution) : We extended GPTSniffer's proposed approaches by additionally masking variable names → `_V_`, string literals → `_S_`, and numeric literals → `_N_`, while preserving 100+ language specific keywords (if , else, etc.). Since the model testing includes unseen domains and unseen languages, this approach is used to force model to learn underlying structural patterns instead of language-specific naming conventions that would not generalize to unseen languages.All experiments used identical hyperparameters: batch size 32 with gradient accumulation (effective batch 128), learning rate 2e-5, weight decay 0.01, maximum sequence length 512, and 3 training epochs with early stopping (patience=3). Training was conducted on NVIDIA A100 GPUs with FP16 precision.

3.7 XGBoost with Hybrid Feature Engineering

We implemented an XGBoost classifier to evaluate the effectiveness of combining lightweight, interpretable stylistic features with traditional lexical representations. Unlike deep learning approaches that require heavy computational resources for token embedding, this method leverages gradient boosting on decision trees to capture non-linear interactions between code structure and keyword usage.

For feature engineering we used two components. The first one is getting the advantage of handcrafted structural features. We extracted the stylistic features of the code to gain

upperhand knowledge among different languages. The feature set includes line count, average line length, indentation patterns (average leading spaces), and the density of special characters (e.g., brackets {}, (), and semicolons ;). These numerical features were normalized using standard scaling based on training set statistics. The second method we used is the traditional TDF-IDF method to capture the vocabulary frequencies capped on 5000 most frequently used features to maintain memory efficiency. We trained the XGBoost classifier using the hist tree method with CUDA acceleration (NVIDIA A100) to handle the sparsity of the data efficiently. The model was configured with 2,000 estimators, a maximum depth of 8 to capture complex feature interactions, and a learning rate of 0.05. To prevent overfitting, we employed a subsampling rate of 0.8 for both data instances and columns per tree, and utilized early stopping monitored on the validation set.

4. Results

The ModernBERT-based classifier has shown remarkable performance on the validation set for Subtask A, capable of distinctly distinguishing human-written code from machine-generated code with a macro F1-score of 0.9970 along with macro precision and macro recall both equal to 0.9970, indicating balanced performance across classes. As visible in Appendix A for the ModernBERT-based classifier, with time, both the train and validation losses decrease with no signs of overfitting being visible. The confusion matrix visible only in Appendix A indicates that the vast majority of predictions are correct, with only a negligible number of misclassifications, reflecting excellent class separability. As visible from Appendix A for ModernBERT-based classifiers, the precision-recall graph approaches perfection for a vast majority of the recall range with an approximate AUC value of 0.9999. All such observations confirm that ModernBERT with its massive pre-training coupled with innovations such as Rotary Positional Embeddings is a reliable solution for accuracy on the validation set for Subtask A of the SemEval-2026 Task 13.

The CatBoost classifier achieves very strong performance on the validation set, with macro F1, macro precision, and macro recall all reaching 0.97. The corresponding confusion matrix (see Appendix B) shows balanced behavior across both classes, with relatively low false positive and false negative rates. The precision-recall curve on the validation set further supports this observation, yielding an average precision of 0.99 and remaining well above the chance level. These results indicate that the model is able to effectively separate human-written and machine-generated code when evaluated on data drawn from the same distribution as the training set. However, performance drops substantially on the test set, where the macro F1 score decreases to approximately 0.38. This behavior is also reflected in the test set precision-recall curve, which lies only modestly above the chance baseline. The discrepancy between validation and test performance highlights the difficulty of the task, suggesting that generalization to unseen distributions remains a significant challenge. We also analyzed feature importance using SHAP values. The SHAP beeswarm plot (Appendix B) reveals that the number of empty lines is the most influential feature, closely aligning with findings reported in CoDet-M4. Other highly influential features include average identifier length**,** AST depth, and average function length. During experimentation, we explored incorporating further features, such as the maximum length of decision operators, but these were ultimately excluded due to unclear definitions in the referenced papers and limited empirical benefit. All CatBoost related results, including confusion matrices, precision-recall curves, and SHAP visualizations, are provided in Appendix B.

UniXcoder achieved strong validation performance, reaching a macro F1 score of approximately 0.994 (accuracy ≈ 0.994), with balanced macro precision and recall (both ≈ 0.994). This indicates that the pretrained code-aware encoder is highly effective at distinguishing human-written and machine-generated code when evaluated on in-distribution validation data. However, when evaluated on the hidden test set via the official Kaggle submission, the UniXcoder-based model attained a public macro F1 score of 0.28840, reflecting a substantial

performance drop relative to validation. Our novel approach on masking variables collapsed the model to 0.18 test F1 score by only predicting AI for nearly 99.8% of the samples despite the 0.982 validation F1 score. Which proves masking damages the AST structure UniXcoder trusts on . Another experimental setup we used for UniXcoder is by removing the comment section from all data samples while preserving code structure resulted in 0.31 test F1, 93% human precision, but only 14% recall due to class imbalance. This discrepancy mirrors the behavior observed for other models in our study and highlights a key challenge of SemEval-2026 Task 13: generalization across unseen distributions and generation strategies. Despite strong inductive biases and extensive pretraining on source code, UniXcoder appears to rely on patterns that do not transfer reliably to the test set, suggesting sensitivity to dataset artifacts or generator-specific cues present in the training data. These findings reinforce the importance of robustness-focused evaluation and motivate further discussion on distribution shift and artifact reliance in Section 5.

The TF-IDF-based lexical baselines achieved moderate performance on the validation set. The Multinomial Naive Bayes classifier with unigram and bigram features reached a validation accuracy of approximately 0.83 (macro F1 $\approx$ 0.83), while the linear SVM with unigram to trigram features improved validation accuracy to approximately 0.85. These results indicate that surface-level lexical statistics alone can capture meaningful differences between human-written and machine-generated code, although they lag behind encoder-based models in in-distribution evaluation. However, when evaluated on the hidden test set via the Kaggle submission system, the TF-IDF + SVM model achieved a public macro F1 score of 0.45890, substantially outperforming all transformer-based models in terms of test generalization. This finding suggests that although lexical models underperform on validation data, they may be less sensitive to generator-specific artifacts and thus generalize more robustly to unseen distributions. The strong relative performance of this simple baseline highlights the severity of distribution shift in SemEval-2026 Task 13 and underscores the limitations of relying solely on large pretrained encoders for AI-generated code detection.

GraphCodeBERT achieved very high validation performance, reaching an accuracy of approximately 0.995 and a macro F1 score of approximately 0.995. Across epochs, both training and validation losses remained low, and validation macro precision and recall were balanced ($\approx$ 0.995 each), indicating strong class separability on the validation split. However, when evaluated on the hidden test set via Kaggle submission, the GraphCodeBERT model attained a public macro F1 score of 0.2505, representing a substantial drop relative to validation. This outcome further supports the broader pattern observed in our experiments: encoder-based transformer models can reach near-ceiling scores on the released validation data yet fail to generalize to the test distribution. In particular, GraphCodeBERT's strong in-distribution performance but weak test score suggests that high-capacity pretrained encoders may exploit dataset-specific artifacts or generator-dependent cues that do not transfer to unseen data. We discuss this distribution-shift behavior and its implications for robust AI-generated code detection in Section 5.

We evaluated different configurations for CodeBERT to investigate cross-language generalization. In our first experiment with CodeBERT, no preprocessing applied and this version achieved 0.995 validation F1 score whereas it only gives 0.25 on the test set which shows overfitting to training language patterns. For the second experiment with CodeBERT we removed comments and imports stated as C8 configuration in GPTSniffer paper and achieved similar results 0.988 on validation F1 and 0.25 on test F1, which again shows basic preprocessing does not address the generalization problem. Since the CodeBERT model is mostly predicting AI (approximately %90 of predictions) , we tried threshold calibration using value of t as 0.9, we could improved the test F1 to 0.345 which stem from the class distribution mismatch between training data (%50 AI) and test data (%22 AI). For experimenting a novel approach that is not tested in GPTSniffer paper, we tried to mask variables to help the model achieve better generalization performance by focusing on underlying patterns of code instead of

semantic features. It achieved the best test performance by 0.345 test F1 without the need for calibrating the threshold. Per language validation scores were 0.9888(for Python) , 0.9339 (For Java) and 0.9153 (for C++). Detailed metrics and confusion matrices are provided in Appendix C.

The XGBoost classifier, utilizing a hybrid of handcrafted structural features and TF-IDF vectors, showed a great performance on the validation set, achieving a macro F1 score of 0.9831 and an accuracy of 0.9831. The classification report reveals a highly balanced profile, with the model identifying human-written code with a recall of 0.99 and machine-generated code with a precision of 0.99. As indicated by the confusion matrix (see Appendix D), the model successfully separates the two classes. However, as it is in previous experiments, this performance did not show up to the unseen test data. The model achieved a public macro F1 score of only 0.27 on the official leaderboard. This precipitous drop—from ~0.98 to 0.27—indicates that while the structural features are highly predictive within the specific constraints of the training data, they are not robust to the distribution shifts present in the test set.

5. Discussion

We selected the official SemEval-2026 Task 13 Subtask A dataset for this study, enabling direct comparison with other teams. The composition of this dataset was very decisive on the performance because of the generalization problem. While the training and validation sets were perfectly balanced, test was very inconsistent with 22% machine-generated code. Furthermore, the sharp discrepancy between our near-perfect validation scores (Macro F1 ~0.99) and the substantially lower test scores (Macro F1 ~0.25–0.46) suggests that the training data likely contained unique import patterns or variable naming conventions, that were absent or modified in the test set. This distribution shift caused our models to memorize training statistics rather than learning generalized distinctions between human and AI logic.

Our methodology combined feature-based systems (XGBoost, CatBoost) with encoder-based transformers (ModernBERT, CodeBERT, UniXcoder) to capture both surface-level structure and deep semantic patterns. The primary advantage of the transformer-based approach was its ability to model complex dependencies, yielding distinct class separability on in-distribution data. Conversely, the feature-based approaches provided interpretability. However, a major disadvantage of the handcrafted feature approach was its fragileness; structural heuristics like indentation depth or line length proved unstable across the unseen languages in the test set. Similarly, the unmasked transformer models suffered from overfitting to superficial tokens, lacking the robustness to generalize to new generation strategies.

When comparing our results against standard baselines, a counter-intuitive trend emerged where simpler models outperformed complex architectures on the test set. The TF-IDF + Linear SVM baseline achieved the highest test performance (Macro F1 $\approx$ 0.46), significantly outperforming state-of-the-art models like ModernBERT (Test F1 $\approx$ 0.28–0.35) and GraphCodeBERT (Test F1 $\approx$ 0.25). Additionally, our variable masking strategy on CodeBERT had the best neural test performance (F1 $\approx$ 0.35) by forcing focus on structure, yet the same strategy severely degraded UniXcoder (F1 $\approx$ 0.18), indicating that UniXcoder relies heavily on the semantic interplay between variable names and AST nodes which masking disrupted.

The most significant limitation of the proposed system is its lack of robustness to distribution shifts and unseen domains. Despite achieving validation F1 scores exceeding 0.99 across almost all models, the system failed to maintain this performance on the test set. Specifically, the XGBoost and CatBoost models relied on structural features (e.g., average line length, AST depth) that are language-dependent, making them ineffective when the test set introduced programming languages with different formatting norms. Additionally, the system

struggled with the class imbalance in the test set, leading to high recall but low precision as models over-predicted the minority “AI” class.

Given more time and resources, we would prioritize several improvements to address these generalization and robustness issues. First, we would expand the training set using adversarial generation and data augmentation (e.g., varying indentation, renaming variables) to break the model's reliance on specific stylistic artifacts. Second, instead of relying on individual models, we would implement a weighted ensemble combining the robust generalization of the TF-IDF SVM with the structural understanding of the masked CodeBERT. Finally, to address the class imbalance, we would apply post-hoc probability calibration or dynamic thresholding to optimize the decision boundary for the target distribution.

6. Conclusion

In this work, we propose to tackle SemEval-2026 Task 13 Subtask A, which deals with code discrimination between code produced by a human versus code produced by a machine. We systematically compare a large range of methods, from feature-based, lexical approaches, to encoder-based transformers, in a SemEval-compliant setting. Despite nearly all models reaching nearly perfect scores on the validation set, this work highlights that a dramatic drop in scores on the public test set exists, indicating a strong setting of distribution shift and a lack of generalization to out-of-distribution data, which represents the primary issue of this particular problem at hand. Critically, simple lexical approaches, which use TF-IDF together with linear models, generalize better than state-of-the-art transformers, which have been trained on a large amount of data. Techniques like variable masking improve the generalization abilities of certain models, but hinder those of others. Overall, this work has shown that code produced by a machine can be identified by code detectors that have high in-distribution performances, but puts forward a series of requirements regarding generalization that must be fulfilled by future work.

7. Individual Contributions

İsa Utku Dursunoğlu led the development and fine-tuning of the CodeBERT and UniXcoder classification model integrated with GPTSniffer-based preprocessing and additional variable masked preprocessing for novelty. This work included tokenizer initialization, masked-input preparation, model training, and evaluation on the benchmark dataset.

Filiz Ilgaz Sönmez developed the CatBoost baseline by designing and implementing the feature extraction pipeline for the code data, and evaluating baseline performance in comparison with transformer-based models.

Enes Çağlar implemented and evaluated an additional encoder-based transformer model, ModernBERT. Responsibilities included model configuration, tokenization, and conducting performance-focused experiments.

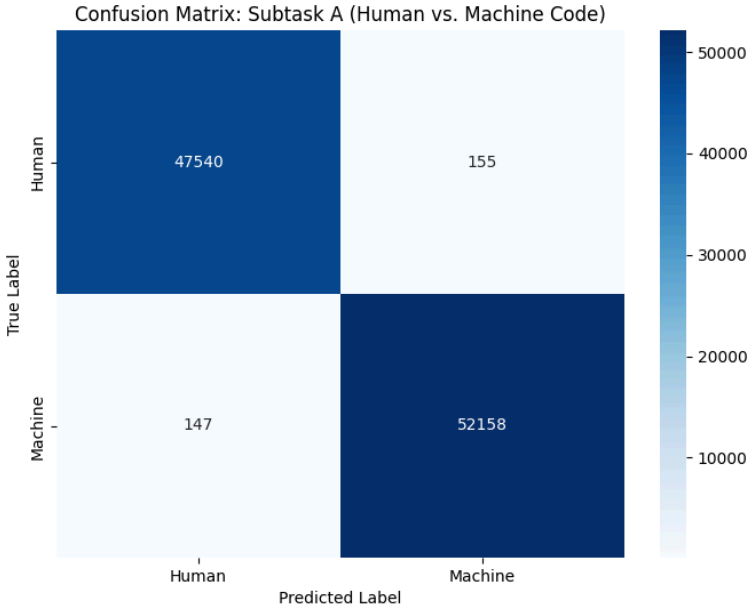
Mehmet Barış Baştuğ implemented and evaluated the UniXcoder encoder-based transformer, along with additional neural and TF-IDF-based lexical baselines, and conducted performance-focused experiments with comparative analyses of model effectiveness and generalization behavior.

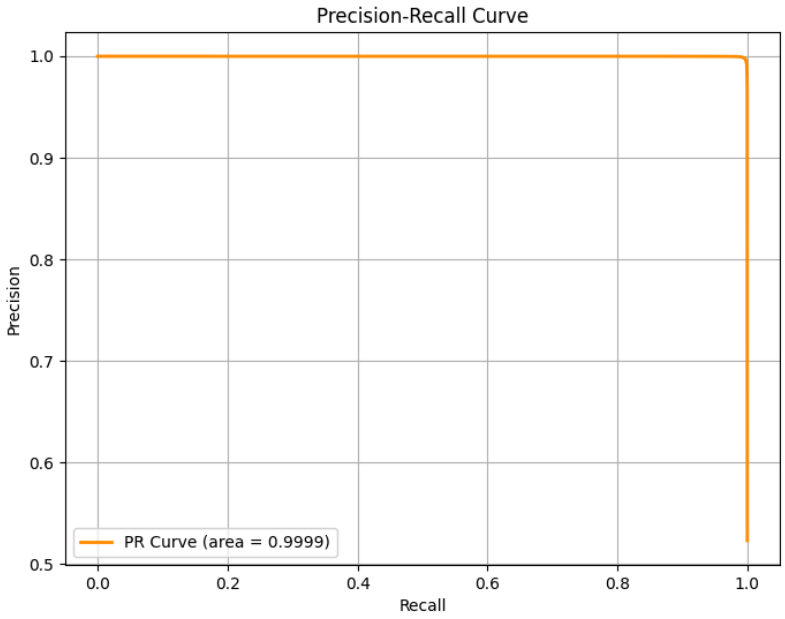
Barbaros Yahya was responsible for implementing the XGBoost-based baseline model and conducting performance-focused experiments, including model training, evaluation, and comparative analysis within the overall experimental pipeline.

Appendix

Appendix A: ModernBERT Evaluation Results

```
--- FINAL VALIDATION METRICS ---
Macro F1-Score: 0.9970
Macro Precision: 0.9970
Macro Recall:    0.9970
```

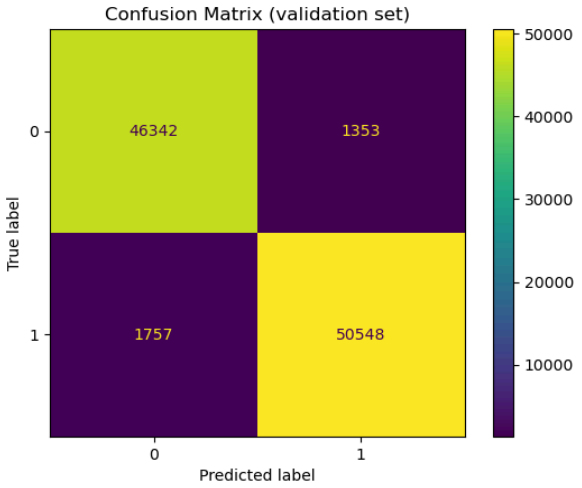


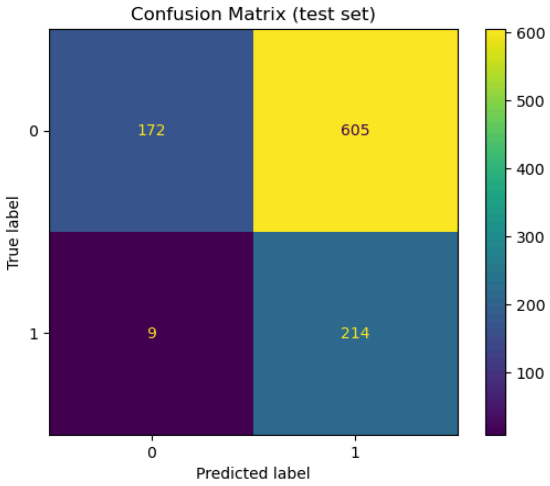

Precision-Recall Curve
Precision
Recall
PR Curve (area = 0.9999)
1.0
0.9
0.8
0.7
0.6
0.5
0.0
0.2
0.4
0.6
0.8
1.0

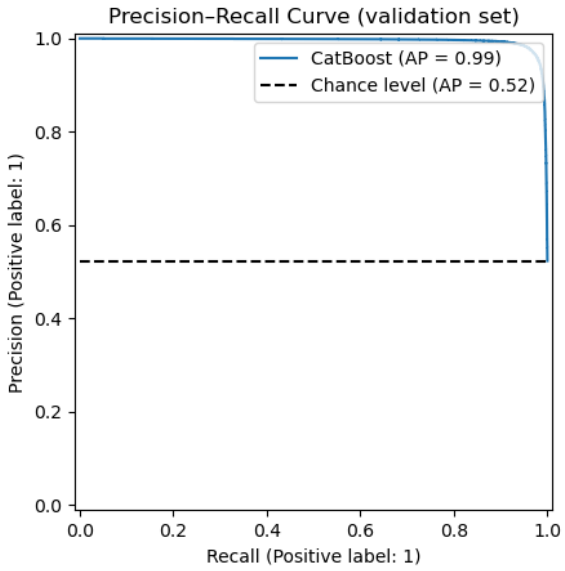
Appendix B: CatBoost Evaluation Results

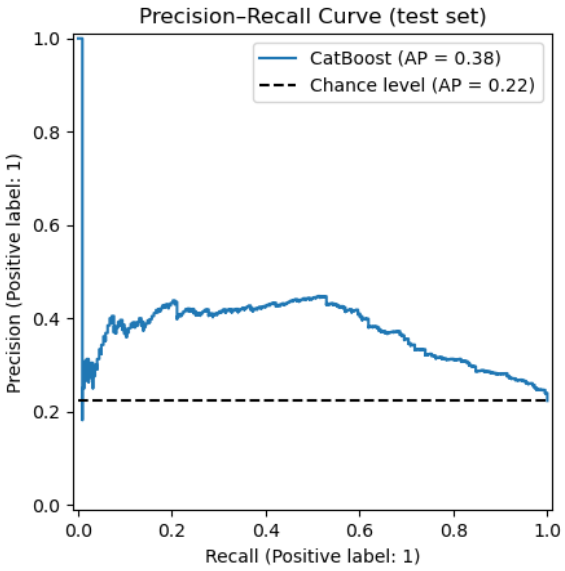
```
Evaluation on the validation set
Macro F1 score (validation): 0.9688
Macro precision (validation): 0.9687
Macro recall (validation): 0.9690

Classification report (validation set):
              precision    recall  f1-score   support

           0       0.96      0.97      0.97     47695
           1       0.97      0.97      0.97     52305

    accuracy                           0.97    100000
   macro avg       0.97      0.97      0.97    100000
weighted avg       0.97      0.97      0.97    100000
```

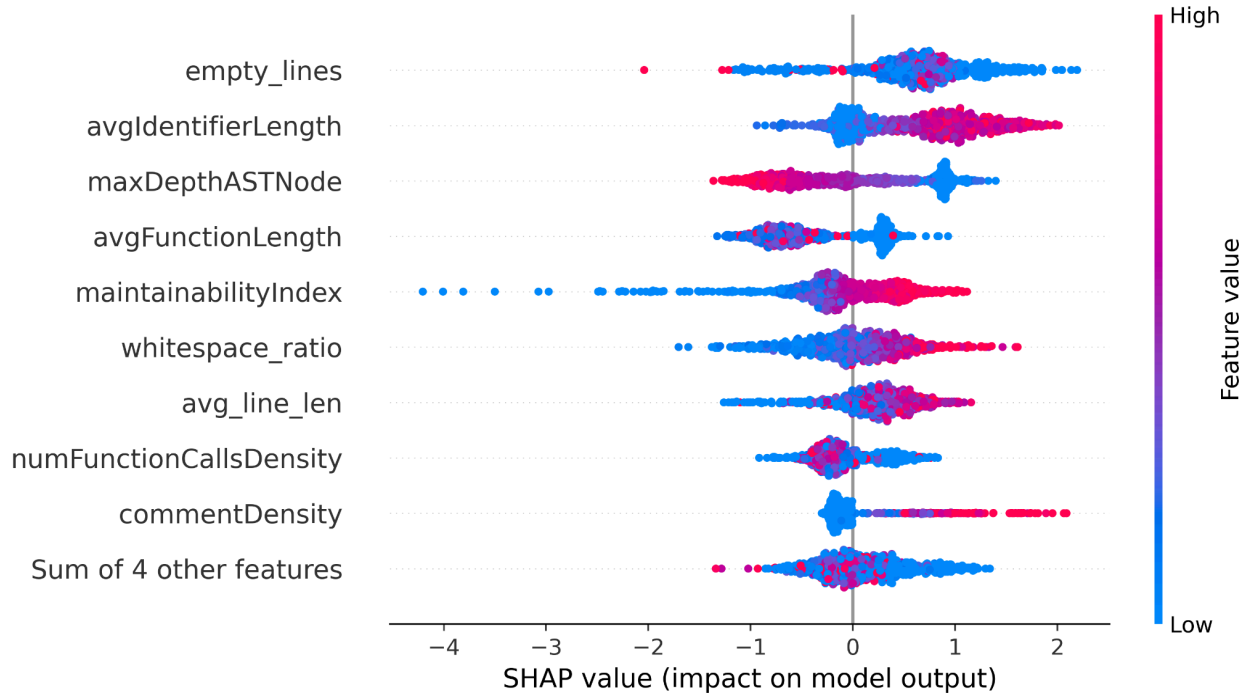
```
Evaluation on the test set
Macro F1 score (test): 0.3849
Macro precision (test): 0.6058
Macro recall (test): 0.5905

Classification report (test set):
              precision    recall  f1-score   support

           0       0.95      0.22      0.36       777
           1       0.26      0.96      0.41       223

    accuracy                           0.39      1000
   macro avg       0.61      0.59      0.38      1000
weighted avg       0.80      0.39      0.37      1000
```






Precision–Recall Curve (validation set)
CatBoost (AP = 0.99)
Chance level (AP = 0.52)
Precision (Positive label: 1)
Recall (Positive label: 1)


Precision–Recall Curve (test set)
CatBoost (AP = 0.38)
Chance level (AP = 0.22)
Precision (Positive label: 1)
Recall (Positive label: 1)


empty_lines
avgIdentifierLength
maxDepthASTNode
avgFunctionLength
maintainabilityIndex
whitespace_ratio
avg_line_len
numFunctionCallsDensity
commentDensity
Sum of 4 other features
SHAP value (impact on model output)
Feature value
High
Low

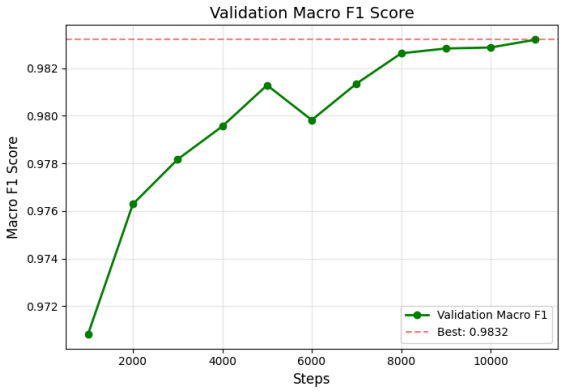
Appendix C: CodeBERT with Masking Evaluation Results

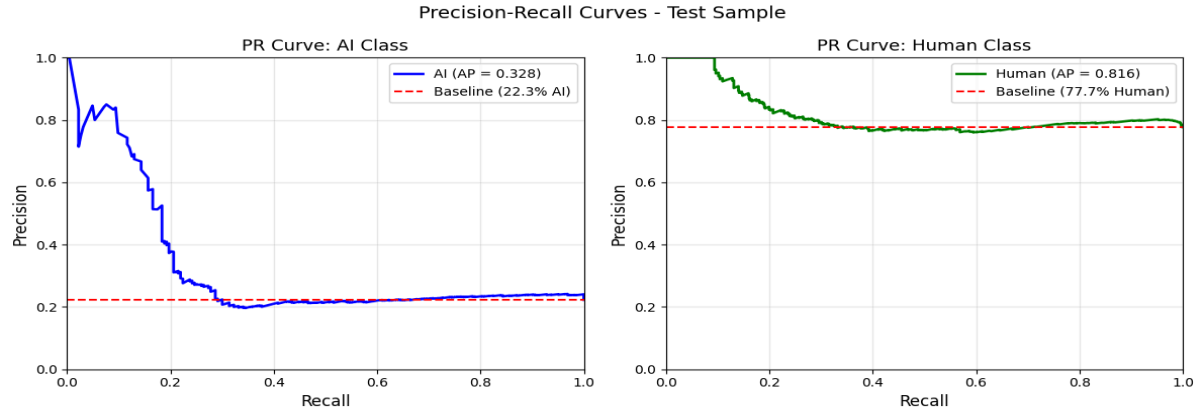
TEST SAMPLE CLASSIFICATION REPORT

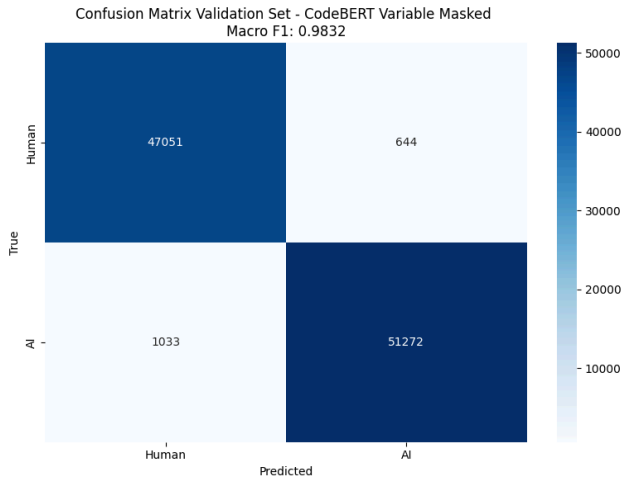
	precision	recall	f1-score	support
Human	0.8407	0.1969	0.3191	777
AI	0.2372	0.8700	0.3727	223
accuracy			0.3470	1000
macro avg	0.5389	0.5334	0.3459	1000
weighted avg	0.7061	0.3470	0.3310	1000

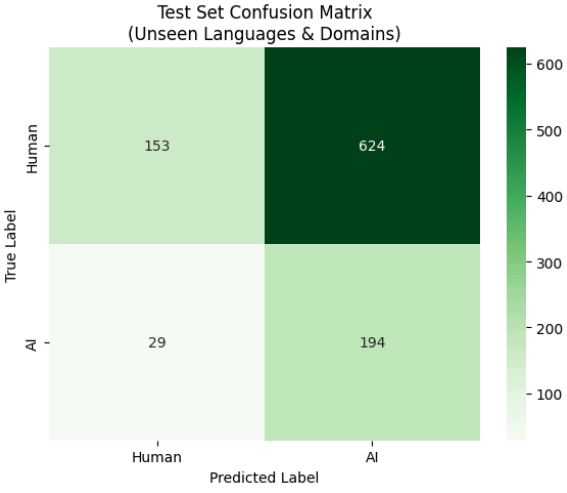
CLASSIFICATION REPORT FOR VALIDATION SET

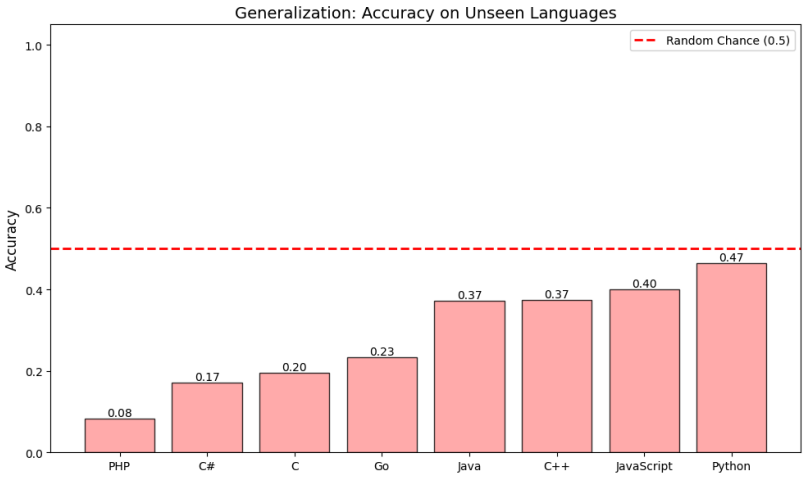
	precision	recall	f1-score	support
Human	0.9785	0.9865	0.9825	47695
AI	0.9876	0.9803	0.9839	52305
accuracy			0.9832	100000
macro avg	0.9831	0.9834	0.9832	100000
weighted avg	0.9833	0.9832	0.9832	100000



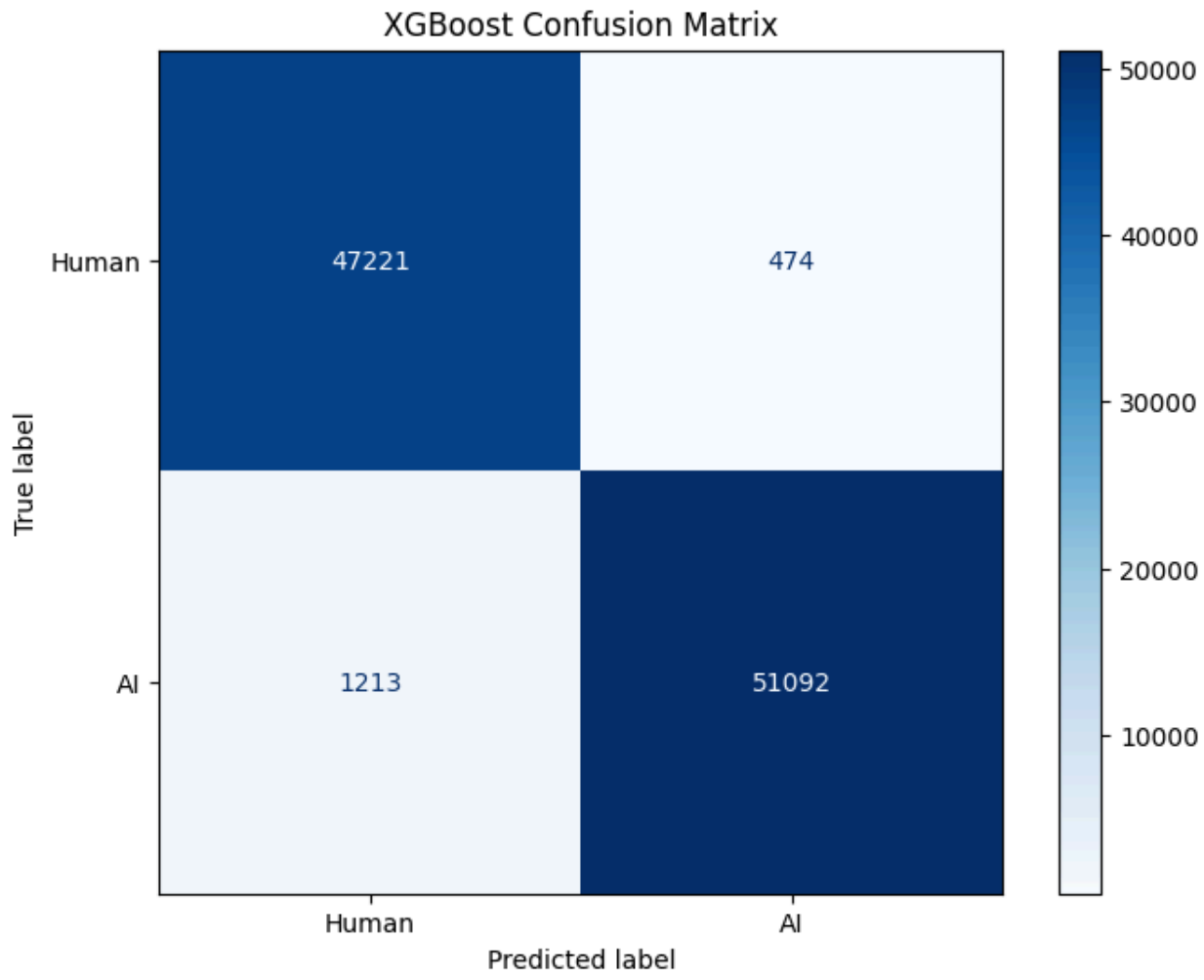







Generalization: Accuracy on Unseen Languages
Random Chance (0.5)
Accuracy
1.0
0.8
0.6
0.4
0.2
0.0
0.08
0.17
0.20
0.23
0.37
0.37
0.40
0.47
PHP
C#
C
Go
Java
C++
JavaScript
Python

Appendix D: XGBoost Confusion Matrix on Validation Set



References

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. In T. Cohn, Y. He, & Y. Liu (Eds.), *Findings of the Association for Computational Linguistics: EMNLP 2020* (pp. 1536–1547). Association for Computational Linguistics. https://doi.org/10.18653/v1/2020.findings-emnlp.139

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., & Yin, J. (2022). UniXcoder: Unified cross-modal pre-training for code representation. *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 7212–7225. Association for Computational Linguistics. https://doi.org/10.48550/arXiv.2203.03850

Nguyen, P. T., Di Rocco, J., Di Sipio, C., Rubei, R., Di Ruscio, D., & Di Penta, M. (2024). GPTSniffer: A CodeBERT-based classifier to detect source code written by ChatGPT. *Journal of Systems and Software*, 214, 112059. https://doi.org/10.1016/j.jss.2024.112059

Microsoft. (n.d.). *CodeBERT*. GitHub. Retrieved November 22, 2025, from https://github.com/microsoft/CodeBERT

Microsoft. (n.d.). *Code metrics - Maintainability index range and meaning*. Microsoft Learn. Retrieved January 4, 2026, from https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=visualstudio

Orel, D., Azizov, D., & Nakov, P. (2025). CoDet-M4: Detecting machine-generated code in multi-lingual, multi-generator, and multi-domain settings. In W. Che, J. Nabende, E. Shutova, & M. T. Pilehvar (Eds.), *Findings of the Association for Computational Linguistics: ACL 2025* (pp. 10570–10593). Association for Computational Linguistics. https://doi.org/10.18653/v1/2025.findings-acl.550

Orel, D., Paul, I., Gurevych, I., & Nakov, P. (2025). Droid: A resource suite for AI-generated code detection. In C. Christodoulopoulos, T. Chakraborty, C. Rose, & V. Peng (Eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing* (pp. 31251–31277). Association for Computational Linguistics. https://doi.org/10.18653/v1/2025.emnlp-main.1593

Warner, B., Chaffin, A., Clavié, B., Weller, O., Hallström, O., Taghadouini, S., Gallagher, A., Biswas, R., Ladhak, F., Aarsen, T., Adams, G. T., Howard, J., & Poli, I. (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) (pp. 2526–2547). Association for Computational Linguistics. https://aclanthology.org/2025.acl-long.127.pdf