Detecting Machine-Generated Code in Multiple Programming Languages and Domains

by

Rakhat Khamitov

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

NAZARBAYEV UNIVERSITY

June 2026

Author .....
Department of Computer Science
April 11

Certified by .....
Zohaib Latif
Assistant Professor, School of Engineering and Digital Sciences
Thesis Course Coordinator

Certified by .....
Michael Lewis
Associate Professor, School of Engineering and Digital Sciences
Thesis Course Coordinator

Accepted by .....
Yelyzaveta Arkhangelsky
Dean, School of Engineering and Digital Sciences

Detecting Machine-Generated Code in Multiple Programming Languages and Domains

by

Rakhat Khamitov

Submitted to the Department of Computer Science
on April 11, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

The widespread adoption of large language models for software development has created an urgent need for reliable detection of machine-generated code. This thesis studies *machine-generated code detection* under realistic conditions where code varies across programming languages, application domains, model families, and generation strategies.

The experiments are grounded in SemEval-2026 Task 13, Subtask A, an externally organized benchmark for machine-generated code detection. The benchmark used in this thesis contains training and validation data in three programming languages and evaluation data spanning eight programming languages, multiple domains, unseen generator families, adversarial examples, and human–AI co-authored settings.

This thesis contributes a systematic comparison of lexical, structural, neural-embedding, metric-learning, comment-embedding, and stylometric approaches under in-distribution and out-of-distribution evaluation. The results show that high in-distribution validation performance does not predict robust detection: direct classifiers reach validation Macro-F1 above 0.94 but fall to 0.24–0.41 OOD Macro-F1. The strongest configuration, a comment-embedding SVM, achieves 0.671 OOD Macro-F1 on the labeled diagnostic test sample and a 0.638 Kaggle submission score, suggesting that comment style is a more stable cross-language signal than code-surface patterns alone.

Keywords: AI-generated code detection; large language models; out-of-distribution generalization; comment embeddings; SemEval.

Thesis Course Coordinator: Zohaib Latif
Title: Assistant Professor, School of Engineering and Digital Sciences

Thesis Course Coordinator: Michael Lewis
Title: Associate Professor, School of Engineering and Digital Sciences

Acknowledgments

I would like to express my sincere gratitude to my thesis course coordinators, Prof. Michael Lewis and Prof. Zohaib Latif, for their guidance and support throughout the thesis submission process.

I am especially grateful to Prof. Lisa Chalaguine, who supervised this research from its inception and whose mentorship, encouragement, and technical insight shaped this work in fundamental ways.

Contents

List of Tables

Chapter 1

Introduction

Large language models (LLMs) are now widely used to generate and edit source code in IDEs and AI-agentic workflows. Tools such as GitHub Copilot [15], Cursor [5], and Claude Code [1] can suggest entire functions, complete algorithmic tasks, and assist with debugging. While those tools can improve developer productivity, they also raise concerns around academic integrity, software security, and intellectual property [20, 21]. In educational settings, students may submit AI-generated code as their own work. In professional contexts, LLM-generated code has been shown to contain security vulnerabilities [2], and may reproduce copyrighted material from training data [30]. In many settings, it is therefore important to determine whether a particular code snippet was generated using AI tools.

1.1 Motivation

Machine-generated code differs from natural language text. Code satisfies strict syntactic constraints and follows language-specific patterns. Modern LLMs can mimic human coding style and be further humanized through rewriting or preference tuning. A combination of both makes robust detection challenging, especially when detectors face unseen programming languages, coding domains, and generators.

Prior work on machine-generated code detection has largely been evaluated in narrow settings: a single programming language, a limited set of LLM generators,

or a single coding domain [12, 17]. While these studies report strong in-distribution performance, Orel et al. [20] showed that detectors suffer significant performance drops when evaluated on unseen languages and domains. The externally organized SemEval-2026 Task 13 benchmark [14] directly addresses this gap by providing a standardized, community-wide benchmark derived from the DroidCollection resource suite [21]. DroidCollection provides the broader source corpus, while the Subtask A data used in this thesis trains on three programming languages and evaluates on eight languages across seen and unseen domains. The training data covers only three programming languages and only the algorithmic problem-solving domain, while the evaluation sets introduce unseen languages, unseen domains, and unseen LLM generators. This makes it a realistic and challenging testbed for studying out-of-distribution generalization in machine-generated code detection.

1.2 Problem Statement

This thesis investigates the following research questions:

- **RQ1:** Can statistical and structural features of code – such as AST-based metrics, variable naming patterns, and formatting statistics – reliably detect machine-generated code across different unseen programming languages and domains?
- **RQ2:** Do neural models pre-trained for code understanding tasks generalize better than feature-based approaches under distribution shift?
- **RQ3:** Does combining structural features with neural embeddings improve out-of-distribution detection performance?

1.3 Contributions

This thesis makes the following contributions:

- A structured review of related work on machine-generated code detection and its relationship to machine-generated text detection, including identification of the research gap that SemEval-2026 Task 13 directly addresses.
- An empirical comparison of statistical, structural (AST-based), and neural feature representations under in-distribution and out-of-distribution conditions on the DroidCollection benchmark.
- An analysis of detector failure modes under language and domain shift, including the effect of decision threshold calibration on out-of-distribution performance for structural models.

1.4 Thesis Organization

Chapter 1 introduces the problem and motivation. Chapter 2 reviews related work. Chapter 3 presents datasets, methodology, and experimental setup. Chapter 4 presents results and analysis, and Chapter 5 concludes with discussion and future directions.

Chapter 2

Related Work

This chapter reviews recent literature in machine-generated code detection and organizes the related work into three categories: large language models for code generation, machine-generated text detection, and machine-generated code detection.

2.1 Large Language Models for Code Generation

Large language models (LLMs) have recently become indispensable tools for automated code generation. Models such as ChatGPT [18], StarCoder [26], CodeLlama [22], and Claude [3] have been trained on vast datasets of source code and natural language instructions from various domains. Fine-tuning and prompting these code models proved their effectiveness in solving different downstream software engineering tasks.

Today, several integrated development environments, such as Cursor [5] and VS Code with GitHub Copilot [15], integrate AI-powered code assistance, while autonomous software engineering agents such as Codex [19] and Claude Code [1] are emerging to let developers code faster by writing natural language instructions.

However, widespread adoption of LLM-based coding generation has raised significant concerns. Recent studies [2] showed that 35% of code snippets generated by GitHub Copilot contain security vulnerabilities of various types. In addition, intellectual property violations have emerged, as LLMs may generate copyrighted code

from their training data [30]. The use of AI tools for code generation raises academic integrity concerns in educational contexts. These concerns underscore the need for detecting AI-generated code.

2.2 Machine-Generated Text Detection

To understand the problem of machine-generated code detection, it is first necessary to examine the broader field of machine-generated text detection. Early research on machine-generated content detection has focused on detecting outputs from specific models like GPT-2, using statistical features and perplexity-based methods. Tools like GPTZero [8], Sapling [23], and DetectGPT [16] emerged for detecting machine-generated text.

Wang et al. [29] introduced M4, a comprehensive multi-generator, multi-domain, and multilingual machine-generated text detection benchmark. They evaluated text detection performance across various LLMs (GPT-3.5, GPT-4, PaLM), diverse domains (news, creative writing, and scientific text), and multiple languages. They found that it is challenging for detectors to generalize well on unseen domains and generators, thus underscoring the need for robust detectors. However, source code is different from generic text. Code follows formal grammars and syntax, which vary across different programming languages. Jian Wang et al. [27] and Suh et al. [25] have shown that text detectors such as GPTZero often fail to detect machine-generated code. This important observation highlights the limitations of applying text detection methods to the coding domain, thus motivating the development of code-specific detection methods.

2.3 Machine-Generated Code Detection

Source code carries stylistic fingerprints that reflect its authorship. Patterns in variable naming, comment density, whitespace usage, and syntactic structure differ systematically between human and machine-generated code [13, 20]. Gurioli et al. [10]

demonstrated that a single transformer-based classifier trained on stylometric features of code can detect AI-written programs across ten different programming languages, achieving an average accuracy of 84.1%. However, such evaluations are conducted within a fixed set of generators and a controlled dataset, leaving open the question of how stylometric approaches perform when the programming language, coding domain, or generator family is unseen at training time. Furthermore, prior work has not systematically compared statistical, structural, and neural feature representations under such out-of-distribution conditions, making it unclear which type of signal is most robust to language and domain shift. SemEval-2026 Task 13 [14] directly addresses this gap by providing a standardized benchmark for evaluating detectors under realistic distribution shift.

Early works on machine-generated code detection, such as Idilu et al. [12], focus on decision tree-based classifiers trained on code-level statistical features, which demonstrated effectiveness for authorship identification. However, these approaches relied more on hand-crafted features, failing to capture semantic properties of code. Other work, such as GPTSniffer [17], fine-tuned CodeBERT [6] to classify code snippets as LLM-generated or human-written. On their evaluation dataset of Java code from GitHub, GPTSniffer achieved over 95% accuracy. However, their evaluation was limited to code generated by ChatGPT and only the Java programming language, raising questions about generalization to other generators and programming languages.

To address these generalization challenges, Orel et al. [20] introduced CoDet-M4, extending the M4 framework [29] to code detection. CoDet-M4 evaluated detection across three dimensions: programming languages, code generators, and domains. They performed evaluation on out-of-domain scenarios such as authorship detection, hybrid authorship detection, and generalization to unseen domains, generators, and programming languages. Their extensive experiments showed that CoDet-M4 works effectively at distinguishing human-written and machine-generated code.

Building on CoDet-M4, Orel et al. [21] released Droid, a comprehensive resource suite for AI-generated code detection. DroidCollection consists of over one million code samples generated by 43 different generators, over three coding domains, and

seven programming languages. In addition to purely AI-generated code, their resource suite also includes human-AI co-authored code and adversarial samples that were generated to evade detectors.

To directly address these generalization challenges at community scale, SemEval-2026 Task 13 [14] was organized, a shared task on detecting machine-generated code across multiple programming languages, coding domains, and generator families, providing a standardized evaluation benchmark for the research community.

Chapter 3

Datasets and Methodology

This chapter describes the datasets and methodology used to study machine-generated code detection for SemEval-2026 Task 13 [14], Subtask A. The chapter is divided into three parts: datasets, methodology, and experimental setup.

3.1 Dataset

3.1.1 SemEval-2026 Task 13

SemEval-2026 Task 13 [14] evaluates detection of machine-generated code across multiple languages, domains, and generator families. This thesis focuses on Subtask A, which is a classic binary classification (human vs machine) task. The workshop task guidelines use macro-F1 as the primary evaluation metric.

There are data and model restrictions to this task. Additional training data is not allowed, only official training and evaluation sets must be used. It is also not permitted to use models that have been fine-tuned specifically for machine-generated code detection. However, general-purpose and code-oriented pre-trained models can be used [14].

Table 3.1 shows different evaluation settings for subtask A. Different evaluation settings allow detector robustness to be evaluated under language, domain, and generator shifts.

Setting	Programming Language	Domain
Seen languages, seen domain	C++, Python, Java	Algorithmic
Unseen languages, seen domain	Go, PHP, C#, C, JavaScript	Algorithmic
Seen languages, unseen domains	C++, Python, Java	Research, Production
Unseen languages, unseen domains	Go, PHP, C#, C, JavaScript	Research, Production

Table 3.1: Subtask A evaluation settings in the official SemEval-2026 Task 13 description [14].

3.1.2 Dataset Overview

The Subtask A dataset consists of training, validation, and test splits. Table 3.2 provides an overview of the dataset. The official evaluation is done on a Kaggle test set of 500,000 samples. Since this set has no labels, the task organizers also provided a 1,000-sample subset of the test set, which includes labels. This smaller labeled set is used for error analysis and for examining how detectors handle different programming languages, generators, and domains.

Split	Samples	Labels	Languages
Training	500,000	Yes	Py, C++, Java
Validation	100,000	Yes	Py, C++, Java
Test Set Samples	1,000	Yes	8 languages
Official Test	500,000	**No**	8 languages

Table 3.2: Overview of the SemEval-2026 Subtask A dataset splits.

3.1.3 Dataset Distributions

The training data exhibits a significant imbalance in terms of programming languages. As shown in Table 3.3, Python accounts for 91.5% of the total training set, while C++ and Java represent only 4.7% and 3.9% respectively. This is a major challenge because the official test sets contain five additional languages (Go, PHP, C#, C, and JavaScript) that are not present in the training data. This requires the detectors to generalize across syntax and structures that were never seen during the training phase.

Language	Count	Share (%)
Python	457,306	91.5%
C++	23,392	4.7%
Java	19,302	3.9%
Total	**500,000**	**100%**

Table 3.3: Language distribution within the Subtask A training set.

There is also a notable shift in label distribution between the splits. While the training and validation sets are roughly balanced, the 1,000-sample `test_sample` is 77.7% human-authored code, as detailed in Table 3.4. This shift is critical for evaluating whether the detectors remain robust and maintain a low false-positive rate when the proportion of machine-generated code is significantly lower than what was encountered during training.

Label	Train Share (%)	Test Sample Share (%)
Human (0)	47.7%	77.7%
Machine (1)	52.3%	22.3%

Table 3.4: Comparison of label distributions between training data and the diagnostic test sample.

3.2 Methodology

The methodology part of this thesis aims to address the challenge of out-of-distribution (OOD) generalization. Motivated by prior work on machine-generated code detection [12, 20], this thesis evaluates three main approaches to feature extraction: code-level statistical features, features derived from Abstract Syntax Trees (AST), and neural embeddings.

3.2.1 Feature Extraction and Representation

It is important to identify effective features that can be used to detect machine-generated code. Following the feature taxonomy established in prior work [12, 20], four representation families are proposed. This allows for an ablation study to determine which features are most sensitive to language and generator shifts.

Feature Family	Description	Role in OOD Robustness
Lexical (TF-IDF)	Character and word-level N-gram frequencies.	Baseline; captures surface-level style but prone to overfitting.
Handcrafted Statistical	Formatting metrics (whitespace, line length, comment density).	Captures stylistic artifacts common in specific LLM families.
Structural (AST)	Features derived from the Abstract Syntax Tree (depth, node types).	Language-agnostic logic; should remain stable across shifts.
Neural Embeddings	Dense vectors from pre-trained models (CodeT5+ [28], UniXcoder [9]).	Captures high-level contextual and semantic relationships.

Table 3.5: Taxonomy of feature representations used in this study.

3.2.2 Preprocessing Pipeline

To ensure the models focus on meaningful patterns rather than superficial artifacts, a deterministic preprocessing pipeline is applied. Two primary input variants are evaluated: raw code and normalized code. The raw code preserves all original formatting, comments, and whitespace, which may contain stylistic "fingerprints" of specific generators. On the other hand, the normalized code undergoes a deep cleaning process that removes all comments, standardizes whitespace, and canonicalizes literals. A comparison of these two variants serves as an ablation study to determine whether the detectors rely on underlying structural logic or brittle formatting shortcuts that often characterize machine-generated code.

3.2.3 Baseline Models

To set a starting point for the experiments several baseline models are used, as summarized in Table 3.6. First, TF-IDF with LinearSVC and Logistic Regression. Second, Random Forest and SVM models are used with handcrafted features to see how basic code-level statistics might perform. Third, XGBoost is used to test different AST features and embeddings. Finally, CodeBERT [6] is fine-tuned to provide a deep learning baseline. In Subtask A, CodeBERT is considered to be a baseline model.

Using these different models it is possible to track if performance of the detectors depends on specific algorithms or the features being used.

Model Group	Algorithm	Feature Set
Lexical Baselines	LinearSVC, Logistic Regression	TF-IDF (N-grams)
Statistical Baselines	Random Forest, SVM	Handcrafted Statistics
Structural Baselines	XGBoost	AST Features
Neural Baselines	CodeBERT	Sequence Embeddings

Table 3.6: Summary of baseline models and their corresponding feature representations.

3.2.4 Detection Pipeline

The detection pipeline evaluated in this work is divided into two parts. The first part is a structural pipeline that uses multiple AST features with a LinearSVC model. This model uses "balanced" class weights and focuses on code structure patterns that stay the same across different AI generators.

The second part uses a neural embedding pipeline. In this process, UniXcoder [9] and CodeT5+ [28] are fine-tuned on a balanced subset called `diverse_train`. In this thesis, `diverse_train` denotes a 60,000-sample subset of the official training data, constructed with 10,000 human-authored and 10,000 machine-generated examples from each of the three training languages (Python, C++, and Java). This subset was constructed by stratified sampling over programming-language and class labels. It reduces the extreme Python dominance of the full 500,000-sample training set while preserving class balance within each included language. After fine-tuning, the embeddings are extracted from these models and used to train the classifiers. These classifiers are tested both on their own and combined with AST features.

3.2.5 Neural Embedding Pipeline

The neural embedding pipeline decouples representation learning from classification. Rather than fine-tuning an end-to-end classifier, a pre-trained code model is fine-tuned on `diverse_train` to produce high-quality embeddings, and a separate linear

classifier is then trained on the extracted vectors. This two-stage design allows the classifier to be trained and threshold-tuned independently, without re-running GPU inference.

Two models are fine-tuned: UniXcoder [9] (`microsoft/unixcoder-base`, 125M parameters) and CodeT5+ [28] (`Salesforce/codet5p-110m-embedding`, 110M parameters). UniXcoder is a cross-modal pre-trained model supporting code understanding across multiple programming languages. CodeT5+ is an encoder pre-trained specifically for code representation via contrastive learning. UniXcoder uses a single linear projection head; CodeT5+ uses a two-layer head with hidden dimension 256 and 0.1 dropout between layers. Both models are trained with AdamW optimisation and mixed-precision training (FP16) on the `diverse_train` split (54k training samples).

The rationale for using `diverse_train` rather than the full 500k training set is that the full set is 91.5% Python, causing fine-tuned representations to overfit to Python-specific formatting patterns. The balanced, three-language composition of `diverse_train` encourages the models to learn generator-agnostic representations.

After fine-tuning, classification heads are discarded and penultimate-layer representations are extracted as fixed-dimensional embeddings: 768-dimensional for UniXcoder and 256-dimensional for CodeT5+. These vectors are used as input features for a LinearSVC classifier. The classifier uses the same threshold-calibration procedure and a grid search over $C \in \{0.01, 0.05, 0.1, 0.5, 1.0\}$.

3.2.6 Multiple Instance Learning with Chunk-based Pooling

A practical limitation of standard transformer encoders is the maximum sequence length of 512 tokens. For many code files, particularly those from production and research domains, relevant discriminative patterns may appear beyond this truncation boundary. To address this, a Multiple Instance Learning (MIL) approach is adopted, treating each code file as a bag of overlapping token chunks.

Each sample is divided into overlapping windows of 400 tokens with a stride of 200, yielding a maximum of 8 chunks per sample. The fine-tuned UniXcoder

model encodes each chunk independently, producing one `[CLS]` embedding per chunk. In transformer encoders, `[CLS]` denotes the special classification token whose final hidden state is commonly used as a compact representation of the sequence. The final sample representation is obtained by element-wise *max-pooling* over all chunk embeddings, resulting in a single 768-dimensional vector.

Pooling is a standard neural-network operation for reducing a set of activation vectors to one fixed-size representation [7]. Element-wise max-pooling keeps the largest value observed for each embedding dimension across all chunks, whereas mean-pooling would average the corresponding values across chunks. Max-pooling is chosen here because if any chunk of a code file contains strong AI-generated patterns, that signal should dominate the final representation. Mean-pooling would dilute such signals by averaging with chunks that appear more neutral, reducing sensitivity to locally concentrated evidence.

The MIL embeddings are concatenated with AST features (130 dimensions) and CodeT5+ single-chunk embeddings (256 dimensions), forming a 1,154-dimensional feature vector. This combined representation is classified using a LinearSVC with $C = 0.01$ and threshold calibration.

3.2.7 Metric Learning with ModernBERT

ModernBERT (`answerdotai/ModernBERT-base`, 149M parameters) supports a context window of up to 8,192 tokens, making it attractive for long code files. As a preliminary step, a direct classification experiment is conducted: ModernBERT with a linear head is fine-tuned on `diverse_train` with a maximum sequence length of 1,024, achieving 0.988 validation Macro-F1. However, OOD Macro-F1 on the labeled `test_sample` collapses to 0.243 — lower than even the TF-IDF baseline. This result suggests that the OOD failure of direct classifiers is not caused only by sequence truncation but also by shortcut learning of training-set patterns, regardless of model architecture or context window size.

To address this, a metric learning approach using batch-hard triplet loss is explored [11]. Rather than predicting class labels, the model is trained to produce

an embedding space where human-written and machine-generated code are well-separated. A projection head consisting of Linear($768 \to 256$), GELU activation, and L2 normalisation is added on top of ModernBERT. A `BalancedBatchSampler` ensures each mini-batch contains 8 samples per class, guaranteeing valid triplet formation. The batch-hard objective mines the hardest positive and hardest negative within each batch:

$$\mathcal{L}_{\text{triplet}} = \max(0,\ m + d(a, p^*) - d(a, n^*)) \,,$$

where a is the anchor, p^* is the hardest positive, n^* is the hardest negative, $m = 0.3$ is the margin, and $d(\cdot, \cdot)$ denotes Euclidean distance in the projected space.

The intuition behind triplet loss in this setting is that it prevents the encoder from exploiting generator-specific formatting shortcuts. Instead of distinguishing individual generators seen during training, the model must learn what it means for code to be "AI-like" in a generator-agnostic sense. Embeddings from the trained model are extracted and classified using a LinearSVC with threshold calibration.

3.2.8 Comment Embedding Detection

The comment embedding approach is motivated by an empirical observation: the most consistently distinguishable characteristic of AI-generated code is not its structure, syntax, or formatting, but rather the style of its inline comments and docstrings. AI-generated code tends to include verbose, instructional, and conversationally toned comments — phrases such as *"This function computes..."*, *"Note:..."*, or *"Returns the result of..."* — regardless of the programming language or the specific generator.

Comments are extracted using language-aware regular expressions that capture single-line comments, multi-line block comments, and docstrings for all eight evaluation languages. All extracted comments from a given sample are concatenated into a single string. Samples with no comments are represented by an empty string.

These strings are encoded using `sentence-transformers/all-mpnet-base-v2`, a general-purpose sentence embedding model trained on diverse natural language

data (768-dimensional output). A code-specific model is intentionally not used here. Code-specific models are pre-trained to capture syntactic and structural properties of code; in the experiments reported in Chapter 4, these models show weaker OOD Macro-F1 when trained on the Python-dominated training distribution. The general NLP model, by contrast, captures the conversational tone and linguistic style of AI-generated comments without directly modeling code structure, making it less sensitive to programming-language syntax. Embeddings are L2-normalised before classification.

The classifier is a LinearSVC trained on `diverse_train` embeddings ($60{,}000 \times 768$ dimensions), with a grid search over $C \in \{0.01, 0.05, 0.1, 0.5, 1.0\}$ and threshold calibration over a 400-point grid on the decision function range. LinearSVC is preferred over kernel SVMs for computational reasons: training and prediction on 60k and 500k samples respectively would require hours with an RBF kernel, whereas LinearSVC completes training in approximately 380 seconds on CPU.

3.2.9 Stylometric Feature Detection

The stylometric method extracts 23 language-agnostic string-level features that require no parser, making the approach robust to languages not seen during feature engineering. Features are organised into five groups: (1) line-level statistics, including blank line ratio, average and maximum line length, and coefficient of variation of line lengths; (2) indentation patterns, including average, standard deviation, maximum, and coefficient of variation of indent depth, as well as tab and space usage ratios; (3) character-level features, including Shannon entropy and ratios of alphabetic, uppercase, digit, and special characters; (4) token-level features, including average and maximum token length, type-token ratio, and total token count; and (5) comment density, including comment line ratio and overall comment presence ratio.

Two LLM-specific marker features are also included: `has_llm_marker`, which records whether any marker phrase is present, and `llm_marker_count`, which counts marker occurrences. Both are derived from 11 regular expression patterns matching phrases commonly found in AI-generated comments, such as *"sure, here's"*, *"this func-*

tion", and *"note:"*. These markers exploit the same signal as the comment embedding approach but in a lightweight, interpretable form that requires no embedding model.

An RBF-kernel SVM with balanced class weights and threshold calibration is used as the classifier, trained on `diverse_train`. The stylometric approach requires no GPU and no pre-trained models, making it a fast and interpretable alternative for resource-constrained environments.

3.2.10 Evaluation

The primary metric for evaluation is macro-F1, with accuracy serving as a secondary measure. Macro-F1 is appropriate for this task because it weights the human and machine classes equally, rather than allowing the larger class in a shifted evaluation split to dominate the score. This is important because the labeled `test_sample` is substantially more human-heavy than the training split (Table 3.4), and because the official shared task uses macro-F1 as its primary ranking metric [14]. For a given class c, the F1-score is calculated using precision (P_c) and recall (R_c), following standard classification-evaluation practice [24]:

$$F1_c = \frac{2P_cR_c}{P_c + R_c}, \qquad \text{Macro-F1} = \frac{1}{2}\left(F1_{\text{human}} + F1_{\text{machine}}\right).$$

Performance is reported across three different scenarios: standard validation (IID), validation on unseen generators, and external evaluation on the labeled `test_sample`. Throughout the results chapter, "OOD Macro-F1" refers to macro-F1 on this labeled `test_sample` unless explicitly stated otherwise. This allows for a clear comparison between "seen" and "unseen" data. Additionally, final performance scores for the official test set are obtained by submitting model predictions to the Kaggle competition platform, as the labels for this split are not publicly available.

3.3 Experimental Setup

3.3.1 Training Configuration

The training process for probabilistic classifiers focuses on minimizing binary cross-entropy loss, a standard objective for binary neural classification [7]:

$$\mathcal{L}_{\text{BCE}}(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log f_\theta(x_i) + (1-y_i)\log(1-f_\theta(x_i))\right].$$

For linear-margin SVMs, the optimization follows the regularized hinge loss [4]:

$$\min_{w,b}\ \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}\max\left(0, 1 - y_i^{\pm}(w^\top x_i + b)\right),$$

where $y_i^{\pm} \in \{-1, +1\}$. These standard loss functions ensure that the models are optimized for binary classification performance.

Model Family	**Hyperparameter Settings**
Random Forest	100 estimators; handcrafted feature variants (53 and 58 features).
SVM	$C \in \{0.01, 0.05, 0.1, 0.5, 1.0\}$; linear and RBF kernels; balanced class weights.
XGBoost	Feature ablation sets; tuned depth, estimators, and learning rate.
CodeBERT	Sequence length 256/512; early stopping; best checkpoint by validation macro-F1.
UniXcoder	Batch 16; lr 2×10^{-5}; max length 512; 3 epochs; AdamW; FP16.
CodeT5+	Batch 32; lr 3×10^{-5}; max length 512; early stopping; AdamW; FP16.
ModernBERT (direct)	Batch 16; lr 2×10^{-5}; max length 1024; 3 epochs; AdamW; FP16.
ModernBERT (triplet)	Batch 16 (8 per class); lr 2×10^{-5}; max length 512; 5 epochs; margin 0.3; AdamW; FP16; grad clip 1.0.
Comment Embeddings	`all-mpnet-base-v2`; batch 256; L2 normalised; LinearSVC C grid search.

Table 3.7: Model-selection and tuning parameters used in the experiments.

The primary comparison uses a default decision threshold of $\tau = 0.5$. Sensitivity analysis is conducted by searching for the optimal threshold over a range of values,

which is reported separately to maintain the integrity of the primary results. For neural models, early stopping and macro-F1-based checkpoint selection are used to ensure the best possible generalization.

3.3.2 Hardware and Software Environment

The experiments were conducted in a Linux environment using a high-performance DGX server equipped with NVIDIA V100 GPUs. This infrastructure provided the necessary computational power for the large-scale training and fine-tuning tasks required by the study.

The software stack is based on the Python machine learning ecosystem. Classical models were implemented using `scikit-learn`, while gradient-boosted trees were built with the `XGBoost` library. Neural model fine-tuning and embedding extraction were performed using `PyTorch` and the Hugging Face `Transformers` library.

Chapter 4

Results and Analysis

4.1 Baseline Model Results

The initial experiments established a performance baseline using standard lexical and neural models. Table 4.1 shows that on the internal validation set (IID), all models achieve near-perfect scores, with Macro-F1 values above 0.94. This confirms that the models successfully learned the features of the training set.

However, a dramatic performance drop is observed when these same models are applied to the out-of-distribution (OOD) test sample. The classical Random Forest and SVM models, which rely on text-level statistics, saw their F1 scores fall to between 0.30 and 0.38. Similarly, the CodeBERT neural baseline, despite its high validation Macro-F1 of 0.988, only reached an OOD Macro-F1 of 0.40. This gap highlights that high performance on training data does not guarantee the ability to detect machine-generated code in new languages or from different generators.

Model Family	**Val Acc**	**Val Macro-F1**	**OOD Acc**	**OOD Macro-F1**
Lexical (RF/SVM)	0.968	0.967	0.328	0.324
Neural (CodeBERT)	0.988	0.988	0.412	0.403

Table 4.1: Comparison of baseline performance on IID validation vs. OOD test sets.

4.2 Detection Pipeline Results

Since the default threshold of $\tau = 0.5$ may not be optimal under distribution shift, sensitivity analysis over the decision threshold is conducted to identify configurations that improve OOD performance.

As shown in Table 4.2, using AST-only features with a Linear SVM provided a more stable representation of code logic. While the default performance was still low, the threshold sensitivity analysis in Table 4.3 shows that these models respond much better to calibration. By shifting the decision threshold from 0.5 to 2.15, the AST-based model's OOD Macro-F1 increased from 0.31 to 0.54. This represents a gain of +0.23, the highest improvement across all tested configurations.

Model / Features	**Val Macro-F1**	**OOD Macro-F1**
XGBoost (Embeddings only)	0.8848	0.2733
XGBoost (AST only)	0.9633	0.3246
XGBoost (AST + Embeddings)	0.9687	0.3051
Linear SVM (AST + Embeddings)	0.9250	0.3356
Ensemble (XGB + SVM)	**0.9689**	**0.2978**

Table 4.2: Performance of AST-centered models using default decision thresholds.

Model Configuration	**Default F1**	**Tuned F1**	**Threshold**	**Gain**
SVM (AST+Embed, C=0.01)	0.2963	0.5232	2.60	+0.227
SVM (AST+Embed, C=0.1)	0.3044	0.5321	2.85	+0.228
SVM (AST only, C=0.1)	**0.3142**	**0.5494**	**2.15**	**+0.235**
SVM (AST only, C=1.0)	0.3143	0.5487	2.15	+0.234

Table 4.3: Impact of threshold calibration on out-of-distribution (OOD) performance.

4.3 Neural Embedding Pipeline Results

Table 4.4 summarises the performance of the neural embedding pipeline on IID validation and the OOD `test_sample`. Fine-tuned CodeT5+ embeddings achieve the best result within this group, with an OOD Macro-F1 of 0.555 and a Kaggle submission score of 0.533. This represents a substantial improvement over the CodeBERT direct classifier (OOD Macro-F1: 0.403), confirming that the two-stage embedding approach generalises better than end-to-end fine-tuning.

Combining AST features with neural embeddings does not consistently improve performance. While it sometimes improves OOD Macro-F1 on the `test_sample`, it degraded Kaggle performance, suggesting that the combination overfits to the specific distribution of the 1,000-sample diagnostic set rather than the full 500k official test set.

Method	Feature Set	Val Macro-F1	OOD Macro-F1	Kaggle
CodeT5+ embed + SVM	Neural (256-dim)	0.965	0.555	0.533
UniXcoder embed + SVM	Neural (768-dim)	0.984	0.516	0.506
AST + CodeT5+ + SVM	AST + Neural	0.969	0.532	—
MIL + AST + SVM	MIL + AST (1154-dim)	—	0.602	∼0.590

Table 4.4: Neural embedding pipeline performance on IID validation vs. OOD test sets.

The MIL approach achieves the best OOD Macro-F1 within this group at 0.602, with an estimated Kaggle score around 0.590. The improvement over standard CodeT5+ (0.555) and UniXcoder (0.516) confirms that truncating code at 512 tokens discards meaningful discriminative signal in longer files. Max-pooling over overlapping chunks allows the model to capture AI-generated patterns wherever they appear in a file, not only in its first 512 tokens.

4.4 Metric Learning Results

Table 4.5 compares the ModernBERT direct classifier with the ModernBERT triplet loss model. The direct classifier achieves the highest in-distribution validation Macro-F1 of all tested methods (0.988), yet collapses to 0.243 OOD — lower than even the TF-IDF baseline. This confirms that the OOD failure of direct classifiers is not caused by truncating long code files but by shortcut learning, regardless of model architecture or context window size.

Replacing cross-entropy with batch-hard triplet loss substantially recovers OOD performance to 0.486. The metric learning objective forces the encoder to produce a more generator-agnostic embedding space. However, the recovery is partial: the limited diversity of the training set — three languages and one coding domain — constrains the quality of the learned metric.

Model	Objective	Val Macro-F1	OOD Macro-F1
ModernBERT Direct	Cross-entropy	0.988	0.243
ModernBERT Triplet	Batch-hard triplet	—	0.486

Table 4.5: Comparison of ModernBERT direct classification vs. metric learning.

4.5 Comment Embedding and Stylometric Results

The comment embedding approach achieves the best OOD and Kaggle performance of all methods evaluated. Using $C = 0.1$ for the LinearSVC, the model achieves an OOD Macro-F1 of 0.671 and a Kaggle submission score of 0.638. Using $C = 0.05$ yields a marginally higher local OOD Macro-F1 of 0.672, though this configuration had not yet been submitted at the time of writing. Results are shown in Table 4.6.

The stylometric method, which uses only 23 language-agnostic string-level features without any pre-trained model, achieves an OOD Macro-F1 of 0.580. While lower than the comment embedding approach, this result is notable given the method's simplicity: it requires no GPU, no parser, and no embedding model. The LLM marker features contribute meaningfully to this performance, confirming that lexical cues in comments and string literals are a strong indicator of machine generation.

Method	Features	Val Macro-F1	OOD Macro-F1	Kaggle
Comment Embed + SVM (C=0.1)	all-mpnet-base-v2 (768-dim)	—	0.671	**0.638**
Comment Embed + SVM (C=0.05)	all-mpnet-base-v2 (768-dim)	—	0.672	—
Stylometric + RBF-SVM	23 string features	—	0.580	—

Table 4.6: Comment embedding and stylometric method performance.

4.6 Overall Method Comparison

Table 4.7 presents a unified comparison of all ten methods, ordered by OOD Macro-F1. A consistent pattern emerges: methods that optimise directly for training-set classification collapse severely under distribution shift, regardless of whether they use lexical features, handcrafted statistics, or neural classification heads. Methods that either capture generator-agnostic structural properties (AST, stylometric), represent code at a semantic embedding level (neural embedding, MIL), or target the most

linguistically stable aspect of AI-generated code (comments) achieve substantially better generalisation.

Method	Feature Type	Val Macro-F1	OOD Macro-F1	Kaggle
Comment Embed + SVM	NLP embeddings (comments)	—	**0.671**	**0.638**
MIL + AST + SVM	MIL + structural	—	0.602	∼0.590
Stylometric + RBF-SVM	23 string features	—	0.580	—
CodeT5+ embed + SVM	Neural (256-dim)	0.965	0.555	0.533
SVM (AST only, tuned)	AST (121 feat.)	0.963	0.549	0.467
UniXcoder embed + SVM	Neural (768-dim)	0.984	0.516	0.506
ModernBERT Triplet	Metric learning	—	0.486	—
CodeBERT Direct	Sequence embeddings	0.988	0.403	—
RF/SVM Handcrafted	Statistical (53–58 feat.)	0.968	0.324–0.380	—
TF-IDF + LinearSVC	N-gram frequencies	0.970	0.300	—
ModernBERT Direct	Sequence (8192 ctx.)	0.988	0.243	—

Table 4.7: Comprehensive comparison of all methods sorted by OOD Macro-F1. Kaggle scores are from official competition submissions.

Two cross-cutting findings are worth noting. First, threshold calibration provides a consistent and substantial gain across all methods where it was applied (+0.15 to +0.23 Macro-F1), confirming that the default threshold is poorly calibrated for the OOD label distribution (22.3% machine vs. 52.3% in training). Second, training data composition has a decisive effect: models trained on the full 500k training set, which is 91.5% Python, overfit severely to Python-specific patterns, while the balanced `diverse_train` split consistently produces better OOD generalisation across all method families.

Chapter 5

Conclusion

5.1 Summary

This thesis investigated out-of-distribution generalisation in machine-generated code detection, grounded in the SemEval-2026 Task 13 benchmark. The central challenge is that the available training data covers only three programming languages and a single coding domain, while the evaluation set introduces five additional languages, two additional domains, and dozens of generators unseen during training. Ten detection methods were systematically evaluated, spanning lexical baselines, structural features, neural embedding pipelines, metric learning, comment-based embeddings, and stylometric analysis. The main findings are as follows.

In-distribution performance does not predict OOD robustness. All methods that optimise directly for training-set classification — including TF-IDF, handcrafted statistics, and direct neural classifiers (CodeBERT, ModernBERT) — achieve validation Macro-F1 values above 0.94 but collapse to OOD Macro-F1 values between 0.24 and 0.41. This failure occurs regardless of model architecture, parameter count, or context window size, confirming that the failure mode is shortcut learning rather than insufficient model capacity.

Structural features generalise better than lexical features. AST-based models, which represent the logical structure of code rather than its surface formatting, achieve an OOD Macro-F1 of 0.549 after threshold calibration — substantially

outperforming all direct-classification baselines.

Two-stage embedding approaches generalise better than end-to-end classification. Fine-tuning a code model and training a separate linear classifier on the extracted embeddings avoids the shortcut learning that plagues end-to-end classifiers. The best result in this group, using MIL chunk-pooling with UniXcoder, achieves an OOD Macro-F1 of 0.602, confirming that truncating code at 512 tokens also discards meaningful discriminative signal.

Code comment style is the most OOD-robust discriminative signal. The comment embedding approach achieves the best performance across all metrics: OOD Macro-F1 of 0.671 and Kaggle submission score of 0.638. The key insight is that AI-generated code exhibits a consistent conversational and instructional tone in its comments — verbose, structured, and explanatory — that remains stable across programming languages and generator families. Targeting this signal with a language-agnostic embedding model avoids the code-structure overfitting that limits code-specific encoders.

Threshold calibration is essential under label shift. The OOD test distribution (77.7% human) differs substantially from training (47.7% human). Without adjusting the decision threshold for this shift, all probabilistic and margin-based classifiers over-predict the majority class. A simple threshold search yields gains of +0.15 to +0.23 Macro-F1 across all tested methods.

Training data composition matters more than training set size. The full 500k training set, being 91.5% Python, causes severe language-specific overfitting. The balanced `diverse_train` split — 10k samples per language per class across Python, C++, and Java — provides substantially better cross-language generalisation across all method families evaluated.

5.2 Future Work

Comment–code joint modelling. The comment embedding approach demonstrates that linguistic signals in comments are highly discriminative. A natural ex-

tension is a unified model that jointly represents code and its comments, learning to upweight comment regions during detection. Cross-modal attention or dual-encoder architectures with learned fusion could exploit this multi-modal structure more directly than the current feature concatenation approach.

OOD-aware training objectives. The current best methods use pre-trained models not specifically optimised for OOD robustness. Training with explicit OOD objectives — domain-adversarial training, contrastive loss across language groups, or invariant risk minimisation — could further improve generalisation. The DroidDetect paper [21] demonstrates that contrastive learning with uncertainty-based resampling provides consistent gains; applying these techniques to the comment embedding model is a promising direction.

Subtask B and authorship attribution. This thesis focused exclusively on Subtask A (binary human vs. machine detection). Subtask B requires attributing code to specific LLM families (DeepSeek, Qwen, Meta-LLaMA, OpenAI, etc.), a substantially harder problem. The comment embedding insight may extend to authorship attribution, as different LLM families likely exhibit distinct comment styles that the `all-mpnet-base-v2` embedding model could capture.

Adversarial robustness. The evaluation test set includes adversarially humanised code samples designed to evade detection. The current methods were not specifically trained for adversarial robustness. Incorporating humanised adversarial samples into `diverse_train` and exploring targeted data augmentation strategies such as comment paraphrasing and variable renaming are worthwhile directions.

Larger pre-trained models. Experiments in this thesis were constrained by NVIDIA V100 GPU memory, limiting model size to approximately 125–149M parameters. Larger code models such as CodeT5+-770M, StarCoder, or CodeLlama may provide richer representations. The two-stage embedding pipeline is particularly well-suited to benefit from larger encoders, since inference is performed once per sample and embeddings can be stored for repeated classifier training.

Deployment calibration. In real-world settings such as academic submission screening or code repository auditing, the base rate of AI-generated code is unknown

and likely much lower than in the benchmark. Calibrating detector confidence scores for deployment-time label distributions, and designing systems that operate at high precision with adjustable recall thresholds, are important practical extensions of this work.

Bibliography

[1] Anthropic. Claude code, 2024. Coding assistant capability of Claude.

[2] Owura Asare, Meiyappan Nagappan, and N. Asokan. Is github's copilot as bad as humans at introducing vulnerabilities in code? *arXiv preprint arXiv:2204.04741*, 2024.

[3] Yuntao Bai, Sébastien Jones, Kamal Ndousse, et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.

[4] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[5] Cursor AI. Cursor, 2024. AI-native IDE.

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

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[8] GPTZero. Gptzero, 2024. AI writing detection platform.

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

[10] Andrea Gurioli, Maurizio Gabbrielli, and Stefano Zacchiroli. Is this you, LLM? recognizing AI-written programs with multilingual code stylometry. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 394–405. IEEE Computer Society, 2025.

[11] Alexander Hermans, Lucas Beyer, and Bastian Leibe. In defense of the triplet loss for person re-identification. *arXiv preprint arXiv:1703.07737*, 2017.

[12] Oseremen Joy Idialu, Noble Saji Mathews, Rungroj Maipradit, Joanne M. Atlee, and Mei Nagappan. Whodunit: Classifying code as human authored or gpt-4 generated - a case study on codechef problems. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, pages 394–406, New York, NY, USA, 2024. Association for Computing Machinery.

[13] Vaibhavi Kalgutkar, Ratinder Kaur, Hugo Gonzalez, Natalia Stakhanova, and Alina Matyukhina. Code authorship attribution: Methods and challenges. *ACM Computing Surveys*, 52(1):1–36, 2019.

[14] MBZUAI NLP Lab. Semeval-2026 task 13: Detecting machine-generated code with multiple programming languages, generators, and application scenarios. GitHub repository, 2026. Accessed: 2026-02-16.

[15] Microsoft and GitHub. Visual studio code with github copilot, 2024. IDE and AI coding assistant.

[16] Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D. Manning, and Chelsea Finn. Detectgpt: Zero-shot machine-generated text detection using probability curvature. *arXiv preprint arXiv:2301.11305*, 2023.

[17] Phuong T. Nguyen, Juri Di Rocco, Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Massimiliano Di Penta. Gptsniffer: A codebert-based classifier to detect source code written by chatgpt. *Journal of Systems and Software*, 214:112059, 2024.

[18] OpenAI. Gpt-4 technical report. Technical report, OpenAI, 2023.

[19] OpenAI. Introducing codex. Blog post on OpenAI website, May 2025. Research preview of the Codex software engineering agent.

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

[21] Daniil Orel, Indraneil Paul, Iryna Gurevych, and Preslav Nakov. `Droid`: A resource suite for ai-generated code detection. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing (EMNLP 2025)*, pages 31251–31277, Suzhou, China, November 2025. Association for Computational Linguistics.

[22] Baptiste Roziere, Loubna Ben Allal, Jiezhong Li, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[23] Sapling AI. Sapling, 2024. AI writing assistant and detection tools.

[24] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 45(4):427–437, 2009.

[25] Hyunjae Suh, Mahan Tafreshipour, Jiawei Li, Adithya Bhattiprolu, and Iftekhar Ahmed. An empirical study on automatically detecting ai-generated source code: How far are we? *arXiv preprint arXiv:2411.04299*, 2024.

[26] Lewis Tunstall, Leandro von Werra, Younes Belkada, et al. Starcoder: May the source be with you. *arXiv preprint arXiv:2305.06161*, 2023.

[27] Jian Wang, Shangqing Liu, Xiaofei Xie, and Yi Li. An empirical study to evaluate aigc detectors on code content. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pages 844–856, New York, NY, USA, 2024. Association for Computing Machinery.

[28] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088, 2023.

[29] Yuxia Wang, Jonibek Mansurov, Petar Ivanov, Jinyan Su, Artem Shelmanov, Akim Tsvigun, Chenxi Whitehouse, Osama Mohammed Afzal, Tarek Mahmoud, Toru Sasaki, Thomas Arnold, Alham Fikri Aji, Nizar Habash, Iryna Gurevych, and Preslav Nakov. M4: Multi-generator, multi-domain, and multi-lingual black-box machine-generated text detection. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (EACL 2024)*, pages 1369–1407. Association for Computational Linguistics, March 2024.

[30] Weiwei Xu, Kai Gao, Hao He, and Minghui Zhou. Licoeval: Evaluating llms on license compliance in code generation. *arXiv preprint arXiv:2408.02487*, 2025.