GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Bachelor's Thesis

submitted in partial fulfillment of the
requirements for the course "Computer Science(two major)"

Detecting AI-Generated Code in Student Programming Learning Using Pre-trained Language Models

---

Zhuohang Yu

MatrNr: 29815778

First Supervisor: Julian Kunkel
Second Supervisor: Patrick Höhn

Georg-August-Universität Göttingen
Institute of Computer Science

ISSN: 1612-6793

April 29, 2026

Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

☎ +49 (551) 39-172000
FAX +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de
🌍 www.informatik.uni-goettingen.de

Abstract

The rapid growth of large language models has changed programming education. Now, students always generate code using Artificial Intelligence (AI) tools. This shift poses new challenges for automated assessment tools like SmartBeans, as it is crucial to differentiate between code written by students and that generated by AI. Current detection tools, such as GPTZero and DetectGPT, focus mainly on natural language. They often struggle in structured programming environments like C. These tools show low recall and inconsistent predictions, limiting their effectiveness in actual educational settings. To tackle this issue, this study suggests a hybrid detection framework that merges CodeBERT semantic representations with an XGBoost-based feature fusion model. The new method is tested with both labeled data and large-scale real-world submissions from the SmartBeans platform. The results indicate that the model achieves a recall rate of 75% and significantly outperforms general baseline methods in both accuracy and consistency. Further analysis over 44,258 submissions shows that the AI adoption rate has reached 20.13% since the release of ChatGPT. This rate remains steady over time. This finding indicates that AI-assisted programming has become common among students and marks a lasting change in learning behavior.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☑ During brainstorming
- ☑ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☑ For the development of software source texts
- ☑ For optimizing or restructuring software source texts
- ☑ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Acknowledgements

I acknowledge Professor Julian Kunkel for agreeing to act as the first supervisor and Dr. Patrick Höhn for serving as the second supervisor for this thesis, thereby supporting the formal completion of this work. Special appreciation is extended to Lorenz Glissmann, who provided continuous guidance in both the experimental work and thesis writing, as well as detailed feedback and practical support throughout the development of this thesis. I also acknowledge the SmartBeans platform for providing real-world student programming data, which formed the basis of the empirical analysis in this study. In addition, this work made use of established tools and frameworks, including CodeBERT and XGBoost, which supported the implementation of the proposed methodology.

The source code used for the analysis in this thesis are available at: `https://gitlab.gwdg.de/zhuohang.yu/ai-test.git`.

Contents

List of Tables vi

List of Figures vii

List of Listings viii

List of Abbreviations ix

1 Introduction 1
- 1.1 Research Questions . . . 2
- 1.2 Contributions . . . 2

2 Background and Foundations 3
- 2.1 Large Language Models for Code Generation . . . 3
- 2.2 Characteristics of AI-Generated Code . . . 4
- 2.3 Detection of AI-Generated Code . . . 4
  - 2.3.1 Statistical-based Detection . . . 4
  - 2.3.2 Pre-trained Representation-based Detection . . . 5
  - 2.3.3 Perturbation-based Detection . . . 5
  - 2.3.4 Threshold Calibration and False Positive Rate Control . . . 6
- 2.4 Evaluation Metrics . . . 6

3 Related Work 6
- 3.1 Statistical Detection and Code Naturalness . . . 6
- 3.2 Deep Learning-based Representation and Classification . . . 9
- 3.3 Zero-shot Detection and Perturbation-based Methods . . . 9
- 3.4 AI Governance and Empirical Studies in Educational Contexts . . . 10

4 Methodology 12
- 4.1 Data Collection and Pre-processing . . . 12
- 4.2 Feature Engineering . . . 13
  - 4.2.1 Code structure and style characteristics . . . 13
  - 4.2.2 Statistical Validation and Feature Selection . . . 14
- 4.3 Statistical Validation of Features . . . 14
  - 4.3.1 Statistical Tests of Feature Differences . . . 14
  - 4.3.2 Feature Filtering . . . 14
- 4.4 Model Construction and Optimization . . . 15
  - 4.4.1 Baseline: Perplexity Threshold Classifier . . . 15
  - 4.4.2 Proposed: XGBoost Ensemble . . . 16
  - 4.4.3 Algorithm selection criteria . . . 16
- 4.5 Comparative Methods . . . 16
- 4.6 Model Calibration and Deployment . . . 17
- 4.7 Detection Analysis and Temporal Trends . . . 18
  - 4.7.1 AI code feature representation in the post-ChatGPT era . . . 18
  - 4.7.2 ChatGPT adoption rate trend analysis before and after release . . . 19

5 Results and Analysis **19**
5.1 Model Performance Evaluation 19
5.2 Three Model Comparative 21
5.3 Cross-Model Consistency Analysis 22
5.4 Temporal Adoption Trends 24
5.5 Characteristic Profiling 26
5.6 Example: Human vs. AI 27
5.7 summary 29

6 Conclusion and Future Work **30**
6.1 Research Summary 30
6.2 Threats to Validity 30
6.3 Future Work 30

References **32**

A Additional Materials **A1**
A.1 AI-generated Code Sample A1
A.2 Human-written Code Sample A2

List of Tables

1 Performance Comparison between Baseline and XGBoost Fusion Model . . 20

List of Figures

1 Research Framework for AI-generated Code Detection and Evolution Analysis 13
2 Statistical Validation and Feature Selection. 15
3 Feature Comparison: Student vs AI. 15
4 Feature Importance 16
5 Model Comparison 17
6 Model Performance Evaluation: Comparison between PPL Baseline (top) and XGBoost Fusion Model (bottom). 20
7 Performance Comparison of AI Code Detection Models on Labeled Dataset 22
8 Prediction Confidence Distribution Across Three Models 22
9 Multi-feature fusion model: CodeBERT + XGBoost 23
10 scheme2 post 2024 consistency matrix 24
11 2022-2024 ChatGPT adoption rate trend 25
12 Longitudinal Trend of AI Reliance Risk Post-ChatGPT 26
13 Comprehensive Analysis of AI Code Features and Detection Rates. 27

List of Listings

1 **AI-generated code** 28
2 **Human-written code** 29

List of Abbreviations

ROC Receiver Operating Characteristic

AUC Area Under the Curve

FPR False Positive Rate

PPL Perplexity

LLM Large Language Model

AI Artificial Intelligence

NLP Natural Language Processing

MLM Masked Language Modeling

LoC Lines of Code

NPR Normalized Perturbation Log-Rank

TPR True Positive Rate

AST Abstract Syntax Tree

RTD Replaced Token Detection

1 Introduction

In recent years, code generation technologies based on Large Language Models (LLMs) have developed rapidly and are gradually becoming important tools in software development and programming practice [Fan+23]. Systems such as ChatGPT and Claude can automatically generate structurally complete, syntactically correct, and highly readable code based on natural language descriptions. These models have demonstrated significant value in multiple development stages, such as automatic generation of unit tests [JSR25], code completion [Bav+22], error prediction, and code refactoring, thereby significantly improving development efficiency. However,their impact on learning programming is more controversial, as some studies suggest that excessive reliance on such tools may hinder the development of fundamental problem-solving skills. With the widespread application of these tools in actual development, students have increasingly started using these tools in programming courses. In the teaching process, students can use LLMs to generate multiple feasible solutions in a very short time, and even directly complete course assignments. In our course setting, the use of such tools is explicitly permitted, which makes it possible to study their impact on student behavior. However, it is important to note that in many other courses, the use of AI tools is restricted or actively discouraged. While this capability provides powerful support for learning, it has also raised profound concerns about learning effectiveness and academic integrity. If students rely excessively on AI tools, they might lead to a lack of independent thinking and logical construction processes, their understanding of programming knowledge may remain superficial, thus weakening the achievement of teaching objectives [LPP23]. This problem is particularly prominent in programming courses centered on automated assessment systems. For example, in the SmartBeans platform used at the University of Göttingen, student code submissions are typically evaluated by an automated grading system. And these systems in programming courses typically evaluate submissions based on predefined test cases and output correctness, and generally do not attempt to determine the origin of the code. As a result, identifying whether a solution was written by a student or generated by AI remains challenging.Traditional plagiarism detection methods, usually based on code similarity analysis, can identify copying behavior between students, but often fail with AI-generated code[Tay+23]. This is because AI can generate semantically equivalent but expressively diverse code, while human-written code can also vary in structure and expression. AI-generated code can produce a large number of semantically equivalent yet syntactically diverse solutions within a very short time, making it difficult for traditional methods to identify at both the structural and textual levels. Therefore, distinguishing between human-written code and AI-generated code has become a critical issue that needs to be addressed in current programming education. In the context of programming education, the primary concern is not the correctness of generated code, but its impact on learning outcomes. In the extreme case, students may simply input assignments into an AI system and submit the generated solution without engaging with the underlying concepts. In such cases, meaningful learning is unlikely to occur. In this context, developing effective AI code detection methods is of great significance. However, compared to natural language text, code has stronger structural and syntactic constraints, significantly limiting the performance of existing text detection tools such as GPTZero in code scenarios [XS24].Furthermore, many existing methods only output the probability of whether something was generated by artificial intelligence, lacking interpretability. Therefore, in

an educational setting, this lack of transparency raises fairness concerns, as students may be labelled without comprehensible or explainable evidence. In recent years, some studies have proposed using the statistical properties of language models to identify AI-generated content. For example, the DetectGPT method [Mit+23], based on perturbation-based analysis, proposes that AI-generated text often lies at local extrema in the model's probability distribution, and slight perturbations significantly reduce its log probability. This idea can be extended to the code domain, suggesting that AI-generated code may exhibit characteristics different from human code in terms of perplexity and stability. Moreover, the theory of code naturalness also indicates that human-written code tends to be more irregular, while AI-generated code tends to be more consistent and standardized. Based on the above observations, two key hypotheses can be proposed:

- AI-generated code differs significantly from human code in statistical characteristics such as perplexity and burstiness. (Perplexity measures how predictable a piece of code is under a language model, while burstiness captures the variability of token usage patterns).
- AI code exhibits specific patterns in identifier naming, nested loop structures, and redundancy. These characteristics provide an important foundation for building automatic detection models.

Although existing research has made some progress in text detection, systematic research on programming code remains relatively limited. In particular, there is still considerable room for research on methods combining behavioral analysis, statistical features, and model interpretability based on real educational platform data. Furthermore, it is currently unclear how much students rely on AI tools in actual learning, and whether there is a discrepancy between their self-reported behavior and actual usage. To address these issues, this thesis systematically studies the characteristics of AI-generated code and its detection methods based on real submission data from the SmartBeans platform. We extract features from multiple dimensions, including code structure, naming conventions, and statistical indicators, and construct a hybrid detection framework.

1.1 Research Questions

Based on the above methods, this thesis focuses on the following research questions:

- RQ1: In SmartBeans programming data, which features can effectively correlate with students' AI usage behavior?
- RQ2: Compared with human-written code, does AI-generated code show significant differences in statistical features?
- RQ3: How has the usage of AI tools within the SmartBeans platform evolved?
- RQ4: How well do machine learning models perform in detecting AI-generated code?

1.2 Contributions

The main contributions of this thesis are as follows: First, we systematically analyzed the multidimensional feature differences between AI-generated code and human code in

a real-world educational data environment. Second, we explored a detection framework integrating statistical features and deep learning representations, and provided evidence supporting interpretability. Third, by tracking data from the SmartBeans platform over several years, the study revealed the evolving patterns in students' AI usage behavior, providing data support for universities to formulate academic integrity policies and optimize programming curriculum design. The code source at `https://gitlab.gwdg.de/zhuohang.yu/ai-test.git`.

2 Background and Foundations

2.1 Large Language Models for Code Generation

Large Language Models (LLMs) may suggest that the most significant developments in AI and Natural Language Processing (NLP) have emerged from this particular class of models. Moreover, the findings could indicate that these models, trained on significant amounts of text data, demonstrate that understanding, generating, and processing human language appears achievable at scale. Furthermore, early results, such as [Bro+20], may suggest that models like GPT-3 could demonstrate the capacity to support tasks such as translation, summarization, and code generation. In light of the evidence, the key results might indicate that these models appear to acquire grammar, meaning, and general programming patterns without task-specific training. Most modern LLMs are based on the Transformer architecture proposed by [VRW23]. The Transformer uses a self-attention mechanism which allows the model to capture long-distance relationships in a sequence. This ability is very important for both natural language and programming languages because context affects meaning. For example, in programming, variable definitions and function calls may appear far apart in the code. LLMs work in an autoregressive way, which, given an input prompt, they predict the next token based on previous tokens. In this way, they generate code step by step. In addition, studies such as [Kap+20] show that Transformer models scale well. They can handle billions of parameters and achieve strong performance in many tasks. Although the evidence might suggest that LLMs originated from natural language tasks, it seems clear these models are now widely used in programming. The results may indicate that training on large datasets containing both code and text allows for learning syntax, semantics, and common coding patterns. Furthermore, this has led to the creation of specialized models like CodeBERT [Fen+20], CodeT5 [Wan+21], PolyCoder [Fen+20], and PaLMCoder [Cho+22], suggesting that understanding and generating code can be done at a higher level. The findings show these models are built to better handle code-related tasks. They may indicate that pre-training and fine-tuning can help them work across many software engineering tasks. In practice, LLMs can assist with various activities, including code completion, function generation, error detection, and automatic documentation. Tools such as ChatGPT and GitHub Copilot have made these features available to developers. Consequently, programming supported by AI is becoming more common. LLMs are now an essential part of modern software development. [Jia+26]. However, LLMs do have limitations. First, they depend on statistical patterns from their training data and do not understand problems as humans do. Because of this, the code they generate may seem right but can lack logical clarity. Second, LLMs can create incorrect, biased, or unsafe code, raising concerns about reliability and security. Additionally, training these models needs a lot of computational

resources, which raises both costs and environmental impact.

In summary, LLMs have strong capabilities in natural language and code generation, changing how software is created. However, they rely on learned patterns instead of real reasoning, creating a significant difference between AI-generated code and human-written code. This distinction is crucial for the analysis and detection methods discussed in this thesis.

2.2 Characteristics of AI-Generated Code

AI-generated code is distinct from human-written code in several ways. Since AI models learn from vast datasets, their output frequently follows fixed patterns, resulting in consistent structure and format. Research indicates that AI-generated code is usually more concise and tends to utilize standard coding patterns and general-purpose functions. [CIL25]. Regarding structure and complexity, AI-generated code is often simpler than human code. It generally has fewer Lines of Codes (LoCs) and fewer tokens, resulting in lower lexical diversity. [CIL25]. While this gives the code a clean appearance, human-written code often contains more complex logic. For example, it may feature higher cyclomatic complexity and deeper nesting, reflecting how humans solve problems step by step. In contrast, AI often provides more straightforward and shallower solutions.[ENS24]. There are also differences in code quality. AI-generated code may contain common issues, such as unused variables or parameters. It may also include debugging statements like ‘System.out.println()‘ or ‘printStackTrace()‘ [CIL25]. Some software metrics show similar patterns in both AI and human code. For instance, longer code is often linked to specific errors, like issues with local variables. Loop complexity can also lead to poor conditional structures. [PSS24]. Moreover, AI-generated code is not always consistent; even with the same input, the output may vary. It might generate correct code in one instance and incorrect code in another.[Bus23]. AI developers and human developers also differ in coding style and meaning. AI-generated code often uses simple, generic variable names. In contrast, human developers typically choose more specific and meaningful names.[Bul+24]. Comments show differences as well. AI-generated comments usually provide basic function descriptions, while human-written comments often include more details, such as design ideas or explanations.

In conclusion, AI-generated code and human-written code differ in structure, complexity, quality, security, and style. These differences create a valuable foundation for detection methods. By combining statistical features like token distribution with structural features such as complexity and defects, it is possible to develop models that can determine whether code is AI-generated or human-written.

2.3 Detection of AI-Generated Code

From a methodological perspective, existing AI-generated code detection methods can be broadly categorized into three types: statistical-based methods, pre-trained representation-based methods, and perturbation-based methods. These methods differ in the types of information they utilize and their fundamental assumptions.

2.3.1 Statistical-based Detection

Statistical-based methods rely on the "Code Naturalness" hypothesis: although programming languages are human-defined, human-written code exhibits extremely high repeata-

bility and statistical predictability. These methods assume that AI-generated code tends to concentrate probability mass on more predictable token sequences, resulting in lower variance in token-level probabilities compared to human-written code. Two core metrics are Perplexity and Burstiness:

• Perplexity (PPL): Measures the certainty with which a language model predicts a given sequence of lemmas. For a code word sequence $W = (w_1, w_2, ..., w_N)$ of length N, its perplexity calculation formula is:

$$PP(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1, ..., w_{i-1})}} \tag{1}$$

Since LLMs are trained to model the probability distribution of token sequences by maximizing the likelihood of next-token predictions, the generated code often lies in relatively high-probability regions of this distribution. As a result, it tends to exhibit lower perplexity compared to human-written code. Perplexity is most effective when the evaluated sequence aligns with the model's training distribution; however, it becomes less reliable when the input is truncated or lacks sufficient context. For example, removing the beginning of a sequence typically increases perplexity, as the model has less contextual information to condition on.

• Burstiness: Reflects the dynamic changes in perplexity within the sequence. Human code, due to irregular logical transitions, diverse commenting habits, and non-standard naming styles, often exhibits extremely high perplexity fluctuations in local areas; while AI-generated code has a smoother and more uniform distribution [XS24]. The main advantage of this type of method is its extremely high computational efficiency and the fact that it does not require complex syntax parsing for specific programming languages, making it suitable for large-scale real-time monitoring scenarios on platforms such as SmartBeans.

2.3.2 Pre-trained Representation-based Detection

Pre-trained representation-based methods utilize deep learning models to extract deep semantic features from code. Models like CodeBERT [Fen+20] capture the structure and meaning of code. They do this by pre-training on large codebases using Masked Language Modeling (MLM). First, they encode code snippets into high-dimensional embeddings. These vectors reflect the logical flow and purpose of the code, beyond just word frequencies. The semantic representations are then used with classifiers, such as XG Boost, to determine the boundaries between AI and human code for classification. This approach has more expressive power compared to purely statistical methods. It can identify AI code that has been slightly altered, but it requires more computing power and labeled data.

2.3.3 Perturbation-based Detection

Perturbation-based methods, such as DetectGPT [Mit+23], use the probability curvature of language models. This means there is no need to train another classifier. The main idea is that AI-generated code tends to stay close to a local maximum in the model's log probability distribution. When small changes are made to AI-generated code, like renaming variables or adjusting formatting, the log probability usually drops sharply. In contrast, human code is not ideally placed in the probability space, and its probability

changes with small alterations are less predictable and fluctuate more. This method works very well in "zero-shot" situations and can generalize effectively to different model versions and unknown question types.

2.3.4 Threshold Calibration and False Positive Rate Control

In programming education, the fairness of detection methods is important. However, the impact of false positives depends on how detection results are used. In our setting, the detection outcomes are not directly used for punitive measures, but rather as indicators to support further analysis. Therefore, the detection model cannot depend solely on a standard classification threshold. Threshold calibration is essential to control the trade-off between false positives and false negatives. By examining a reference set of human-written code, such as submissions from a period before the widespread adoption of LLMs, a conservative decision threshold can be established to limit the False Positive Rate (FPR).

2.4 Evaluation Metrics

To fairly assess the AI code-detection model's effectiveness, we used several standard machine-learning evaluation metrics. These metrics not only measure the model's overall classification ability but also help balance accuracy with the risk of misclassification in educational settings.

- Accuracy: Accuracy measures the proportion of correctly predicted data across all samples.
- Recall also known as True Positive Rate (TPR) is defined as $\frac{TP}{TP+FN}$. In academic integrity detection, recall reflects the model's ability to capture AI-generated code; that is, what percentage of all actual AI-generated submissions are successfully detected.
- Receiver Operating Characteristic (ROC) Curve: The ROC curve plots the relationship between the TPR and the FPR by continuously adjusting the classification threshold. This curve visually illustrates the performance trade-offs of the model at different levels of strictness. For the SmartBeans platform, the ideal model should maximize TPR while maintaining an extremely low FPR.
- Area Under the Curve (AUC): AUC is the area under the ROC curve, ranging from 0 to 1. AUC provides a composite performance metric independent of specific thresholds: the closer the AUC value is to 1, the stronger the model's ability to distinguish between human-written code and AI-generated code; an AUC of 0.5 indicates that the model's performance is equivalent to random guessing.

3 Related Work

3.1 Statistical Detection and Code Naturalness

Early academic research on the statistical properties of source code laid the theoretical foundation for AI code detection.[Hin+16]proposed the famous "Code Naturalness"

hypothesis, pointing out that programming languages, like natural languages, are statistically highly predictable and repeatable. Through analysis using modern statistical methods, such as n-gram probabilistic language models, they found that source code is actually more regular and has extremely low cross-entropy than natural languages like English. This observation allows probabilistic language models to be directly applied to the code domain, and metrics such as perplexity or cross-entropy can be used to quantify the predictability of code and its "probability distribution". Therefore, statistical detection methods assume that AI-generated code is more predictable than human-written code because it is generated based on the learned probability distributions of massive amounts of code [LK24]. In the field of AI detection, existing research mainly focuses on using probabilistic metrics from model outputs to distinguish between machine and human creations:

- Application of Perplexity: Perplexity measures the confidence of a language model in predicting a sample sequence[LK24]. Many studies have pointed out that because LLMs are optimized on massive codebases, the generated code tends to select word combinations with the highest statistical probability. Therefore, AI-generated code usually exhibits lower perplexity than human code. In contrast, when programming or expressing themselves, humans often write code snippets with higher complexity and perplexity to balance semantic clarity, logical integrity, and personal habits.

- Identification of Burstiness: In addition to overall probability, researchers have begun to focus on local fluctuations in probability. When solving complex programming logic, humans often exhibit extremely high perplexity in certain key lines, such as complex nested loops or non-intuitive algorithm implementations; this fluctuation is called "burstiness." In contrast, AI-generated code is smoother and more predictable throughout [XS24].

- Perturbation Sensitivity: Recent research has expanded the application of statistical detection by analyzing how small perturbations affect the probability distribution of code. The basic idea is that AI-generated code often resides near local maxima of the model's likelihood function, making it more sensitive to perturbations. This observation has inspired a series of perturbation-based detection methods, such as DetectGPT and its variants [Mit+23].

- Limitations of Existing General-Purpose Tools: While commonly used tools like GPTZero and DetectGPT have achieved success in natural language detection, their generalization ability in the code domain is extremely poor [Suh+24]. Due to the unique syntax and writing style of source code, which are drastically different from natural language, these plain text detectors generally have low accuracy in recognizing AI code [Kha+23]. Furthermore, even detectors specifically fine-tuned for code, such as CodeBERT-based GPTSniffer [Ngu+24] exhibit concerning generalization ability when faced with highly optimized code generated from different programming languages, from Java to Python or C++, or different LLMs like Gemini Pro, GPT-4, etc., often resulting in a significantly higher false positive rate [Suh+24].

However, despite the advantages of statistical methods such as perplexity—no need for large-scale labeled data, strong interpretability, and good generalization ability across large models—their limitations in practical applications are becoming increasingly apparent:

- Language dependence and the impact of abstraction level: The effectiveness of statistical detection is significantly affected by the characteristics of the programming language. A large-scale empirical study by [Xu+25] shows that the perplexity method performs exceptionally well in low-level languages, such as C/C++, which AUC exceeding 90%, but performs poorly in high-level languages, such as Python and Ruby, with AUCs of only around 64% and 73%, respectively. This is because languages like C/C++ require developers to manually implement many low-level details, such as memory management, pointers, array slicing, etc., resulting in strong individual coding styles that significantly increase the difficulty for AI to imitate them completely. Python, on the other hand, has rich built-in libraries and extremely concise coding conventions, making human code and AI code statistically highly similar, increasing the difficulty of differentiation.

- Sensitivity to Length and Scale: Statistical metrics are highly unstable when processing short code snippets. The study by[Xu+25] confirmed that when processing code longer than 50 lines (Large-scale), the AUC for perplexity detection can reach 87.81%, but when the code is shorter than 20 lines (Small-scale), the accuracy drops sharply to 69.75%. This is because perplexity calculation is highly dependent on the context. In short code, the first few terms often generate extremely high perplexity due to a lack of sufficient contextual clues. This early extreme fluctuation severely interferes with the average perplexity calculation of the entire short code, leading to confusion between human and AI code scores.

- Bottleneck in Detection Efficiency: Although simple perplexity calculation is theoretically efficient, the best-performing statistical detection methods, such as NPR, DetectCodeGPT, etc. often rely on a "micro-perturbation" strategy, i.e., performing multiple mask replacements or rearrangements on the original code and recalculating the comparison differences. Evaluations by [Suh+24] and [Xu+25] show that this perturbation-based detection method is extremely time-consuming. For example, DetectGPT may take more than 100 to 150 seconds to process a single sample, and even the optimized DetectCodeGPT takes about 10 to 15 seconds. This high computational cost is completely unacceptable for the real-time (Just-in-time) detection requirements in programming competitions, exams, or large-scale code reviews.

- Lack of Semantic Structural: Information Pure statistical detection mainly captures surface-level token distribution features, completely failing to integrate the deep logical intent and syntactic structure of the code. As LLM-generated code becomes increasingly closer to human semantic logic, single probability metrics are insufficient. [Suh+24] proposed that extracting and combining the Abstract Syntax Tree (AST) structural information of the code would significantly improve detection performance. Their experiments demonstrated that using a pre-trained model to extract the AST and generate deep semantic embeddings (AST Embeddings), then inputting them into a machine learning model, achieved an average F1 score of 82.55 (AUC nearly 90%), significantly surpassing existing pure text statistical and perplexity detection tools. [Hin+16]have also pointed out that extending the simple n-gram model to the syntactic and semantic levels can more accurately capture the deep rules of the code.

3.2 Deep Learning-based Representation and Classification

Deep learning-based methods have become the mainstream approach for detecting AI-generated code because they can capture semantic and contextual information. These methods typically rely on pre-trained models based on the Transformer architecture to encode source code into high-dimensional vector representations, often called code embeddings[KVT24; Suh+24]. These embeddings can then be used as input to downstream classifiers, like logistic regression, support vector machines, or tree-based models to distinguish between AI-generated code and human-written code. CodeBERT is one of the most widely used models in this field [Fen+20]. As a bimodal model, CodeBERT combines multiple pre-training objectives and is trained on large-scale corpora of natural and programming languages. In addition to masked MLM, CodeBERT also employs Replaced Token Detection (RTD), which requires the model to distinguish between original lexical terms and plausible substitution lexical terms generated by a small generator network. Compared to traditional BERT-style objectives, this mechanism enables models to better capture subtle semantic inconsistencies and logical correctness, making it particularly suitable for structured data such as source code. Therefore, CodeBERT is able to learn meaningful relationships between natural language descriptions and code sequences at both syntactic and semantic levels. In practical applications, the embedding vectors generated by such models are often combined with traditional machine learning classifiers. Recent research [Che+21] has shown that combining deep learning-based feature extraction with gradient boosting decision trees, such as XGBoost can achieve greater robustness than fully end-to-end neural network models, especially when dealing with small or imbalanced datasets. This hybrid approach leverages the representational power of deep models while maintaining the flexibility and interpretability of classic classifiers. To further improve representation quality, subsequent models incorporate richer structural information. For example, GraphCodeBERT [Guo+24] extends CodeBERT by integrating a data flow graph during the pre-training phase. It no longer treats code as a simple sequence of labeled data but explicitly models dependencies between variables and captures "value-source" relationships. This allows models to gain a deeper understanding of program semantics and often results in higher performance than purely statistical methods. While deep learning-based methods are effective, they also face several significant limitations. First, they require large amounts of labeled data for training and fine-tuning, which is often difficult to obtain, especially for niche programming languages or specific domains [Kum+26]. Furthermore, these models are often treated as black boxes, making it difficult to explain why a particular code sample is classified as AI-generated. This lack of interpretability is particularly pronounced in sensitive domains such as education, where decisions must be transparent and fair [MRA25]. Finally, these methods are susceptible to domain transfer. Models trained on specific datasets, e.g., Python code generated by certain LLMs, often experience significant performance degradation when applied to unseen programming languages, domains, or next-generation models, such as Gemini or GPT-4. [Ore+26].

3.3 Zero-shot Detection and Perturbation-based Methods

Compared to supervised methods, zero-shot and perturbation-based methods offer an efficient alternative for detecting AI-generated code without requiring labeled training data [Mit+23]. These methods do not require training a separate classifier; instead, they directly utilize the probability distribution of a pre-trained language model, particularly

its sensitivity to small changes in input [Su+23]. DetectGPT is a foundational work in this field, based on the observation that machine-generated content tends to lie near local maxima of the model's log-probability function [Mit+23]. To achieve this, DetectGPT applies a small perturbation, typically using a masked language model(e.g., T5) to a given sequence and measures the resulting change in log probability. Empirical results show that AI-generated samples typically exhibit a significant decrease in probability after perturbation, while human-written content shows more stable and less directional changes. This principle has been extended to the source code domain. However, due to the strict syntactic and semantic constraints of programming languages, simple perturbation strategies such as random lexical substitution can easily compromise code correctness [Liu+24a]. Therefore, more conservative perturbation strategies, such as inserting spaces or modifying formatting, are employed for specific code variants like DetectCodeGPT[Shi+24] to induce measurable probability changes while preserving the program's semantics. These methods retain the key advantage of zero-shot detection and can be generalized to different tasks without labeled datasets [Xu+25]. To address the high computational cost of repeated perturbations, several improvements have been proposed [Bao+24]. For example, Fast-DetectGPT[Bao+24] introduces the concept of conditional probability curvature, replacing explicit perturbations with lexical-level sampling in a single forward propagation, thereby improving efficiency by several orders of magnitude. Similarly, the DetectLLM [Su+23] series of methods explores alternative metrics such as the log-rank ratio. In particular, the Normalized Perturbation Log-Rank (NPR) metric significantly reduces the number of perturbations required while maintaining competitive detection performance. Despite these advances, perturbation-based methods still face several significant challenges in code detection. First, they are highly sensitive to semantic constraints. Even minor lexical-level modifications can alter program logic, introduce noise, and thus affect probability estimation [Liu+24a]. Second, these methods heavily rely on the underlying language model. When the generative model is unknown or significantly different from the detection model, performance can degrade dramatically due to distribution shifts [Su+23]. Third, even with efficiency improvements, the computational demands remain high for large-scale or real-time applications, such as educational platforms with ongoing code submissions[Xu+25]. Recent research has tried to address these issues by introducing more controllable perturbation strategies. For instance, frameworks like PECOLA suggest selective perturbation mechanisms that alter non-critical components while keeping important lexical terms intact. This approach reduces noise while maintaining meaningful structural changes. Additionally, hybrid methods that combine perturbation signals with representation learning or contrastive learning have shown promise in improving robustness under limited supervision [Liu+24b].

In summary, perturbation-based methods provide a solid and data-efficient way to detect AI-generated code. However, their reliance on probabilistic signals and their sensitivity to structural constraints limit their effectiveness in complex coding situations. This has led to the development of hybrid methods that incorporate statistical, structural, and learned representations.

3.4 AI Governance and Empirical Studies in Educational Contexts

The rapid rise of generative AI tools in programming and computer science education raises concerns about academic integrity and learning outcomes. Recent empirical studies

show that students are increasingly turning to tools like ChatGPT and GitHub Copilot for help with programming tasks, debugging, and even creating complete solutions. While these tools can greatly increase task completion rates and learning efficiency, especially with techniques like problem decomposition, they also introduce a major issue known as the "Assistance Dilemma." First identified by [KA07], this dilemma highlights the challenge of providing useful guidance while avoiding the risk of students becoming passively dependent on these tools [PCS24]. In reality, students often use AI systems as shortcuts rather than as aids to learning, blurring the lines between legitimate help and academic dishonesty [PCP25]. Empirical research also shows that while many students use AI tools, the distribution is uneven. Less skilled students tend to depend more on AI-generated code to understand complex algorithms and complete graded assignments [PCS24]. Meanwhile, there are noticeable differences between students and educators regarding their knowledge, attitudes, and perceptions of AI. A cross-dimensional study by [Kam+24] indicates that students generally view AI systems as additional "personal tutors," while educators express significant concerns. Specifically, 78.6% of the teachers surveyed believe AI will promote plagiarism, 61.4% think it will hinder critical thinking, and over 74% worry that AI will weaken traditional assessment methods. Additionally, many educators feel unprepared; over 63% reported their institutions lack the necessary resources, training, and support to effectively incorporate AI into their teaching practices. To tackle these challenges, the governance framework for AI in education is undergoing significant changes. Recent research has introduced the "Living GenAI Governance Model," which focuses on coordinated regulation across multiple layers. This includes overarching regulatory frameworks, like the General Data Protection Regulation, institutional policies, AI literacy training, and responsible use by students and educators[PCP25; GC25]. At the university level, schools are encouraged to revamp their assessment methods, moving away from merely testing knowledge retention to evaluating higher-order skills such as critical thinking, analytical ability, and collaboration between humans and machines [Spi+23]. Furthermore, the use of AI systems raises significant questions about data privacy, algorithmic bias, and digital equity. AI models depend on extensive student data, which creates compliance challenges and risks worsening existing educational inequalities [LZ25]. Institutions in resource-limited settings may struggle to access this technology, widening the digital divide. Additionally, the use of opaque "black box" AI systems for assessment and detection raises serious concerns about false positives and fairness, especially when evaluating student performance [GC25]. Recent research emphasizes the need for transparency and open data in AI governance. [Nia+24] argue that responsibly using open data can improve understanding and clarify the relationship between complex AI systems and stakeholders. Research also shows that trust in AI systems is crucial for successfully implementing AI-driven educational policies. Sustainable governance can only be achieved when students, educators, and institutions view these systems as transparent and fair. Importantly, these governance and empirical issues drive the need for trustworthy and interpretable AI-generated code detection systems. In large educational platforms like SmartBeans, detection tools must not only be highly accurate but also ensure transparency, scalability, and fairness. This underscores the need for methods that can balance statistical reliability, computational efficiency, and interpretability in real-world educational environments.

4 Methodology

This chapter introduces the overall methodology and experimental design of this study. This research primarily addresses several common problems, including overly simplistic feature sets, unstable models, and a lack of real-world validation. To solve these problems, we propose a method that uses multiple features together and statistically adjusts the results. The entire process includes data processing, feature extraction, model building and optimization, and method comparison and time analysis. These steps are interconnected to form a complete research process. This process can both explain the principles and be applied in practice. Compared to some studies that use only small-scale or simulated data, this study uses data from a real teaching platform. Therefore, the conclusions are closer to reality and have greater practical significance.

4.1 Data Collection and Pre-processing

This study establishes a multi-layered data system to support model training, performance evaluation, and deployment testing in real-world scenarios. The data primarily comes from two sources: manually labeled data and real-world platform data. The labelled dataset contains 160 code samples, including 60 C programs generated by ChatGPT and 100 student code submissions from the SmartBeans platform. This dataset is mainly used for model training and classification performance evaluation. The AI-generated samples were constructed based on real programming tasks extracted from student submissions such as taskid 43 and taskid 60. For each task, a standardized prompt describing the original assignment requirements was provided to ChatGPT to generate multiple solutions. To increase diversity and better simulate realistic coding behavior, different prompting strategies were used to produce solutions with varying coding styles, more structured vs. more concise implementations. All generated code samples were manually reviewed and tested in a C programming environment to ensure syntactic correctness and functional validity. Only solutions that produced the expected outputs were retained. The generated samples were then formatted to match the SmartBeans dataset structure, ensuring consistency with student submission data. It should be noted that while student-submitted code is considered human-written, its authenticity cannot be completely guaranteed; AI-assisted generation cannot be ruled out. Therefore, this portion of the data may contain label noise to some extent. However, this uncertainty reflects real-world educational scenarios, making the dataset more representative of practical applications. Prior research suggests that moderate label noise may improve model robustness under real-world conditions. Therefore, this thesis treats these samples as approximate representations of human-written code and further evaluates model stability using large-scale unlabeled data. To further mitigate the impact of this unquantifiable noise, we deliberately employed XGBoost, a tree-based ensemble method known for its inherent robustness to mislabeled instances compared to highly parameterized neural networks. In terms of data processing, we first clean up duplicates. The final version for each student in each task is selected from all submissions. This removes intermediate code from the debugging process, making the data closer to the students' final results. There are a total of 44,258 code submission. Next, the data is divided into two phases based on time. One phase, before November 2022, is considered the "pre-AI phase," where code is believed to be manually written and used to calibrate the model's misclassification rate. The other phase, after June 2024, is considered the "post-AI phase," used to observe changes following the widespread adoption of

AI. This time division is similar to a natural experiment, dividing data into "before" and "after" parts and comparing the differences to analyze the impact of AI on programming behavior.

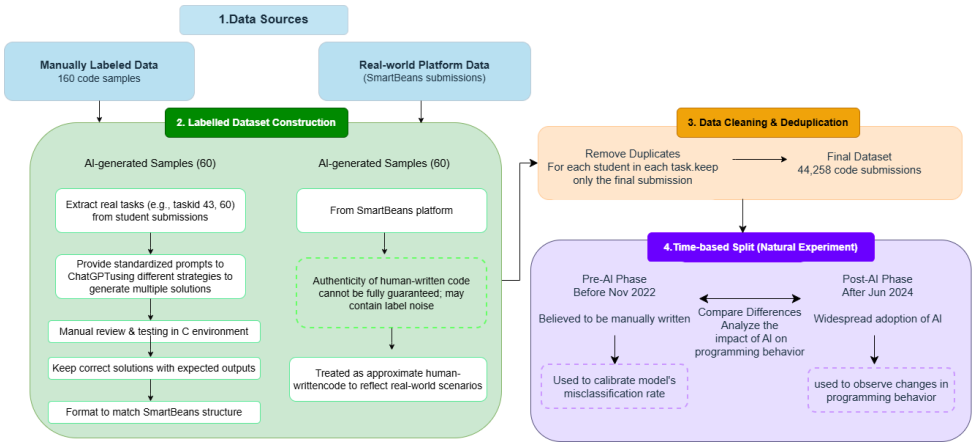


Figure 1: Research Framework for AI-generated Code Detection and Evolution Analysis

4.2 Feature Engineering

To more clearly distinguish between AI code and human code, this study designed a multi-dimensional feature extraction method. All features were automatically extracted by the AICodeAnalyzer tool and converted into structured data. All features were divided into two categories: statistical features and structural and stylistic features. These two feature categories are used together to describe the code from both the "probabilistic patterns" and "code structure" perspectives.

4.2.1 Code structure and style characteristics

This study uses the CodeBERT model as the foundational tool to calculate the probabilistic information of code. Compared to traditional models, CodeBERT learns two tasks simultaneously during training: mask prediction and substitution detection. Therefore, it has a more accurate understanding of code structure and is better suited for program analysis. Based on this, several metrics are extracted to measure whether the code is "natural." Perplexity represents the difficulty the model faces in predicting the code. Given a sequence of length N and a code sequence $W = (w_1, w_2, \ldots, w_N)$, the perplexity is calculated as follows:

$$PP(W) = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\ln P(w_i \mid w_1, \ldots, w_{i-1})\right) \tag{2}$$

This value can be interpreted as “whether the model finds this code easy to guess.” AI-generated code tends to select high-probability words, thus resulting in lower perplexity, resembling a very standard, unsurprising sentence. The *Average Token Probability* is the probability that each word appears. The higher this value, the more conventional the code is, and the more it resembles a “template-based expression”. Mean entropy is used to represent the uncertainty of prediction; its value measures the degree of variation in code. Lower entropy indicates more stable code and greater predictability. Burstiness represents the degree of fluctuation in probability changes; this value can be seen as the

"magnitude of fluctuation." Human code exhibits significant changes at key points, like a sudden shift in expression during writing, while AI code shows less variation, resembling a smooth curve. According to existing research[Hin+16], code is statistically similar to natural language. AI code often conforms too closely to this pattern, thus exhibiting an "overly neat" characteristic, as if written using a template.

4.2.2 Statistical Validation and Feature Selection

In addition to probabilistic features, this study also extracted a set of structural and stylistic features to reflect programming habits and the overall form of the code. Code length, average line length, and line length variation were used together to describe code size. These metrics reflect code neatness. AI-generated code is generally more regular, like a well-formatted article, while human code is more like a casually jotted-down note. Comment ratio indicates the number of comments. Experiments show that AI code typically contains more standardised comments, like a well-written instruction manual. Identifier entropy measures the diversity of variable names. Human developers are freer and have a more personal style when naming things, while AI tends to use common names, resulting in more repetition. N-gram repetition rate is used to detect duplicate segments. A high repetition rate indicates that the code is more likely to have been copied or generated from a template. These features are used together, like observing code from both "appearance" and "style" perspectives.

4.3 Statistical Validation of Features

Before constructing the classification model, this study rigorously validated the 10 extracted features to determine their actual effectiveness in distinguishing between human and AI code. Since code feature distributions are typically non-normal, we used the Mann-Whitney U test to compare the two classes of samples, setting the significance level at$\alpha = 0.05$. At the same time, Cohen's d coefficient was introduced to quantify the effect size, measuring the actual impact of feature differences.

4.3.1 Statistical Tests of Feature Differences

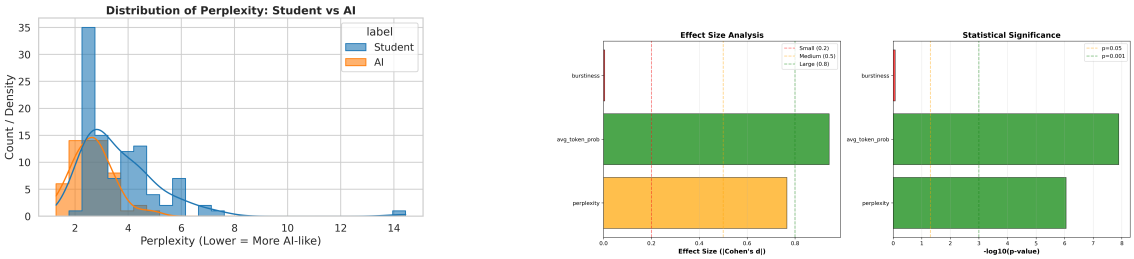
Statistical tests show that some features exhibit significant differences between the two classes of samples. For example, perplexity and average token probability demonstrate strong discriminative power, reflecting differences in the probability distribution of language models from different source code sources. Furthermore, some structural features (such as code length and comment ratio) also show some degree of difference. Meanwhile, some features did not show significant differences. For example, the statistical test result for burstiness did not reach the significance level, indicating that this feature has relatively limited discriminative power in the current task.

4.3.2 Feature Filtering

Not all features possess good discriminative power. For example, the statistical test result for burstiness is:

$$p = 0.845 > 0.05$$

This indicates that this feature does not show a significant difference between AI and human code. This phenomenon reflects the special nature of programming languages: in



(a) Distribution of PPL: Student vs AI. (b) Three Features Comparison.

Figure 2: Statistical Validation and Feature Selection.

environments with strong syntactic constraints, such as C, the code structure is highly standardized, and the variation of lexical probabilities is limited by language rules, thus weakening the discriminative power of the "expressive fluctuation" feature. In the model,

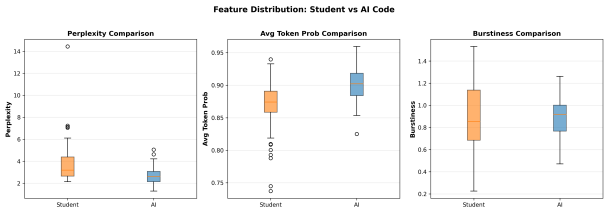


Figure 3: Feature Comparison: Student vs AI.

this study does not directly remove these features, but retains them in the feature matrix and automatically adjusts their weights using XGBoost's feature selection capabilities. In this way, low-contribution features are naturally weakened during training, ensuring that the model focuses on truly discriminative features.

4.4 Model Construction and Optimization

This study employs a stepwise construction approach, evolving the model from a simple to a complex form, and uses comparative results to illustrate the importance of multiple features. The model is initially set up as a single-feature form and then expanded into a multi-feature fusion form. This process helps determine the role of different features in real-world teaching environments.

4.4.1 Baseline: Perplexity Threshold Classifier

In the initial stage, a threshold classification model based on perplexity is constructed and used as a benchmark for comparison with subsequent models. This model classifies samples by setting different thresholds. Specifically, it iterates over thresholds within a preset range and selects a suitable threshold as the final classification criterion based on validation results. This process can be understood as adjusting the classification boundary to achieve preliminary differentiation between samples of different categories. Since this model relies on only a single statistical feature, its structure is relatively simple, facilitating an intuitive observation of the role of perplexity in distinguishing different source codes. Simultaneously, this model also provides a reference baseline for subsequent multi-feature models, used to evaluate the improvement effect brought about by feature fusion.

4.4.2 Proposed: XGBoost Ensemble

To address the aforementioned issues, we constructed a multi-feature model using the XGBoost algorithm. During the data processing phase, the 10 features extracted in Section 4.2 were input into the model. These features contain both statistical and structural information. Before training, all features were standardized, a step that eliminates the influence of different units, thus making the model's learning more stable. In terms of model setup, the XGBClassifier operator was used, with 100 iterative decision trees and a maximum depth (max_depth) of 5. This non-linear structure can capture complex relationships between features, such as strong AI signals when "low perplexity" and "high annotation ratio" coexist.The model outputs the probability value of a sample belonging to AI-generated code, and classification is achieved by setting a threshold. The relevant thresholds are determined through a validation process to adapt to the needs of different application scenarios. This multi-feature model provides the core experimental object for subsequent performance evaluation and method comparison.

4.4.3 Algorithm selection criteria

This study chose a combination of XGBoost and CodeBERT features, based on two considerations: Firstly, stability. Tree models are easier to train and less prone to overfitting in small datasets or with imbalanced data. Existing research [Che+21] has also yielded similar conclusions. Secondly, interpretability. Through the SHAP method, the impact of each feature on the result can be calculated, meaning the model's judgment process can be interpreted. Figure 4 show that perplexity and average entropy have the greatest impact on the results, indicating that the model primarily relies on these two key features for judgment. This "interpretable result" is crucial in educational settings, just as grading standards need to be made public, ensuring a fairer and more transparent judgment process.

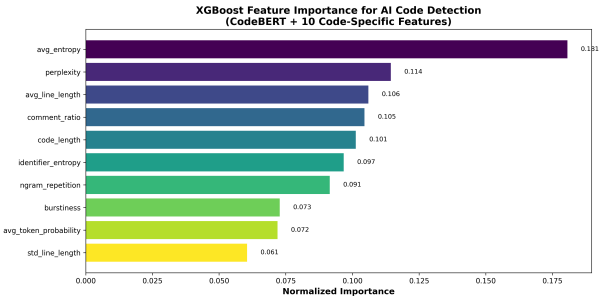


Figure 4: Feature Importance

4.5 Comparative Methods

To evaluate the effectiveness of the proposed method, this study selects three representative detection approaches for comparative analysis, namely GPTZero, DetectGPT, and the CodeBERT-based multi-feature fusion model proposed in this thesis. All three methods are tested under the same data conditions to ensure the comparability of the results. The comparison design consists of two parts. First, experiments are conducted on labeled data to evaluate the classification performance of each method. This dataset includes 60 AI-generated code samples (the generation process has been described in Section 4.1) and 5000 student submissions collected from the SmartBeans platform before

November 2022. Since the number of human-written samples is significantly larger than that of AI-generated samples, the human data is down-sampled by randomly selecting 300 samples, resulting in a class ratio of approximately 1:5, thereby alleviating the class imbalance problem. In the labeled data experiment, the manually constructed and validated AI-generated code samples, together with real student code, are used as input for all three methods. On this basis, class weights are further applied to balance the data distribution, and 5-fold cross-validation is used to evaluate learning-based methods. This part of the experiment aims to measure the overall performance of different methods in distinguishing AI-generated code from human-written code. The evaluation metrics include accuracy, precision, recall, F1-score, and AUC-ROC. Second, a consistency analysis is conducted on large-scale real-world data extracted from the SmartBeans platform to examine the stability of each method in practical scenarios. This dataset contains 44,257 student submissions. Since no ground-truth labels are available, this part does not directly evaluate classification accuracy, but instead analyzes the detection results from multiple perspectives. In the large-scale data analysis, a five-dimensional analysis framework is adopted, including: (1) the distribution of predicted AI proportions across methods, (2) sample-level consistency between methods, (3) the distribution characteristics of prediction confidence, (4) the relationship between code features and detection results, and (5) the identification of high-risk samples. To improve experimental efficiency, 1,500 samples are randomly selected each time, and the experiment is repeated three times, with the final results reported as the average across runs. At the same time, the framework supports extension to the full dataset for comprehensive analysis. Through this comparative design, this thesis provides a consistent framework for analyzing results, allowing for a systematic comparison of performance differences between standard text detection methods and specialized code detection methods in programming contexts.

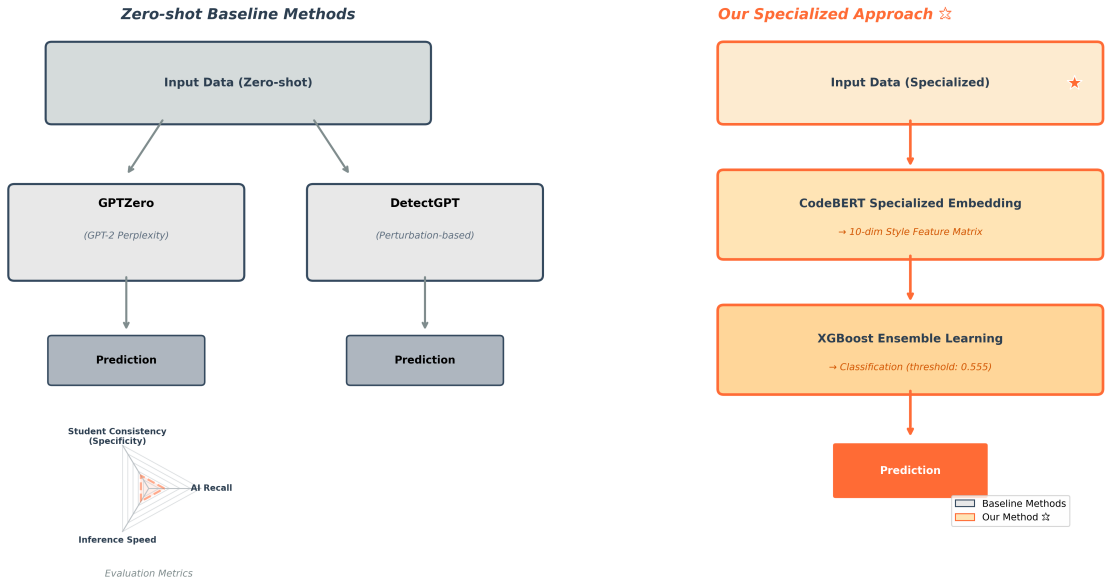


Figure 5: Model Comparison

4.6 Model Calibration and Deployment

To ensure the robustness and feasibility of the detection model in a real teaching environment SmartBeans platform, we introduced a calibration and deployment process after

model training. In programming education scenarios, misclassifying student-generated code can lead to academic controversy, therefore, controlling false positives is essential. We utilize the "Pre-AI Era" dataset (5,000 historical submissions before the release of ChatGPT on November 1, 2022, referred to as Subset A) as a calibration benchmark. This data is considered entirely human-written because AI tools were not yet available or widely used at that time. Therefore, this dataset can serve as a reference standard for "purely human code." Specifically, this dataset is input into the model for prediction, and the changes in model output are observed under different threshold settings to select a threshold that meets the preset false positive control objective. The core of this process is to adjust the classification boundary so that the model maintains its detection capabilities while reducing the risk of misclassifying normal student code. In terms of classification strategy, we introduced a three-classification mechanism, dividing the model output into three categories: "AI-generated," "human-written," and "unknown." The "unknown" category is used to identify samples with confidence levels in the middle range, thus avoiding forced judgments by the model when evidence is insufficient. This mechanism provides a buffer for model decision-making to some extent and supports subsequent manual review. After calibration, the model is deployed to a large-scale real-world dataset, and uniform feature extraction and batch prediction are performed on all code samples. This process extends the model from the experimental environment to real-world application scenarios, providing a data foundation for subsequent results analysis and behavioral research. Furthermore, the model's inference efficiency is evaluated during deployment to ensure stable operation under large-scale data conditions. The overall process embodies a complete closed-loop design from model training to practical application.

4.7 Detection Analysis and Temporal Trends

To further understand the application performance of the model in real-world scenarios from a data perspective and to analyze the changing trends of AI-generated code over time, we constructed two analysis modules: one is a code style comparison analysis based on multi-dimensional features, and the other is a behavioral trend analysis framework based on time series.

4.7.1 AI code feature representation in the post-ChatGPT era

To examine the fundamental differences in programming style between AI-generated code and student-created code, this study conducted a systematic multidimensional feature comparison analysis on 5,000 recent samples from June 2024 onwards. During data processing, only samples explicitly predicted as "AI" and "Human" were retained, while the "Unknown" category was excluded to improve the reliability of the statistical results. In feature design, 16 dimensions were extracted, covering aspects such as code size, comment behavior, identifier usage, and structural complexity. Specifically, these included: character count, The extracted features include line count, comment ratio, English comment hints, identifier quantity and diversity, and naming style (`snake_case` vs. `camelCase`), and control flow and nesting depth. In terms of analysis methodology, the mean and median were calculated for both AI and Human samples, and the top features with the greatest differences were selected for focused comparison to identify the most discriminative patterns. In addition, we also combined sample-level analysis methods to further observe the high-confidence prediction results, in order to help understand the model's performance under different code modes.

4.7.2 ChatGPT adoption rate trend analysis before and after release

In the final stage of the study, we further analyzed the changing trends in student AI usage over time. To explore whether the proportion of AI-generated content in student code significantly increased after the release of ChatGPT. To this end, this study constructed a time-series-based statistical analysis framework. First, 44,258 code submission records were correlated with model prediction results and aggregated by month (YYYY-MM) based on timestamps. For each month, the following indicators were statistically analyzed: total number of samples, number predicted as AI, AI prediction rate (AI percentage), and average AI probability. This allows us to observe the changing trends in AI usage over time. In defining key time points, we used the ChatGPT release as a significant dividing point, categorizing the data into a "pre-AI phase" and a "post-AI phase," and compares the data distribution between the two phases. Furthermore, to enhance analytical precision, finer-grained time divisions are conducted within the post-AI phase to observe the dynamic evolution of trends. Statistically, non-parametric tests (such as the Mann–Whitney U test) are employed to compare the data distribution across different time phases, and effect size indices are used to assess the actual impact of these differences. Through this analytical framework, this thesis provides a systematic method for time trend analysis in the subsequent results section, enabling a more comprehensive assessment of the potential impact of AI tool adoption on students' programming behavior.

5 Results and Analysis

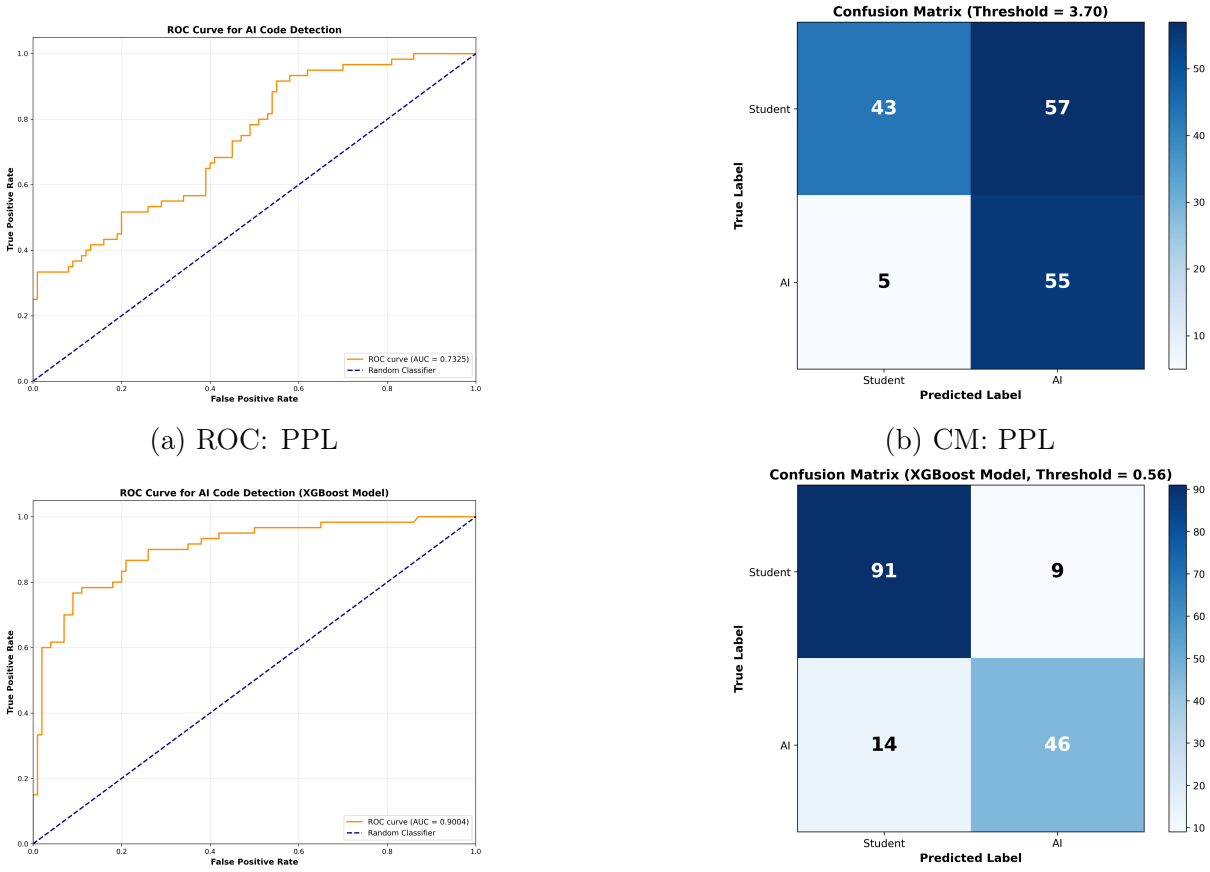
5.1 Model Performance Evaluation

In the final experimental phase, we systematically evaluated the proposed multi-feature fusion model using a dataset of 160 manually labeled code samples, as described in Section 4.1. The dataset includes both AI-generated code and student submissions from the SmartBeans platform. While some degree of label noise may exist due to potential AI-assisted student work, the dataset reflects realistic conditions in programming education. The proposed model was compared with a single-feature baseline model constructed from platform data to assess its effectiveness. Experimental results show that the XGBoost fusion model significantly outperforms the baseline model in all core metrics, demonstrating stronger discriminative ability and higher practical application value. Specifically, the model achieves an accuracy of 85.62%, a precision of 83.64%, and an AUC of 0.9004 on the test set, demonstrating excellent classification performance and probabilistic discrimination ability. Table 1 summarizes the comparison results of the two methods. Table 1 summarizes the comparison results of the two methods.

Table 1: Performance Comparison between Baseline and XGBoost Fusion Model

Metric	Baseline (PPL)	XGBoost Fusion	Delta (Gain)
Accuracy	61.25%	85.62%	+39.8%
Precision	49.11%	83.64%	+34.5%
FPR*	57.00%	9.00%	−42.0%
AUC	0.7325	0.9004	+22.9%

* *FPR: False Positive Rate (Critical Improvement).*



(a) ROC: PPL

(b) CM: PPL

(c) ROC: XGBoost

(d) CM: XGBoost

Figure 6: Model Performance Evaluation: Comparison between PPL Baseline (top) and XGBoost Fusion Model (bottom).

As the table shows, while the single-feature model based on perplexity exhibits high recall, its FPR is over 51%, demonstrating significant "oversensitivity." This means that although the model can capture most AI code, it will incorrectly label a large amount of student-generated code, which is unacceptable in real-world teaching scenarios. In contrast, the multi-feature fusion model maintains high detection capability while significantly reducing the risk of misclassification. In particular, the FPR decreased from approximately 51% to 9%, a decisive improvement. The ROC curves show that the fusion model maintains a high ROC and a low FPR across different thresholds. The curves are generally close to the upper left corner, indicating good discriminative ability and stability. Further analysis of the confusion matrix reveals that the model successfully

controls the misclassification rate at a low level, with approximately 91% of samples correctly classified and only about 9% misclassified. This performance demonstrates the model's strong generalization ability in complex scenarios. From an application perspective, the significant improvement in precision 83.64% has important ethical implications. In educational settings, misclassifying student code as AI-generated can lead to academic integrity controversies. Therefore, reducing false positives is more crucial than simply increasing recall. The model in this study achieves a good balance in this regard, ensuring detection capability while minimizing false negatives for students. From a methodological perspective, this performance improvement primarily stems from the synergistic effect of multi-dimensional features. Unlike a single perplexity metric, the fusion model can simultaneously capture both statistical and structural features of the code. For example, when low perplexity is accompanied by features such as high comment ratios and standardized naming styles, the model can more accurately identify AI-generated patterns; conversely, it is less likely to misclassify student code that is anomalous in only a single dimension.

In conclusion, experimental results demonstrate that the multi-feature fusion method effectively overcomes the limitations of single-feature models, achieving a good balance between detection capability and false positive control, laying the foundation for large-scale applications in the future

5.2 Three Model Comparative

This section presents the experimental results based on the comparative design described in section 4.5 The analysis focuses on three aspects, including detection performance on labeled samples, prediction distribution on real platform data, and consistency between different methods. First, we examine the detection performance on labeled samples. The results show clear differences between the three methods. GPTZero achieves an accuracy of 84.44%, but its recall is only 8.33%. This means that most AI-generated code is not detected and is instead classified as human-written. Its AUC is 0.4976, which is close to random performance. This result shows that GPTZero is highly conservative and only identifies AI code when the signal is extremely strong. DetectGPT shows a different pattern. Its recall increases to 26.67%, which means it can detect more AI-generated code than GPTZero. However, its accuracy drops to 64.72%, and the F1 score remains low at 0.1975. This indicates that while DetectGPT improves detection ability, it also produces more false positives. In other words, it tends to misclassify some human-written code as AI-generated. In contrast, the proposed CodeBERT fusion model performs well across all metrics. The accuracy reaches 91.39%, and the recall increases significantly to 75%. The AUC is 0.9462, which shows a strong ability to distinguish between AI-generated and human-written code. These results demonstrate that a model designed for code features can better capture both structural patterns and semantic information.

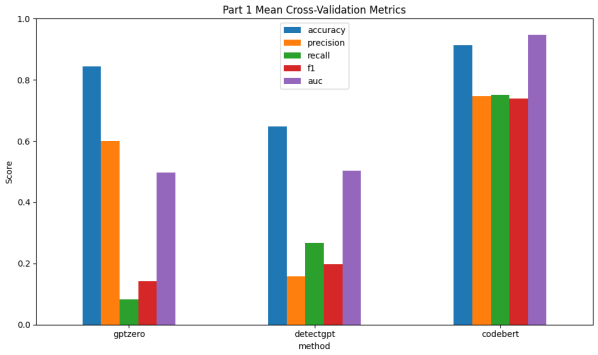


Figure 7: Performance Comparison of AI Code Detection Models on Labeled Dataset

Second, we analyze the prediction distribution on real platform data. When applied to 4,500 code submissions from the SmartBeans platform, the three methods produce very different AI prediction rates. GPTZero identifies only 1.40% of the code as AI-generated. This extremely low value reflects its strong tendency to miss AI-generated content. DetectGPT predicts 38.47% of the code as AI-generated. This high value suggests that many human-written codes may be incorrectly flagged as AI. The CodeBERT fusion model provides a more balanced result. Its AI prediction rate is moderate and consistent with its performance on labeled data. This indicates that the model can offer more reliable estimates when applied to real-world student submissions. Finally, we

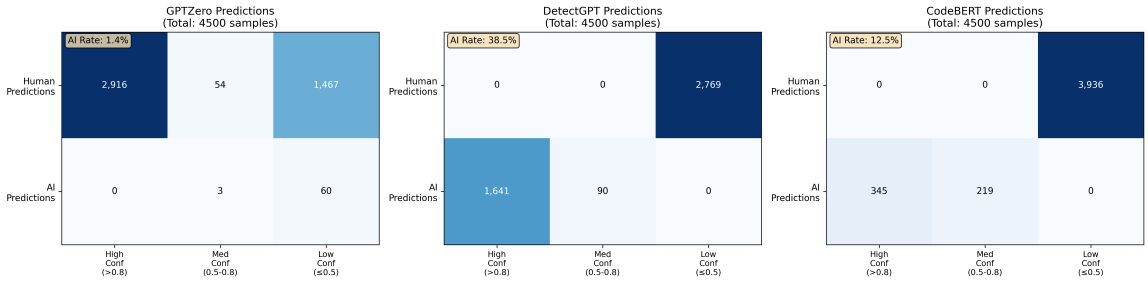


Figure 8: Prediction Confidence Distribution Across Three Models

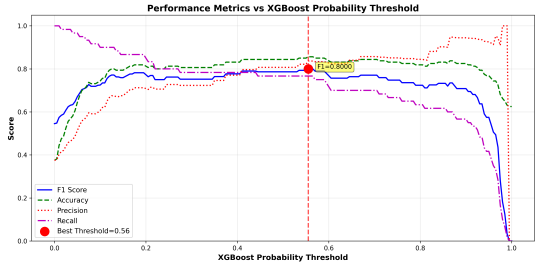
examine the consistency between methods. The agreement rate between GPTZero and DetectGPT is 60.67%. This means that in a large number of cases, the two methods produce different predictions. Such inconsistency shows that general text-based detectors are not stable when applied to programming data. In addition, a considerable number of samples are classified as uncertain, which indicates that many code submissions fall into a gray area where clear decisions are difficult.

Overall, the results show that general text detection methods have clear limitations in the code domain. GPTZero tends to miss most AI-generated code, while DetectGPT tends to over-predict AI and introduce more errors. In contrast, the CodeBERT fusion model achieves a better balance between detection ability and error control. It also shows stronger stability and is more suitable for real educational scenarios.

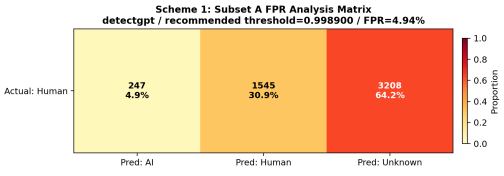
5.3 Cross-Model Consistency Analysis

To evaluate our model's performance, we compared the XGBoost multi-feature fusion model with two common AI detection tools, GPTZero and DetectGPT. The experiments

tested the model's ability to identify AI-generated code and addressed the issue of misclassifying student-generated code. We used the "Pre-AI Era" dataset, which contains 5,000 pieces of code collected before ChatGPT's release on November 1, 2022. This code is entirely human-written. We input this code into our model and predicted it one by one, trying different thresholds to find a point where the false positive rate stayed below 5%. The experiments showed that at an appropriate threshold, the model could keep the false positive rate around 5%. For instance, when using DetectGPT for calibration and setting the threshold at 0.9989, 247 pieces of code were misclassified, making up 4.94% of the total, which met our goal. To account for prediction uncertainty, we introduce a three-category classification scheme: "AI-generated," "human-written," and "unknown." The "unknown" category includes samples with intermediate scores that do not meet high-confidence criteria. The results indicate that 64.16% of the samples fall into the "unknown" category, while 30.90% are classified as human-written and 4.94% as AI-generated. This distribution suggests that a substantial portion of the data cannot be confidently classified. Such behavior is particularly evident in short code snippets or samples with less distinctive features and is primarily due to the internal constraint "too_few_words_for_perturbation," which limits its applicability to short inputs. Under the default threshold (0.7), DetectGPT exhibits an extremely high false positive rate of 97.15%, misclassifying the majority of well-structured student code as AI-generated. To address this issue, we calibrated the decision threshold and identified a more conservative setting that reduces the FPR to approximately 5%. In comparison, GPTZero labels 1,903 samples (38.06%) as "unknown," achieving a higher effective coverage rate of 61.94%. Among the samples that can be processed, the false positive rate under the default threshold (50.0) is only 1.74%, corresponding to 54 misclassified cases. These results suggest that perplexity-based approaches are generally more robust than perturbation-based methods when applied to short or structurally simple code.



(a) XGBoost fusion model performance metrics



(b) FPR calibration matrix based on Subset A

Figure 9: Multi-feature fusion model: CodeBERT + XGBoost

After completing the full-scale deployment, this study further analyzes 5,000 recent submissions collected after June 2024. Scheme 2 is applied to conduct a cross-model consistency analysis. The predictions of GPTZero and DetectGPT are compared directly, and this process helps reveal the limitations of general-purpose detection methods in real teaching environments. The consistency matrix shows that the two methods often disagree with each other. Only 13 samples (0.3%) are jointly classified as AI by both models. This number is very small, which means that without domain-specific design, general detectors can only identify a very limited set of high-risk AI code, and even this part has very low overlap. At the same time, clear conflicts can be observed. A total of 361 samples (7.2%)

are classified differently by the two models, where one predicts AI and the other predicts human. Among these cases, 254 samples are labeled as AI by DetectGPT but human by GPTZero. This pattern shows that DetectGPT is more sensitive, while GPTZero is more conservative. The two models follow different decision tendencies, which leads to unstable results. In contrast, 1,855 samples (37.1%) are consistently classified as human by both models. This indicates that the two methods are more aligned when identifying negative samples, and they tend to default to "human" when the signal is not strong enough. A more important observation comes from the "Unknown"-related results. In total, 2,771 samples (55.4%) involve at least one "Unknown" prediction. DetectGPT alone marks 1,474 samples (29.5%) as uncertain, and GPTZero shows a similar trend. This means that more than half of the code falls into a gray area where the models cannot give a clear decision. This result highlights the importance of the three-class buffering mechanism proposed in this study. In real scenarios, many code samples are short or follow very standard syntax, so their features are not strong enough to support a confident prediction. If a strict binary classification is forced, it may lead to incorrect judgments and even academic disputes. By introducing an "Unknown" category and passing these cases to human review, the system becomes more cautious and better aligned with fairness in educational evaluation. Overall, the consistency matrix in Scheme 2 shows that general detection models lack

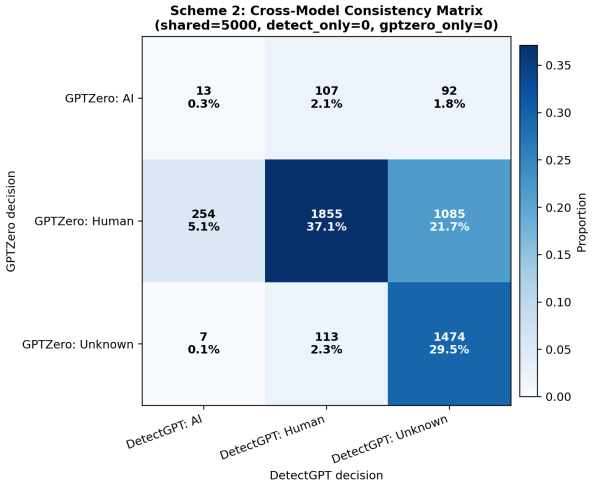


Figure 10: scheme2 post 2024 consistency matrix

stability and agreement when applied to code data. These models do not fully capture structural and semantic features, such as comment patterns or identifier entropy, which are important in programming tasks. As a result, their predictions are often inconsistent in complex real-world scenarios. In comparison, domain-specific models designed for code can provide more reliable and stable results, especially in environments with strict syntax constraints such as the C programming language.

5.4 Temporal Adoption Trends

This section analyzes the usage trend of AI-generated code from a temporal perspective based on 44,250 valid submissions after data cleaning. The analysis focuses on monthly adoption patterns and only includes months with at least 100 samples. In total, 31 months are used to ensure that the results are stable and reliable. The results show that the AI

usage rate changes clearly over time. In the early stage, the usage level is relatively low. For example, in December 2021, the AI prediction rate is 7.69% with a sample size of 416. This represents one of the lowest stable points in the dataset and indicates that most code submissions were written by students themselves. After the release of ChatGPT, the trend begins to increase. Some early months show extreme values, such as October 2022 with an AI rate of 47.37%, but the sample size in that month is only 19. Such results are not reliable and are not used as the main reference. When focusing on months with sufficient data, the upward trend becomes clearer and more stable. In the later stage, the AI usage rate continues to rise and reaches a relatively high level. The highest value appears in March 2025, with an AI rate of 28.41% and a sample size of 3,168. Compared with the early stage, this shows a clear increase and confirms that AI-generated code has become more common in student submissions. From an overall perspective, the trend can be divided into three stages. In the first stage, from 2021 to early 2022, the AI rate remains low and stable, generally below 10%. In the second stage, from late 2022 to 2023, the AI usage rate increases quickly as AI tools become widely available. In the third stage, from 2024 to 2025, the AI usage rate stabilizes at a higher level, around 25% to 28%, and forms a plateau. This plateau is an important signal. It shows that AI is no longer a new tool, but has become part of regular programming practice. At the same time, the stable trend suggests that students are using AI in a more controlled way, such as for debugging, completing code, or generating ideas, instead of relying on it completely. From an educational perspective, this trend reflects a clear shift. The key question is no longer whether students use AI, but how they use it. As AI becomes part of everyday learning, teaching methods and evaluation strategies need to adjust to this change. To further

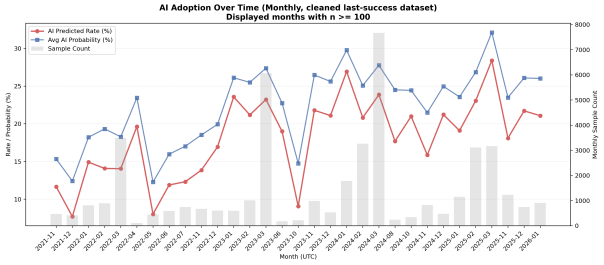


Figure 11: 2022-2024 ChatGPT adoption rate trend

understand the impact of LLM on programming education, we analyzed the data after the release of ChatGPT. The cutoff point is set at November 2022. In this period, the dataset includes 36,234 valid submissions across 37 months. Among these submissions, 7,295 are predicted as AI-generated, and the overall AI prediction rate is 20.13%. This means that about one out of every five code submissions shows clear signs of AI involvement. This value can be seen as a stable reference level for current student behavior. The level of AI usage can be further examined through risk categories provided by the CodeBERT fusion model. Most submissions still belong to the low-risk group, which means they are likely written by students. However, the medium-risk and high-risk groups show clear changes over time. The medium-risk group includes code with partial AI characteristics, while the high-risk group includes code that is highly similar to AI-generated outputs. In several months during 2024 and 2025, the proportion of high-risk submissions increases and reaches more than 10% of all monthly submissions. This is an important signal. It shows that some students are not only using AI for simple help, but are relying on it to generate large parts of their code. These submissions often follow consistent structure,

regular patterns, and typical styles of language models. When compared with the period before AI tools became widely available, this 20.13% level shows a clear shift. AI is no longer a rare or occasional tool. It has become part of normal programming practice for many students. This change reflects both the efficiency of AI tools and their strong influence on learning behavior.

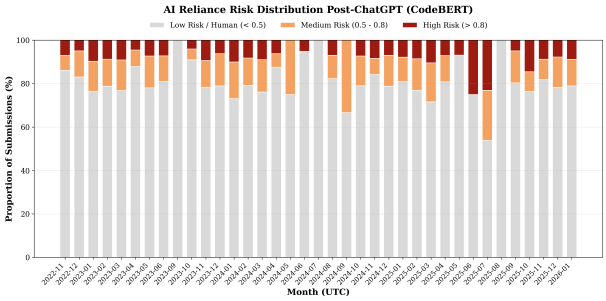


Figure 12: Longitudinal Trend of AI Reliance Risk Post-ChatGPT

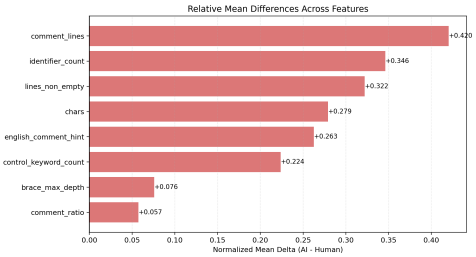
5.5 Characteristic Profiling

Based on sample data from the post-ChatGPT phase, we constructed a 16-dimensional feature space to systematically characterize the features of AI-generated code. Experiments revealed that code labeled as AI by the model exhibits the following significant characteristics:

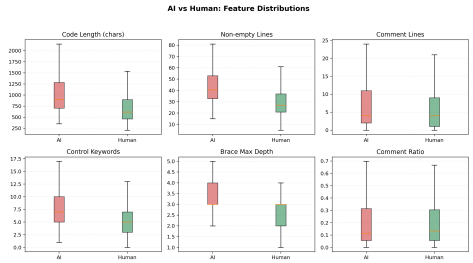
- Scale and Completeness: AI code is typically longer than student code, an average increase of 306.75 characters and 15.19 non-empty lines, indicating a greater tendency to generate structurally complete and logically closed code snippets rather than partial implementations.
- Comment Standardization: AI code has significantly more lines of comments than human code, an average increase of 5.20 lines, and the probability of using English comments is increased by 14.6%, demonstrating a style closer to standard technical documentation.
- Structural Complexity: AI-generated code exhibits deeper nested structures like maximum bracket depth increases by 0.24 and contains more control flow keywords such as if, for, while, etc., an average increase of 1.79, reflecting a more systematic logical organization.
- Naming Consistency: AI code exhibits higher consistency in identifier naming, e.g., uniformly using the `snake_case` style, but its lexical diversity is relatively low, tending to reuse generic variable names. In contrast, student code displays stronger individual characteristics and irregularities.
- Task-Specific Distribution and Reliance: The adoption of AI tools is not uniform across the curriculum. Figure 13(c) presents the AI detection rate and "Unknown" classification rate ranked across various programming tasks. Certain assignments, such as Task 78 and Task 401, exhibit AI detection rates exceeding 40%, indicating a high degree of student reliance. These tasks often involve classic algorithmic

problems where LLMs have seen numerous similar implementations during training. Tasks like 410 and 417 show a high "Unknown" rate. This is primarily attributed to the short length of the required solutions, which limits the effectiveness of perturbation-based detection DetectGPT due to insufficient word counts for statistical significance.

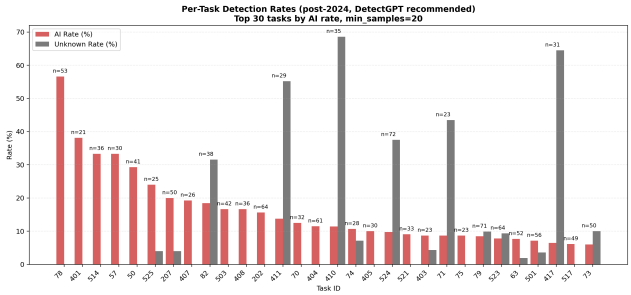
Furthermore, to enhance the interpretability of the analysis, the study further extracted high-confidence AI samples for qualitative analysis, generating sample-level feature matrices and summary statistical reports. These results validate, from a data perspective, the systematic differences in style and structure among AI code. These features collectively constitute a "statistical profile" of the AI code, providing important basis for subsequent model judgment.



(a) Relative differences in mean features.



(b) Key code feature distributions.



(c) AI detection rate ranking across various tasks.

Figure 13: Comprehensive Analysis of AI Code Features and Detection Rates.

5.6 Example: Human vs. AI

To more intuitively demonstrate the difference between AI-generated code and human-written code, this article selects a typical case for analysis. In this case, AI and human code solved the same Josephus problem. This task was selected because it is commonly used in introductory programming exercises and allows for multiple valid implementations, making it suitable for comparing different coding styles. Moreover, the variation in solution structures makes potential differences between human-written and AI-generated code more observable.characteristics. The AI-generated solution exhibits a consistent and standardized coding style. It strictly adheres to naming conventions like `snake_case`, has a clear structure, complete logic, and almost no unnecessary comments or debugging traces. Loops, variables, and conditional statements in the code are all well-ordered, and each operation is closely related to the core logic.

In contrast, the human-written solution shows greater variability in both structure and presentation. The code includes extended comments, such as a detailed problem descrip-

tion in German, as well as traces of intermediate development, including commented-out debugging statements such as `printf`. Variable naming is less consistent, and the implementation reflects a more iterative development process.

This case clearly demonstrates the stylistic differences between the two: AI-generated code tends to be decontextualized and logically pure, like a sophisticated test-taking machine; human code, on the other hand, is full of personalized touches, including the context of the question and debugging notes, like a hand-carved work of art. This comparison not only verifies the model's ability to distinguish between AI and human code but also helps educators understand the diversity of students' coding styles, providing a reference for teaching management and academic integrity assessment.

Figure 11: Comparison of AI-generated and human-written code samples (excerpt).

To illustrate the differences between AI-generated and human-written code, we present representative excerpts from student submissions of the Josephus problem.

Listings 1 and 2 show the comparison between AI-generated and human-written implementations for the same logic.

```
// excerpt
while (i<n)
    {
        j=0;
        while (j<k)
        {
            if (ind==n-1)
            {
                ind=-1;
            }
            ind++;
            if(initial[ind]!=0)
            {
                j++;
            }
        }
        initial[ind]=0;
        joseph[ind]=i+1;

        i++;
}
```

Listing 1: **AI-generated code**

```
// excerpt
//iterate over array
    while(r<=n){

        for(int i=0; i<n; i++){
            if(init[i]!=-1){j++;} //increase count j only if element at i isn't remov
            if(j==k){              //k-th element reached
                init[i] = -1;      //flag for removed element
                res[i] = r;
                r++;               //increase number of rounds
                j = 0;             //set iter var to 0
            }
        }

        /*
        printf("r:%d\n", r);
        printf("init:");
        for(int i=0; i<n; i++){printf("%d ",init[i]);}
        printf("\n");
        printf("res:  ");
        for(int i=0; i<n; i++){printf("%d ",res[i]);}
        printf("\n");
        */
    }
}
```

Listing 2: **Human-written code**

The AI-generated code tends to use more deeply nested control structures and explicit index manipulation, while the human-written code adopts a more compact loop structure and includes additional comments to explain the logic. These differences are consistent with the observed patterns in structural complexity and code readability. The full versions of both code samples are provided in Appendix 1.

5.7 summary

This section shows the benefits of the proposed XGBoost multi-feature fusion model in AI code detection through extensive experiments. Overall, the model performs very well in metrics like accuracy, precision, and AUC. It successfully keeps the false positive rate around 9%, demonstrating both accuracy and strength in real-world teaching settings. Comparative experiments with other models reveal that general text detection models have major drawbacks in the code domain. In contrast, specialized models that focus on code features show clear advantages in recognition rate, false positive control, and inference speed. Additionally, time-trend analysis indicates a notable rise in the number of students using AI-generated code since ChatGPT was released. This trend has stabilized at a high level, showing that AI has become a regular tool for students in programming. Feature analysis and case studies highlight clear differences between AI code and human code in terms of structure, annotation, variable naming, and logical organization. This

information offers measurable support for teaching management and academic integrity assessments.

In conclusion, the experimental results in this section not only confirm the effectiveness of the model but also provide useful data and methods for educational practice, setting a strong basis for further discussions and applications.

6 Conclusion and Future Work

6.1 Research Summary

This study examines the impact of large language models on programming education and proposes a practical framework for detecting AI-generated code. The method combines CodeBERT semantic representations with an XGBoost classifier to capture both structural and semantic features of C programs. Experimental results show that the proposed multi-feature fusion model performs better than general-purpose text detection methods such as GPTZero and DetectGPT. On real-world data from the SmartBeans platform, the model achieves a recall rate of 75% while keeping the false positive rate at a reasonable level. In addition, the longitudinal analysis of large-scale submissions shows that the AI usage rate has stabilized at about 20.13% after the release of ChatGPT. This result indicates that AI-assisted programming is no longer occasional, but has become a common part of student practice.

6.2 Threats to Validity

Although the study provides consistent empirical results, several limitations should be noted.

Internal Validity: The labeled dataset used for training contains only 160 samples, which may limit the stability of the model. To reduce this risk, 5-fold cross-validation and regularization are applied during training. Another issue is that the "human-written" samples may include unknown AI assistance. This type of label noise cannot be completely removed. However, it reflects real-world conditions, and the model shows stable performance during the calibration process.

External Validity: In the large-scale analysis of 153,875 submissions, ground truth labels are not available. Therefore, the reported 20.13% AI rate should be understood as a predicted proportion rather than an exact value. It represents code that shows strong similarity to AI-generated patterns. To reduce potential bias, a three-class mechanism is used, and the analysis focuses on overall trends instead of individual cases.

6.3 Future Work

Future research can extend this work in several directions. First, the dataset can be expanded to include more programming languages, such as Python and Java, as well as outputs from different large language models. This will help test whether the proposed features can generalize across domains. Second, more detailed detection can be explored. For example, identifying specific AI-generated code segments within a single submission may provide more useful feedback for teaching. Finally, as AI tools become more common in education, the goal of detection should shift from simple identification to understanding

usage patterns. This may help educators design better teaching strategies and guide students to use AI in a more appropriate way.

References

[Bao+24] Guangsheng Bao et al. "Fast-DetectGPT: Efficient Zero-Shot Detection of Machine-Generated Text via Conditional Probability Curvature". In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=Bpcgcr8E8Z.

[Bav+22] Mohammad Bavarian et al. *Efficient Training of Language Models to Fill in the Middle*. July 28, 2022. DOI: 10.48550/arXiv.2207.14255. arXiv: 2207.14255 [cs]. URL: http://arxiv.org/abs/2207.14255.

[Bro+20] Tom B. Brown et al. "Language Models Are Few-Shot Learners". In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS '20. Red Hook, NY, USA: Curran Associates Inc., Dec. 6, 2020, pp. 1877–1901. URL: https://dl.acm.org/doi/10.5555/3495724.3495883.

[Bul+24] Luana Bulla et al. "EX-CODE: A Robust and Explainable Model to Detect AI-Generated Code". In: *Information* 15.12 (2024). ISSN: 2078-2489. URL: https://www.mdpi.com/2078-2489/15/12/819.

[Bus23] Alessio Buscemi. *A Comparative Study of Code Generation using ChatGPT 3.5 across 10 Programming Languages*. 2023. arXiv: 2308.04477 [cs.SE]. URL: https://arxiv.org/abs/2308.04477.

[Che+21] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. July 2021. DOI: 10.48550/arXiv.2107.03374. eprint: 2107.03374 (cs).

[Cho+22] Aakanksha Chowdhery et al. *PaLM: Scaling Language Modeling with Pathways*. 2022. arXiv: 2204.02311 [cs.CL]. URL: https://arxiv.org/abs/2204.02311.

[CIL25] Domenico Cotroneo, Cristina Improta, and Pietro Liguori. *Human-Written vs. AI-Generated Code: A Large-Scale Study of Defects, Vulnerabilities, and Complexity*. 2025. arXiv: 2508.21634 [cs.SE]. URL: https://arxiv.org/abs/2508.21634.

[ENS24] Tasneem Muhammed Eltabakh, Nada Nabil Soudi, and Doaa Shawky. "Quality of AI-Generated vs. Human-Generated Code". In: *2024 34th International Conference on Computer Theory and Applications (ICCTA)*. 2024, pp. 200–205. DOI: 10.1109/ICCTA64612.2024.10974782.

[Fan+23] Angela Fan et al. "Large Language Models for Software Engineering: Survey and Open Problems". In: *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. 2023, pp. 31–53. DOI: 10.1109/ICSE-FoSE59343.2023.00008.

[Fen+20] Zhangyin Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. arXiv: 2002.08155 [cs.CL]. URL: https://arxiv.org/abs/2002.08155.

[GC25] Zijun Gao and Shihming Chen. "Artificial Intelligence-Driven Transformation of Educational Governance Models". In: *International Journal of Sociologies and Anthropologies Science Reviews* 5.5 (Aug. 2025), pp. 709–720. DOI: 10.60027/ijsasr.2025.7208. URL: https://so07.tci-thaijo.org/index.php/IJSASR/article/view/7208.

[Guo+24] Xiaowei Guo et al. "Enhancing Robustness of Code Authorship Attribution through Expert Feature Knowledge". In: ISSTA 2024. Vienna, Austria: Association for Computing Machinery, 2024, pp. 199–209. ISBN: 9798400706127. DOI: 10.1145/3650212.3652121. URL: https://doi.org/10.1145/3650212.3652121.

[Hin+16] Abram Hindle et al. "On the Naturalness of Software". In: *Commun. ACM* 59.5 (Apr. 26, 2016), pp. 122–131. ISSN: 0001-0782. DOI: 10.1145/2902362. URL: https://dl.acm.org/doi/10.1145/2902362.

[Jia+26] Juyong Jiang et al. "A Survey on Large Language Models for Code Generation". In: *ACM Transactions on Software Engineering and Methodology* 35.2 (Feb. 2026), pp. 1–72. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3747588. eprint: 2406.00515 (cs).

[JSR25] Kush Jain, Gabriel Synnaeve, and Baptiste Roziere. "TestGenEval: A Real World Unit Test Generation and Test Completion Benchmark". In: *The Thirteenth International Conference on Learning Representations*. 2025. URL: https://openreview.net/forum?id=7o6SG5gVev.

[KA07] Kenneth R. Koedinger and Vincent Aleven. "Exploring the Assistance Dilemma in Experiments with Cognitive Tutors". In: *Educational Psychology Review* 19.3 (2007), pp. 239–264. ISSN: 1573-336X. DOI: 10.1007/s10648-007-9049-0.

[Kam+24] Faouzi Kamoun et al. "Exploring Students' and Faculty's Knowledge, Attitudes, and Perceptions Towards ChatGPT: A Cross-Sectional Empirical Study". In: *Journal of Information Technology Education: Research* 23 (Feb. 12, 2024), p. 004. URL: https://www.informingscience.org/Publications/5239.

[Kap+20] Jared Kaplan et al. *Scaling Laws for Neural Language Models*. 2020. arXiv: 2001.08361 [cs.LG]. URL: https://arxiv.org/abs/2001.08361.

[Kha+23] Muhammad Fawad Akbar Khan et al. *Assessing the Promise and Pitfalls of ChatGPT for Automated Code Generation*. 2023. arXiv: 2311.02640 [cs.SE]. URL: https://arxiv.org/abs/2311.02640.

[Kum+26] Samarth Sanjeev Kumakale et al. "Detection of Vulnerabilities in AI-Generated Java Code Using Random Forest Algorithm". In: *2026 International Conference on Intelligent and Innovative Technologies in Computing, Electrical and Electronics (IITCEE)*. Jan. 2026, pp. 1–6. DOI: 10.1109/IITCEE67948.2026.11394305.

[KVT24] Sotiris Kotsiantis, Vassilios Verykios, and Manolis Tzagarakis. "AI-Assisted Programming Tasks Using Code Embeddings and Transformers". In: *Electronics* 13.4 (Feb. 2024). ISSN: 2079-9292. DOI: 10.3390/electronics13040767.

[Liu+24a] Shengchao Liu et al. "Does DetectGPT Fully Utilize Perturbation? Bridging Selective Perturbation to Fine-tuned Contrastive Learning Detector Would Be Better". In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Lun-Wei Ku, Andre Martins, and Vivek Srikumar. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 1874–1889. DOI: 10.18653/v1/2024.acl-long.103.

[Liu+24b] Shengchao Liu et al. "Does DetectGPT Fully Utilize Perturbation? Bridging Selective Perturbation to Fine-tuned Contrastive Learning Detector would be Better". In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Lun-Wei Ku, Andre Martins, and Vivek Srikumar. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 1874–1889. DOI: 10.18653/v1/2024.acl-long.103. URL: https://aclanthology.org/2024.acl-long.103/.

[LK24] Xurong Liu and Leilei Kong. "AI Text Detection Method Based on Perplexity Features with Strided Sliding Window." In: *CLEF (Working Notes)*. 2024, pp. 2755–2760.

[LPP23] E. Grace Lydia, P. Vidhyavathi, and P. Malathi. "A STUDY ON "AI IN EDUCATION: OPPORTUNITIES AND CHALLENGES FOR PERSONALIZED LEARNING". In: *Industrial Engineering Journal* 52.05 (2023), pp. 750–759. ISSN: 09702555. DOI: 10.36893/IEJ.2023.V52I05.750-759. URL: http://www.journal-iiie-india.com/1_may_23/82_online.pdf.

[LZ25] Zhi Li and Wenxiang Zhang. "Technology in Education: Addressing Legal and Governance Challenges in the Digital Era". In: *Education and Information Technologies* 30.7 (May 1, 2025), pp. 8413–8443. ISSN: 1573-7608. DOI: 10.1007/s10639-024-13036-9. URL: https://doi.org/10.1007/s10639-024-13036-9.

[Mit+23] Eric Mitchell et al. *DetectGPT: Zero-Shot Machine-Generated Text Detection using Probability Curvature*. 2023. arXiv: 2301.11305 [cs.CL]. URL: https://arxiv.org/abs/2301.11305.

[MRA25] Adiba Mahmud, Yasmeen Rawajfih, and Ross Arnold. "Forensic Detection and Attribution of AI-Generated Code: A Multi-Classifier Approach with CVE Validation". In: *2025 IEEE 16th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. 2025, pp. 0108–0115. DOI: 10.1109/IEMCON67450.2025.11381067.

[Ngu+24] Phuong T. Nguyen et al. "GPTSniffer: A CodeBERT-based Classifier to Detect Source Code Written by ChatGPT". In: *Journal of Systems and Software* 214 (2024), p. 112059. ISSN: 0164-1212. DOI: 10.1016/j.jss.2024.112059. URL: https://www.sciencedirect.com/science/article/pii/S0164121224001043.

[Nia+24] Sikander Niaz et al. "AI FOR INCLUSIVE EDUCATIONAL GOVERNANCE AND DIGITAL EQUITY EXAMINING THE IMPACT OF AI ADOPTION AND OPEN DATA ON COMMUNITY TRUST AND POLICY EFFECTIVENESS". In: *Contemporary Journal of Social Science Review* 2.04 (Oct. 22, 2024), pp. 2557–2567. ISSN: 3006-1466. DOI: 10.63878/cjssr.v2i04.1502.

URL: https://contemporaryjournal.com/index.php/14/article/view/1502.

[Ore+26] Daniil Orel et al. *AICD Bench: A Challenging Benchmark for AI-Generated Code Detection*. Feb. 2026. DOI: 10.48550/arXiv.2602.02079. eprint: 2602.02079 (cs).

[PCP25] Isabel Pinho, António Pedro Costa, and Cláudia Pinho. "Generative AI Governance Model in Educational Research". In: *Frontiers in Education* 10 (July 10, 2025). ISSN: 2504-284X. DOI: 10.3389/feduc.2025.1594343. URL: https://www.frontiersin.org/journals/education/articles/10.3389/feduc.2025.1594343/full.

[PCS24] Eric Poitras, Brent Crane, and Angela Siegel. "Generative AI in Introductory Programming Instruction: Examining the Assistance Dilemma with LLM-Based Code Generators". In: *Proceedings of the 2024 on ACM Virtual Global Computing Education Conference V. 1*. SIGCSE Virtual 2024. New York, NY, USA: Association for Computing Machinery, Dec. 5, 2024, pp. 186–192. ISBN: 979-8-4007-0598-4. DOI: 10.1145/3649165.3690111. URL: https://dl.acm.org/doi/10.1145/3649165.3690111.

[PSS24] Abhi Patel, Kazi Zakia Sultana, and Bharath K. Samanthula. "A Comparative Analysis between AI Generated Code and Human Written Code: A Preliminary Study". In: *2024 IEEE International Conference on Big Data (BigData)*. 2024, pp. 7521–7529. DOI: 10.1109/BigData62323.2024.10825958. URL: https://ieeexplore.ieee.org/document/10825958.

[Shi+24] Yuling Shi et al. *Between Lines of Code: Unraveling the Distinct Patterns of Machine and Human Programmers*. 2024. arXiv: 2401.06461 [cs.SE]. URL: https://arxiv.org/abs/2401.06461.

[Spi+23] Oleksandr V. Spivakovsky et al. "INSTITUTIONAL POLICIES ON ARTIFICIAL INTELLIGENCE IN UNIVERSITY LEARNING, TEACHING AND RESEARCH". In: *Information Technologies and Learning Tools* 97.5 (Oct. 30, 2023), pp. 181–202. ISSN: 2076-8184. DOI: 10.33407/itlt.v97i5.5395. URL: https://journal.iitta.gov.ua/index.php/itlt/article/view/5395.

[Su+23] Jinyan Su et al. "DetectLLM: Leveraging Log Rank Information for Zero-Shot Detection of Machine-Generated Text". In: *Findings of the Association for Computational Linguistics: EMNLP 2023*. Ed. by Houda Bouamor, Juan Pino, and Kalika Bali. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 12395–12412. DOI: 10.18653/v1/2023.findings-emnlp.827. URL: https://aclanthology.org/2023.findings-emnlp.827/.

[Suh+24] Hyunjae Suh et al. *An Empirical Study on Automatically Detecting AI-Generated Source Code: How Far Are We?* 2024. arXiv: 2411.04299 [cs.SE]. URL: https://arxiv.org/abs/2411.04299.

[Tay+23] Zachary Taylor et al. "Plagiarism in Entry-Level Computer Science Courses Using ChatGPT". In: *2023 Congress in Computer Science, Computer Engineering, & Applied Computing (CSCE)*. 2023, pp. 1135–1139. DOI: 10.1109/CSCE60160.2023.00189.

[VRW23] Veniamin Veselovsky, Manoel Horta Ribeiro, and Robert West. *Artificial Artificial Artificial Intelligence: Crowd Workers Widely Use Large Language Models for Text Production Tasks*. 2023. arXiv: 2306.07899 [cs.CL]. URL: https://arxiv.org/abs/2306.07899.

[Wan+21] Yue Wang et al. *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation*. 2021. arXiv: 2109.00859 [cs.CL]. URL: https://arxiv.org/abs/2109.00859.

[XS24] Zhenyu Xu and Victor S. Sheng. "Detecting AI-Generated Code Assignments Using Perplexity of Large Language Models". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 38.21 (2024), pp. 23155–23162. ISSN: 2374-3468, 2159-5399. DOI: 10.1609/aaai.v38i21.30361. URL: https://ojs.aaai.org/index.php/AAAI/article/view/30361.

[Xu+25] Jinwei Xu et al. "One Size Does Not Fit All: Investigating Efficacy of Perplexity in Detecting LLM-Generated Code". In: *ACM Transactions on Software Engineering and Methodology* (July 2025). ISSN: 1557-7392. DOI: 10.1145/3748506. URL: http://dx.doi.org/10.1145/3748506.

A Additional Materials

In this appendix, we provide full code examples for the Josephus Problem implementation, contrasting the structured approach of AI-generated code with the commented style of human-written code.

A.1 AI-generated Code Sample

The following implementation was generated by AI. It demonstrates a logical but concise structure.

```c
#include <stdio.h>
int main()
{
    int n, k;
    scanf("%d %d", &n, &k);
    int initial[n];
    int joseph[n];
    int i=0, j=0, ind=0;
    while (i<n)
    {
        initial[i]=i+1;
        i++;
    }
    i=0;
    while (i<n)
    {
        j=0;
        while (j<k)
        {
            if (ind==n-1)
            {
                ind=-1;
            }
            ind++;
            if(initial[ind]!=0)
            {
                j++;
            }
        }
        initial[ind]=0;
        joseph[ind]=i+1;

        i++;
    }
    i=1;
    while (i<n)
    {
        printf("%d ",joseph[i]);
        i++;
    }
    printf("%d ",joseph[0]);
    return 0;


```

```
45
46 }
```

A.2 Human-written Code Sample

The following implementation is from a real student submission. It includes extensive German comments and a different iteration logic.

```
1  #include <stdio.h>
2
3  /*
4  Das Josephus-Problem kann hier auf Wikipedia nachgelesen werden.
5
6  Die Josephus-Permutation der Zahlen 1,...,n für eine Schrittweite k ist die
   ↪ Permutation,
7  die nach folgendem Schema berechnet wird:
8
9  Stellen Sie sich die Zahlen aufsteigend in einem Kreis angeordnet vor.
10 D.h. man beginnt bei 1 zu zählen und wenn man bei n angekommen ist macht man wieder
   ↪ vorne bei 1 weiter.
11 Das ist die ursprüngliche Permutation.
12 Man beginnt vom Anfang k Schritte zu gehen und entfernt das gefundene Element.
13 In der entstehenden Josephus-Permutation wird an dem Index, wo in der ursprünglichen
   ↪ Permutation das Element stand,
14 die aktuelle Rundenzahl eingetragen.
15 Die Rundenzahl ist die Anzahl der bereits entfernten Elemente.
16 Von dem zu letzt entfernten Element aus geht man wieder k Schritte.
17 Bereits entfernte Elemente sollen bei den Schritten nicht mitgezählt werden.
18 Man geht solange durch den Kreis bis jedes Element der ursprünglichen Liste entfernt
   ↪ wurde.
19 Beispiel
20
21 Für n = 4 und k = 3 sieht die Entstehung der Josephus Permutation so aus:
22
23 Init       1         2         3         4        Rundenzahl
24 [1 2 3 4] [1 2 x 4] [1 x   4] [1     x] [x      ] ursprüngliche Permutation
25 [       ] [    1  ] [  2 1  ] [  2 1 3] [4 2 1 3] Josephus-Permutation
26 Schreiben Sie eine Programm, das zwei Zahlen n > 0 und k > 0 einliest und daraus die
27 Josephus Permutation für die Zahlen 1,...,n mit einer Schrittweite k ausgibt.*/
28
29 int main(){
30     //get input
31     int n, k, j=0, r=1;
32     scanf("%d%d", &n, &k);
33
34     //build arrays
35     int init[n], res[n];
36     for(int i=0; i<n; i++){init[i]=i+1;res[i]=0;}
37
38     //iterate over array
39     while(r<=n){
40
41         for(int i=0; i<n; i++){
42             if(init[i]!=-1){j++;} //increase count j only if element at i isn't
               ↪ removed (flag -1)
43             if(j==k){             //k-th element reached
```

```
                init[i] = -1;      //flag for removed element
                res[i] = r;
                r++;               //increase number of rounds
                j = 0;             //set iter var to 0
            }
        }

        /*
        printf("r:%d\n", r);
        printf("init:");
        for(int i=0; i<n; i++){printf("%d ",init[i]);}
        printf("\n");
        printf("res:  ");
        for(int i=0; i<n; i++){printf("%d ",res[i]);}
        printf("\n");
        */
    }

    //output result
    for(int i=0; i<n; i++){
        printf("%d ", res[i]);
    }

    return 0;

}
```