

A Comparative and Explainable Approach for Detecting AI-Generated Source Code Across Programming Languages

Saurabh Tiwari (Pursuing MCA)
saurabhtiwari759901@gmail.com

Shivom Pawar (Pursuing MCA)
Shivompawar2@gmail.com

Abhinav Chauhan (Pursuing MCA)

Singhabhinav2024@gmail.com

Mr. Abhilash Saxena (Assistant Professor) MCA
MIT Moradabad/AKTU Moradabad, Uttar Pradesh, India

ABSTRACT

Large language models (LLMs) have transformed software development by enabling generation of functional source code from natural-language requirements, but this capability creates a new authorship and provenance challenge: whether a submitted program was written by a human, generated by an AI system, or produced via a mixture of both. The problem is particularly challenging across programming languages due to lexical and formatting patterns that appear predictive in one language but not others. This paper proposes X-AIGCD, a comparative and explainable framework for AI-generated source-code detection that leverages language-agnostic statistical and stylometric features, abstract-syntax-tree (AST) structural features, and semantic representations from pre-trained code models to detect plagiarism. The proposed system compares classical machine learning baselines (TF-IDF + Logistic Regression/SVM, Random Forest, and XGBoost) with CodeBERT and GraphCodeBERT-based detectors, and then performs feature-level late fusion. SHAP-based attribution is used to explain individual predictions and quantify the contributions of lexical, structural, and semantic feature groups. The evaluation protocol is designed around cross-language and distribution-shift testing, including training on C++, Python, and Java and testing on unseen languages such as Go, PHP, C#, C, and JavaScript. This design is motivated by recent evidence that in-distribution performance can substantially overestimate real-world detection ability. Importantly, this paper does not fabricate experimental results for the proposed model but instead reports published benchmark evidence and defines a reproducible experimental protocol for obtaining original results. The resulting framework provides a practical basis for comparing accuracy, robustness, computational cost, and explainability. Because this manuscript is being prepared under a short submission deadline, the Results section reports verified published benchmark measurements as secondary empirical evidence and clearly distinguishes them from original X-AIGCD results that require execution on the official dataset.

Keywords—AI-generated code, source-code authorship, LLM detection, cross-language detection, code stylometry, AST, CodeBERT, GraphCodeBERT, XGBoost, explainable AI, SHAP, distribution shift

1. INTRODUCTION

Generative AI systems are now capable of generating code for algorithmic problems, application logic, tests, documentation, and debugging. As a result, code authorship is no longer a binary proposition since there are now means and sources of production beyond human coders. Code can be fully human written, fully machine generated, or a combination of both, with the latter encompassing scenarios where an LLM produces only some functions or edits. A recent benchmarking effort has already identified this situation as a critical practical detection scenario for AI detectors [1], [2].

State-of-the-art detectors have demonstrated that AI code can have telltale statistical, stylistic, structural, and semantic patterns. GPTSniffer, for instance, leverages CodeBERT to classify ChatGPT source code and argues that training on relevant and paired snippets improves performance [3]. At the same time, most recent papers report that one must be careful not to extrapolate the results of a detector trained on one language, model, or task to another.

The cross-lingual generalization ability of AI detectors is of immediate practical concern since code can be written in dozens of programming languages. Specifically, SemEval-2026 Task 13 defines a binary detection scenario, where the train set consists of C++, Python, and Java, while the test set includes additional languages such as Go, PHP, C#, C, and JavaScript. According to the task description, the ranking metric is the Macro-F1 score, since data imbalance can affect the estimation of accuracy. In fact, the preliminary results of the task show that there are significant discrepancies between the validation and test performance for many systems, which highlights the importance of evaluating cross-lingual generalization [4], [5], [6].

Therefore, this paper focuses on a comparative question: what is the best combination of representations that achieve the best trade-off between detection performance, computational overhead, linguistic generality, and interpretability? Our hypothesis is that source code needs to be represented as a multi-view input, where different representations learn diverse patterns. Specifically, we identify three views: one is related to surface statistics and coding style, another is connected to program structure and syntax, and AST-based features constitute the third view, capturing semantic information.

A. RESEARCH CONTRIBUTIONS

- A language-agnostic feature layer capturing lexical, formatting, stylistic, and complexity-related properties of source code.
- An AST-based structural layer representing control-flow and syntactic organization beyond simple token sequences.
- A comparison of classical machine learning, code-specific transformers, and a late-fusion hybrid model.

- A cross-language evaluation protocol distinguishing seen-language, unseen-language, seen-domain, and unseen-domain performance.
- An explainability layer based on SHAP attributing decisions to interpretable groups of features and individual terms.
- A reproducible reporting strategy distinguishing original experimental measurements from results published by previous studies.

B. PAPER ORGANIZATION

The remainder of this paper is organized as follows. Section II contains related works on detecting AI-generated code, code representation learning and analysis, AST-based approaches, and explainable machine learning. Section III defines the research gap, problem statement, objectives, scope, and research questions. Section IV introduces our proposed X-AIGCD framework. Section V describes how we construct, preprocess, engineer, train models, and fuse data. Section VI explains the model's explainability. Section VII defines the experiment's design and evaluation metrics. Section VIII presents benchmark evidence and describes how to report original experimental results. Section IX discusses the limitations and threats to validity, while Section X concludes this study.

2. LITERATURE REVIEW

A. AI-GENERATED CODE DETECTION

Earlier research into methods of detecting AI-generated code was based on the fact that such code may differ from human-written code in formatting, naming conventions, comments, token frequency patterns, or other features. GPTSniffer presented a CodeBERT-based classifier for source-code detection and showed that the choice of training data and pairing strategies has an effect on the performance [3]. This is an essential finding, as a classifier may learn the artifacts specific to a particular generator rather than general properties of machine-code.

The later research utilizes broader sets of potential differences. MultiAIGCD is a data set of human- and machine-generated python/java/go code with varying generators, prompts, and application scenarios, including generation from problem descriptions and AI-enhanced repair of human code [7]. Both human-written code snippets (32,148) and AI-generated code snippets (121,271) were filtered for syntax-quality [7]. The experiments indicate that performance varied across languages, generators, and application scenarios, and that code fixed by AI was harder to detect than code generated directly from problem descriptions.

B. MULTILINGUAL AND DISTRIBUTION-SHIFTED DETECTION

SemEval-2026 Task13 defines the distribution shift as the research task. The Task's Subtask A features 500,000 training instances, with C++, Python, Java as training languages, and the claim to evaluate both seen and unseen languages, as well

as seen and unseen application domains [1]. This definition of distribution shift underpins the present work's cross-language setting.

Several works from 2026 provide evidence on the challenges of this setting. For instance, the LAFED system, which combines structural, character/whitespace, and micro-style features with XGBoost, achieves an official Macro-F1 score of 0.75209, with better performance on the unseen C# and JavaScript and worse performance on Go and PHP [4]. In turn, the Archaeology system reports a Macro-F1 of 0.737 for Subtask A, using CodeBERT, GraphCodeBERT, UniXcoder, and CodeT5+ with leave-one-language-out, augmentation, chunked inference, and calibration [5]. UIT-AMMC, another system, reports a score of 0.802 Macro-F1, using a structure-aware contrastive framework and QLoRA-based code modeling [6]. While these numbers are not directly comparable due to different train-test splits, hyperparameters, training data, model architectures, and other factors, the differences between the results underscore the importance of considering cross-language variations.

C. CODE REPRESENTATION LEARNING

CodeBERT is a Transformer-based pre-trained model for programming and natural languages that was trained using both natural-language/code pairs and unimodal code data [8]. GraphCodeBERT extends this idea by incorporating data-flow information such that the representation is sensitive to relationships between variables and values rather than simply treating code as a token sequence [9]. CodeT5+ explores flexible encoder-decoder representations and multiple pre-training objectives for code understanding and generation [10].

These models offer strong semantic representations, but their computational cost and dependence on training distributions may limit their robustness to language or domain shift. Recent SemEval submissions therefore motivate a combination of transformer representations with structural or statistical features rather than assuming that a single neural model is universally optimal [4], [5], [6].

D. AST AND STRUCTURAL FEATURES

An Abstract Syntax Tree (AST) is a representation of program syntax that captures its internal structure. Derived measurements from ASTs may include statistics of node frequencies, nesting depth, control-flow density, number of functions and classes, operator and operand frequency, and syntax-pattern n-grams. Recent works on the detection of AI-generated-code explicitly considered AST analysis as a possible approach to take advantage of the structure of programs rather than relying only on surface tokens [11]. Structural features are appealing to cross-language studies since lots of the concepts such as function, branch, loops, calls, literals, declarations, nesting, etc., have their equivalent in the vast majority of the mainstream programming languages.

E. EXPLAINABLE MACHINE LEARNING

Explainability is critical for cases when an AI-generated-code detector uses its predictions for academic integrity checks, hiring, software review, or security tasks. A binary prediction with no justification is hard to verify and can be dangerously

misleading if the model sees code sufficiently different from what it was trained on. SHAP provides additive feature attributions based on Shapley-value reasoning and can quantify the contribution of each feature to the prediction for a particular instance [12]. The proposed framework therefore uses SHAP on the interpretable fusion layer, which allows reporting of contributions for both individual features and groups of features.

F. COMPARATIVE REVIEW OF REPRESENTATIVE STUDIES

Study	Core approach	Languages / setting	Reported result
GPTS niffer [3]	CodeBERT classifier	ChatGPT code; study-specific settings	Outperformed two general text detectors in its evaluation
Multi AIGC D [7]	Multiple detection models	Python, Java, Go; multiple LLMs/scenarios	Performance varies by language, model and scenario
LAFED [4]	Language-agnostic features + XGBoost	Train: C++, Python, Java; unseen: C, C#, Go, JS, PHP	Official Macro-F1: 0.75209
Archaeology [5]	CodeBERT/GraphCodeBERT/UniXcoder/CodeT5+	SemEval-2026 Task 13	Subtask-A Macro-F1: 0.737
UIT-AMMC [6]	Structure-aware contrastive + statistical fusion	SemEval-2026 Task 13	Macro-F1: 0.802; 3rd place

3. RESEARCH GAP, PROBLEM STATEMENT, AND OBJECTIVES

A. RESEARCH GAP

The literature demonstrates three key shortcomings. First, detectors are largely evaluated on an in-distribution (ID) setting, while in practice, detectors are applied to new programming languages, domains, generators, and hybrid edit workflows. Second, semantic transformer models and traditional stylometric approaches are often considered in isolation, without a clear understanding of how errors might complement or interfere with each other. Third, the explanations for detectors are less developed than their accuracy: a high score on a given metric provides little insight into the underlying factors that contributed to the decision, particularly distinguishing between meaningful structural regularities and artifacts specific to a language.

Accordingly, the current research gap is the lack of explicit comparative protocols analyzing the behavior of surface, structural, and semantic representations in comparable, cross-

language conditions and providing human-usable explanations for features.

B. PROBLEM STATEMENT

Given a source-code snippet c , written in programming language l , determine whether it was written by a human or an AI, with minimal degradation if either l is unseen or the application domain of c is unseen. The detector should return a rationale $E(c)$ that identifies the main evidence indicating the prediction. The system must not use language-specific syntax as evidence, and it should express uncertainty when evidence is inconsistent or weak.

C. RESEARCH OBJECTIVES

- To build a multi-view representation of source code that is reproducible, leverages language-agnostic, AST-based, and semantic features.
- To compare classical ML, transformer-based, and hybrid detectors in terms of Macro-F1, precision, recall, latency, and model size.
- To quantify the cross-language generalization of detectors by training on a subset of languages and testing on unseen languages.
- To analyze the impact of lexical, structural, and semantic features via ablation experiment
- To provide local explanations using SHAP and aggregate explanations using global feature importance.
- To evaluate the calibration of detectors and uncertainty to avoid presenting low-confidence predictions as certain authorship judgments.

D. RESEARCH QUESTIONS

RQ1: How well can different classical and neural methods classify human versus AI-generated source code?

RQ2: Which of the surface, AST, semantic, and fused representations generalize better to unseen programming languages?

RQ3: Does the late fusion of the structural and semantic evidence lead to better cross-language Macro-F1 scores than the best individual representation?

RQ4: Which features and feature groups are the most indicative of AI-generated code?

RQ5: How do various factors affect detection performance in hybrid editing and adversarial transformation scenarios?

E. SCOPE AND DELIMITATIONS

The main problem is the binary classification of human versus AI. The model-family attribution and four-way hybrid/adversarial classification are considered as secondary tasks. The work focuses on the analysis of code content and structural/statistical features derived from it, without using

private user data, IDE interactions, keyboard input or other potential sources of information.

F. THREAT MODEL AND ASSUMPTIONS

The adversary may use a different LLM than those represented in training, paraphrase or refactor generated code, remove comments, rename variables, reformat code, or combine human and AI-written fragments. The detector is assumed to receive only the submitted source code. It is not assumed to have access to the prompt, generation history, model API logs, or version-control provenance.

4. PROPOSED X-AIGCD FRAMEWORK

X-AIGCD is organized as a modular pipeline consisting of four different analytical views: Surface-Style View, Structural View, Semantic View, and an Explainable Fusion View. By designating evidence between three views as complementary, the first three views generate complementary evidence, while the last layer learns how to synthesize such evidence.

A. SYSTEM ARCHITECTURE

Stage	Input / operation	Output
1. Normalization	Source code, language label; encoding and parser validation	Clean code + parser status
2. Surface-Style View	Tokens, characters, whitespace, comments, identifiers, literals	Statistical/stylometric vector
3. Structural View	AST parsing and language-neutral structural measurements	AST feature vector
4. Semantic View	CodeBERT / GraphCodeBERT embeddings	Dense semantic vector
5. Representation Fusion	Standardization + dimensionality control + concatenation	Unified representation
6. Classifier	Logistic Regression, SVM, RF, XGBoost, transformer heads	Human/AI probability
7. Explainability	SHAP on interpretable fusion features	Feature and group attributions
8. Decision	Calibrated probability and uncertainty threshold	Label + confidence + explanation

B. SURFACE-STYLE FEATURES

The surface layer deliberately avoids using language-specific keywords as the main discriminative feature. It includes the number of normalized tokens, characters, and lines, average and variance of line length, comment-to-code ratios and blank-line ratios, indentation statistics, identifier-length statistics, literal density, operator frequency, punctuation density, whitespace entropy, repeated-line ratios, and naming-pattern statistics. Wherever language-specific differences are present in the features (such as in comment or indentation conventions), the feature extractor is designed to normalize them out.

C. STRUCTURAL FEATURES

The structural layer parses the valid code into an AST or an equivalent language parser representation. Candidate features to be extracted could include AST node-type frequencies, AST depth, number of functions/methods, number of classes, branch density, loop density, call density, return density, declaration density, expression complexity, approximate cyclomatic complexity, and AST n-grams. The features are intended to capture the organizational structure of the program and not the specific token content.

D. SEMANTIC FEATURES

Two semantic backbones are proposed for the task comparison, CodeBERT and GraphCodeBERT. The former gives a multilingual Transformer representation of code while the latter takes into account the data-flow information [8], [9]. For the purpose of computational efficiency, the experiment can extract a fixed size pooled embedding and reduce the dimensionality using PCA that is fitted only on the training set.

E. CLASSIFIER COMPARISON

Model	Representation	Purpose
TF-IDF + Logistic Regression	Token/character n-grams	Simple statistical baseline
TF-IDF + SVM	Token/character n-grams	Strong sparse baseline
Random Forest	Surface + AST	Nonlinear interpretable baseline
XGBoost	Surface + AST	Boosted tree baseline
CodeBERT	Semantic embedding	Transformer semantic baseline
GraphCodeBERT	Semantic + data flow	Structure-aware transformer baseline
X-AIGCD	Surface + AST + semantic	Proposed late-fusion model

F. LATE-FUSION STRATEGY

Let $s(c)$, $a(c)$, $e(c)$ be standardized representations of surface, AST and semantic code, respectively. The fused vector is $F(c) = [s(c) \parallel a(c) \parallel e(c)]$. A tree-based meta-classifier such as XGBoost is next trained on $F(c)$ to predict $P(AI|c)$. The model is then calibrated on a held-out validation set via temperature scaling or isotonic regression, depending on the size of the validation set. Finally, the decision for AI-written code is made if the calibrated probability exceeds threshold τ ; else, it is human-written. Note that τ has to be chosen based on the validation set only.

G. WHY LATE FUSION?

Surface and structural features are cheap and easy to reason about. Semantic embeddings capture nontrivial information that is often not captured by simple statistics, but which is

sometimes sensitive to differences in domain or language. Late fusion retains the information from both views, enabling ablative studies that show the value of individual views.

5. RESEARCH METHODOLOGY AND EXPERIMENTAL DESIGN

A. DATASET CONSTRUCTION

The primary evaluation is expected to be conducted on a publicly benchmarked multilanguage setup (e.g., SemEval-2026 Task 13 Subtask A). The given example already contains 500,000 training and 100,000 validation examples, where C++, Python, and Java are the languages used for training. The evaluation can include seen-language/seen- and unseen-domain, seen-language/unseen- and unseen-domain, unseen-language/seen-, and unseen-language/unseen-domain settings [1]. To test robustness, one also could use the MultiAIGCD benchmark, which covers Python, Java, Go, and several LLMs and generation scenarios [7].

B. PREPROCESSING

- Validate file encodings and parser syntaxes.
- De-duplicate exact code snippets across splits.
- Avoid near-duplicates by hashing normalized code and, where possible, using similarity checks.
- Maintain two variants of the code: the raw code to support semantic modeling and the normalized code to support controlled stylistic experiments.
- Do not use generator metadata as an input feature in the primary binary detector.
- Fit vocabulary, scalars, PCA, and other preprocessing components on the training split only.

C. TRAIN/VALIDATION/TEST PROTOCOL

The central experiment should be carried out under four settings: (1) seen language and seen domain, (2) unseen language and seen domain, (3) seen language and unseen domain, and (4) unseen language and unseen domain. Macro-F1 is the preferred metric since it considers the two classes equally important. For all comparisons that involve selecting a model, one has to rely on the validation set; one must not touch the test set before one freezes the final pipeline

D. ABLATION STUDY.

Variant	Removed component	Research purpose
M1	None	Full X-AIGCD
M2	Semantic view	Measure value of surface + AST
M3	AST view	Measure value of surface + semantic
M4	Surface view	Measure value of AST + semantic
M5	Surface only	Pure stylistic baseline
M6	AST only	Pure structural baseline
M7	Semantic only	Transformer baseline

E. ROBUSTNESS TESTS

To evaluate whether the detector learns superficial artifacts, the study should design transformations that preserve program functionality, if possible: rename identifiers, remove comments and whitespace, dead-code insertion, equivalent expression rewriting, statement reordering if possible, and/or partial human editing of generated code. These transformations should be evaluated individually to isolate the impact of a particular transformation.

F. COMPUTATIONAL EFFICIENCY

The following metrics should be recorded: training time, inference latency per snippet, GPU/CPU memory, and model parameter count. This would allow to make a reasonable comparison between the performance of a lighter tree-based model and a heavier transformer-based model. In practice, a smaller model with a slightly worse result on the target data distribution but a better performance on the cross-language tasks and explain ability would be more preferred.

6. EXPLAINABILITY AND DECISION ANALYSIS

A. LOCAL SHAP EXPLANATIONS

The SHAP values ϕ_i are determined by the contribution of a feature i to the model output relative to the baseline outcome [12]. For a given code snippet c , the resulting explanation can be represented by a list of feature contributions sorted in descending order. Among the six considered features, the comment ratio, maximum nesting depth, average identifier length, AST branch density, function count, and semantic embedding components after controlling for dimensionality can be regarded as the most interpretable ones.

Property	Primary protocol
Task	Binary human vs. AI-generated code
Training languages	C++, Python, Java
Unseen evaluation languages	Go, PHP, C#, C, JavaScript
Training size	500K samples
Validation size	100K samples
Primary metric	Macro-F1
Secondary metrics	Precision, Recall, Accuracy, MCC, AUROC, ECE, latency

B. FEATURE-GROUP ATTRIBUTION

Individual embedding dimensions are not intrinsically informative to the programmer. Therefore, we group them into classes: Surface, Structural, and Semantic. We compute class attribution scores by summing absolute SHAP values across all dimensions in a class, expressed as percentages. Thus, in a given case, our interface could state that, for instance, the structural evidence was stronger than the surface statistics, without committing to any particular feature being decisive for authorship.

C. GLOBAL EXPLANATION

In the test set, mean absolute SHAP values are aggregated for each feature to identify which features show consistent influence. The global explanations are compared across programming languages. If a feature is highly influential for Python but nearly irrelevant for Java or Go, then the study should report that the feature is language-sensitive rather than an AI signature.

D. CONFIDENCE AND UNCERTAINTY

An AI-code detector must not report all predictions as equally reliable. The calibrated probability $P(AI|c)$ should instead come with an uncertainty category, and predictions close to 0.5 can be sent for human review. Expected Calibration Error should be reported to determine if predicted probabilities match the actual accuracy of predictions.

E. EXPLANATION QUALITY CRITERIA

Criterion	Operational definition
Fidelity	Explanation should reflect the behavior of the final classifier.
Consistency	Similar inputs should not produce arbitrarily different explanations.
Stability	Small semantics-preserving changes should not radically change the explanation.
Human interpretability	Features should be understandable to a programmer or reviewer.
Actionability	Explanation should indicate what evidence merits manual inspection.

VI. EVALUATION METRICS AND BENCHMARKING

A. CLASSIFICATION METRICS

Accuracy is reported as a supplementary finding, but the main one is Macro-F1. Precision is a ratio of positively classified AI samples that are actually AI-generated, while recall is a ratio of AI samples positively classified by the classifier. In turn, the F1-score is a harmonic mean of precision and recall. Additionally, the Matthews Correlation Coefficient (MCC) is included in the

results since it is informative in case of an imbalanced distribution of classes. The Area Under the Receiver Operating Characteristic Curve (AUROC) is included as it provides ranking information regardless of thresholds.

Published system	Approach	Reported Macro-F1	Evaluation context
LAFED [4]	Language-agnostic structural/stylometric features + XGBoost	0.75209 official	SemEval-2026 Subtask A
LAFED [4]	Best development model	0.7570	1,000-sample test evaluation
UIT-AMMC [6]	Structure-aware contrastive model + statistical late fusion	0.802	Official Subtask A
Archaeology [5]	CodeBERT + GraphCodeBERT + UniXcoder + CodeT5+	0.737	Official Subtask A
UCSC-NLP [16]	Multi-view UniXcoder	0.845	OOD test; official-task setting

B. CROSS-LANGUAGE METRICS

Performance should be reported separately for each test language, then averaged. Let $F1_l$ denote Macro-F1 for language l . The cross-language mean is the arithmetic mean over languages. The generalization gap can be reported as the difference between the mean seen-language Macro-F1 and the mean unseen-language Macro-F1. The lower the gap the better.

C. CALIBRATION METRICS

Expected Calibration Error (ECE) should be calculated by binning the predicted values according to their confidence scores and then comparing the average confidence to the average accuracy of the binned samples. Reliability diagrams can help you visualize the calibration process.

D. STATISTICAL SIGNIFICANCE

Each stochastic model should be trained on multiple random seeds and the mean and standard deviation should be reported. For paired model comparison, bootstrap confidence intervals or paired permutation test can be used. Reporting only the best seed is not recommended as it could inflate the expect performance.

E. FAIR COMPARISON RULE

Published results from different data sets or competition tracks must not be presented as a measure of X-AIGCD. The following comparative table, therefore, refers to it as literature evidence. Original results of the X-AIGCD should be added only after the proposed pipeline has been executed on a fixed set of data and test protocol.

VII. EXPERIMENTAL RESULTS AND DISCUSSION

A. REAL BENCHMARK EVIDENCE

To avoid the fabrication of experimental results, the current study conducts a secondary empirical analysis of published results of SemEval-2026 Task 13 Subtask A. The task of code detection is to conduct binary classification of code that is

entirely written by human or machine with C++, Python, and Java languages used for training and Go, PHP, C#, C, and JavaScript languages used in unseen-language evaluation. The training set officially contains 500,000 samples, of which 238,000 are human-written code and 262,000 are machine-generated code, while the validation set contains 100,000 samples, and the official evaluation metric is Macro-F1 [1].

B. CROSS-LANGUAGE EVIDENCE

LAFED offers particularly valuable per-language evidence. The reported Macro-F1 across C# is 0.7753, JavaScript is 0.7683, Go is 0.64, and PHP is 0.5238 [4]. Across the four reported unseen languages overall, the arithmetic mean is 0.6769 and the best-to-worst range is 0.2515 F1 points. This is particularly compelling evidence that a detector can do well on some unseen languages while doing substantially worse on others. The C is not included in this calculation because the cited abstract does not report a particular score for it.

C. DISTRIBUTION-SHIFT EFFECT

UCSC-NLP reports a 0.993 Macro-F1 on validation and 0.845 on the test set [16]. An absolute difference of 0.148 Macro-F1 (roughly 14.9%) between the validation and the test set. The authors blame this drop on distribution shift in both language and domain; this result directly informs the main thesis of this research: high in-distribution performance is not indicative of robustness in AI-code detection.

D. COMPARATIVE INTERPRETATION

The published systems indicate that both the feature engineering and code-oriented Transformer models can perform competitively. Particularly, LAFED shows that relatively lightweight, language-agnostic structural and stylometric features can achieve 0.75209 official Macro-F1 [4]. The UIT-AMMC achieves 0.802 using a structure-aware contrastive learning and late fusion [6], while the UCSC-NLP reports 0.845 on their test setting using a multi-view UniXcoder system [16]. The Archaeology reports 0.737 after combining several pre-trained code models and robustness techniques [5]. These numbers should not be interpreted as a strict leaderboard comparison outside the original experimental settings, but they do provide concrete evidence for the proposed comparative research direction.

E. RESEARCH IMPLICATION

The evidence supports a plausible hypothesis that combining the surface, structural, and semantic evidence would mitigate the language-specific failure modes. But, the hypothesis has not yet been tested on the official data that can only be accessed after the submission of this paper. Accordingly, this paper distinguishes the published empirical evidence (reported here numerically) from the original X-AIGCD performance, which requires executing the proposed pipeline.

F. ORIGINAL EXPERIMENT TABLE TO COMPLETE

Model	Seen Macro-F1	Unseen - language Macro-F1	OOD Macro-F1	ECE	Latency
TF-IDF + LR	Pending execution	Pending execution	Pending execution	Pending execution	Pending execution
TF-IDF + SVM	Pending execution	Pending execution	Pending execution	Pending execution	Pending execution
Random Forest	Pending execution	Pending execution	Pending execution	Pending execution	Pending execution
XGBoost	Pending execution	Pending execution	Pending execution	Pending execution	Pending execution
CodeBERT	Pending execution	Pending execution	Pending execution	Pending execution	Pending execution
GraphCodeBERT	Pending execution	Pending execution	Pending execution	Pending execution	Pending execution
X-AIGCD (proposed)	Pending execution	Pending execution	Pending execution	Pending execution	Pending execution

G. PRACTICAL CONCLUSION FROM REAL DATA

The most defensible conclusion from the available real evidence is that AI-generated-code detection is feasible but strongly influenced by distribution shift. The results from the per-language LAFED evaluation suggest that there is a 25.15-point range between the best and worst of the four reported unseen languages, while UCSC-NLP has an absolute validation-to-test drop of 14.8. Thus, a publication-quality detector should prioritize cross-language robustness, leakage control, calibration, and explainability instead of only reporting a high in-distribution accuracy.

VIII. LIMITATIONS AND THREATS TO VALIDITY

First, no detector can be expected to remain permanently effective as generation models improve. A future LLM might deliberately optimize for formatting or code that more closely imitate existing human repositories.

Second, dataset composition can create hidden shortcuts. If AI examples and human examples differ systematically in problem difficulty, code length, formatting conventions, or source origin, a classifier might learn these differences as features that distinguish them instead of authorship.

Third, cross-language transfer is not identical to true language independence. Programming languages differ in syntax, idioms, formatting conventions, and ecosystem practices. A language-agnostic feature schema reduces reliance on syntax-specific cues but not entirely dependent on syntax.

Fourth, AST parsing may fail for incomplete or generated snippets. These failures should be reported rather than silently discarded, since invalidity correlates with the generator.

Fifth, SHAP explains behavior of the trained classifier; it does not establish causal proof that a feature is an intrinsic property of AI-generated code. Explanations should therefore be used as analyst-support evidence.

Finally, published benchmark scores are not directly comparable unless the datasets, labels, test splits, preprocessing strategies, model capacities, and evaluation metrics are aligned. The literature comparison in this paper is therefore a contextual analysis rather than a leaderboard claim.

CONCLUSION AND FUTURE WORK

This paper describes X-AIGCD, a comparative and explainable framework for detecting AI-generated source code across programming languages. The framework uses surface-style statistics, AST-derived structural features, and CodeBERT and GraphCodeBERT encoders of semantic code representations. The detection stage employs a late-fusion classifier and SHAP-based explainability module.

The key insight of this work is to consider AI-code detection as a distribution shift problem rather than a straightforward binary classification task. Recent works, including the SemEval-2026 Task 13 and MultiAIGCD benchmark, report significant distribution shifts between languages, domains, generators, and code-generation approaches for AI-code detection [1], [7].

The current paper does not report any numerical results for the proposed framework. Instead, this work provides the necessary experimental details and discusses the related works and evaluation benchmarks. Once the X-AIGCD pipeline is applied to actual data, the results table in Section VIII can be filled with the corresponding measurements, confidence intervals, ablation results, calibration curves, and SHAP analysis.

The future research directions include the detection of hybrid human-AI code, adversarial paraphrasing, and code produced with unseen generator families at the repository level. It is also interesting to evaluate temporal dynamics in AI-code detectors and apply multilingual program analysis techniques. Another compelling research direction is model-family attribution, where one would detect which family of AI models is more likely to produce the given code, not just whether it is AI generated or not. All future research should prioritize code detectors' evaluation in cross-domain settings, calibration, and explainability.

REFERENCES

- [1] D. Orel, D. Azizov, I. Paul, Y. Wang, I. Gurevych, and P. Nakov, "AICD Bench: A Challenging Benchmark for AI-Generated Code Detection," Proc. EACL, 2026. Available: <https://aclanthology.org/2026.eacl-long.325/>
- [2] D. Orel et al., "SemEval-2026 Task 13: Detecting Machine-Generated Code with Multiple Programming Languages, Generators, and Application Scenarios," SemEval-2026 Task 13, 2026. Available: <https://github.com/mbzuai-nlp/SemEval-2026-Task13>
- [3] P. T. Nguyen, J. Di Rocco, C. Di Sipio, R. Rubei, D. Di Ruscio, and M. Di Penta, "GPTSniffer: A CodeBERT-based classifier to detect source code written by ChatGPT," Journal of Systems and Software, vol. 214, 112059, 2024. DOI: 10.1016/j.jss.2024.112059. Available: <https://github.com/MDEGroup/GPTSniffer>
- [4] J. Villate Lemus, "LAFED at SemEval-2026 Task 13: Language-Agnostic Feature Engineering for Cross-Lingual AI-Generated Code Detection," Proc. SemEval, pp. 59–64, 2026. DOI: 10.18653/v1/2026.semeval-1.9. Available: <https://aclanthology.org/2026.semeval-1.9/>
- [5] J.-G. Ispas and S. Nisioi, "Archaeology at SemEval-2026 Task 13: Fine-Tuning Pre-Trained Code Models for AI-Generated Code Detection," Proc. SemEval, pp. 3052–3059, 2026. DOI: 10.18653/v1/2026.semeval-1.383. Available: <https://aclanthology.org/2026.semeval-1.383/>
- [6] C. Pham, M. Nguyen, M. Le, A. Nguyen, and C. Nguyen, "UIT-AMMC at SemEval-2026 Task 13: Exploiting Structural Formatting Signatures for Robust AI-Generated Code Detection," Proc. SemEval, pp. 418–425, 2026. DOI: 10.18653/v1/2026.semeval-1.60. Available: <https://aclanthology.org/2026.semeval-1.60/>
- [7] B. Demirok, M. Kutlu, and S. Mergen, "MultiAIGCD: A comprehensive dataset for AI generated code detection covering multiple languages, models, prompts, and scenarios," Neural Computing and Applications, 2026. DOI: 10.1007/s00521-026-12336-0. Available: <https://link.springer.com/article/10.1007/s00521-026-12336-0>
- [8] Z. Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," Findings of EMNLP, 2020. Available: <https://arxiv.org/abs/2002.08155>
- [9] D. Guo et al., "GraphCodeBERT: Pre-training Code Representations with Data Flow," ICLR, 2021. Available: <https://arxiv.org/abs/2009.08366>
- [10] Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi, "CodeT5+: Open Code Large Language Models for Code Understanding and Generation," Proc. EMNLP, pp. 1069–1088, 2023. DOI: 10.18653/v1/2023.emnlp-main.68. Available: <https://aclanthology.org/2023.emnlp-main.68/>
- [11] "CDUP: Detecting AI Generated Code Using Abstract Syntax Tree Analyst," Procedia Computer Science, vol. 270, pp. 2088–2095, 2025. DOI: 10.1016/j.procs.2025.09.329. Available: <https://www.sciencedirect.com/science/article/pii/S1877050925030029>
- [12] S. M. Lundberg and S.-I. Lee, "A Unified Approach to Interpreting Model Predictions," NeurIPS, vol. 30, pp. 4765–4774, 2017. Available: <https://arxiv.org/abs/1705.07874>
- [13] J. Morrison S, J. Hariharakrishnan, A. Deborah S, and R. S, "Pixel Phantoms at SemEval-2026 Task 13: Exploring Classical and Neural Approaches for AI-Generated Code Detection," Proc. SemEval, pp. 2040–2045, 2026. DOI: 10.18653/v1/2026.semeval-1.256. Available: <https://aclanthology.org/2026.semeval-1.256/>
- [14] Y. Ramesh and T. Wu, "Team Yuwan at SemEval-2026 Task 13: Task-Adaptive Ensemble Strategies for AI-Generated Code Detection," Proc. SemEval, pp. 2089–2097, 2026. DOI: 10.18653/v1/2026.semeval-1.263. Available: <https://aclanthology.org/2026.semeval-1.263/>
- [15] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," Proc. KDD, pp. 785–794, 2016. DOI: 10.1145/2939672.2939785. Available: <https://doi.org/10.1145/2939672.2939785>