AIDAR: AI-GENERATED CODE DETECTION VIA AST STRUCTURES

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models can generate convincing solutions to programming problems, making surface cues such as comments, naming, and style increasingly unreliable for identifying AI-generated code. Yet most existing approaches remain text-based and overlook the rich structural information in programs. We first introduce **AST-TED**, a non-neural baseline that compares complete abstract syntax trees (ASTs) using normalized tree edit distance. We then develop **AIDAR** by improving this framework in two ways: Discriminative Subtree Selection identifies informative local AST regions, while Prototype Distribution Matching replaces direct edit-distance comparison by matching distributions over shared structural prototypes for complete ASTs and selected subtrees. On CodeNet100 and a historical contest benchmark (C1), AIDAR achieves 95.05% and 99.00% accuracy, respectively, the highest among the evaluated methods. In a subsequent live contest (C2), we deployed AIDAR as part of a screening workflow that referred 17 of 1,176 full-score submissions for human review. The workflow's eight highest-ranked referrals matched the eight cases human experts flagged as suspected AI-generated code. These results demonstrate the value of AST structure for detection and for prioritizing human investigation. Code is available at `https://anonymous.4open.science/r/AIDAR-3BCA` for anonymous review.

1 INTRODUCTION

Large language models (LLMs) have become increasingly capable of generating code, including solutions to challenging algorithmic problems (Li et al., 2022; Achiam et al., 2023; Chowdhery et al., 2023). This capability has implications for computer science education, programming competitions such as USACO, ICPC, and IOI, and online practice platforms such as LeetCode and Codeforces. How AI tools will reshape these settings remains an open question, but their potential use for cheating is already a practical concern. Detecting whether a program or code snippet was generated by AI has therefore become increasingly important.

Our work is motivated by our collaboration with a nationwide programming contest. The contest is held in person at more than 200 exam labs across multiple cities, most of them organized by universities. Participants submit solutions during the contest to a central testing server, which automatically evaluates each submission and returns the result. Ideally, lab computers would be able to reach this server but no other online resources. In practice, several routes to unauthorized LLM use remain: a participant may access an online LLM from a lab computer with unrestricted internet access; use a smartphone to photograph a problem statement and then manually type the generated solution; or reach an LLM hosted on a university server through a lab's campus network. Network restrictions and invigilation can reduce these risks, but coordinating more than 200 labs leaves room for lapses in oversight. Reliable detection of AI-generated code would help organizers investigate suspicious submissions and protect the contest's fairness and credibility.

AI-generated text detection (Gehrmann et al., 2019; Mitchell et al., 2023; Bao et al., 2024; Kirchenbauer et al., 2023), including attribution to a particular model (Uchendu et al., 2020; Guo et al., 2024), has received considerable research attention. Some detection systems are already used in university library services alongside plagiarism checking. Inspired by this work, AI-generated code detection has attracted attention in both the AI and software engineering communities. Because

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

programs in high-level languages can also be represented as text, text detection methods provide a natural starting point. Existing approaches adapt these methods to the programming domain, for example by using a code-oriented encoder such as CodeBERT in place of a general-purpose text encoder such as BERT (Nguyen et al., 2024). Software engineering research has also explored structural representations such as abstract syntax trees (ASTs). However, structural information is often incorporated only as an auxiliary input: for example, Suh AST Embedding (Suh et al., 2025) serializes an AST to augment a learned representation without explicitly using its tree topology.

As discussed above, existing methods for detecting AI-generated code are largely adapted from AI-generated text detection. These methods were useful in the early stages of AI code generation, when generated programs often had conspicuous textual features: abundant comments, descriptive variable and function names, and a consistent, polished style. At that stage, human experts, text detection methods, and even simple feature-based methods could distinguish many AI-generated programs from human-written ones. However, recent LLMs can readily conceal these cues, while programs retain a rich logical structure that textual features do not capture. For example, when we ask GPT-5.6 to generate code that looks human-written, it immediately removes all comments. In our experience with the programming contest, such surface features are no longer reliable and reveal only a few careless cases of AI use. Even human experts often cannot determine the origin of a submission from the code alone; ambiguous cases require further investigation, such as reviewing examination room recordings. These challenges motivate us to focus on program structure, including control flow, rather than shallow textual features.

To address these challenges, we propose **AIDAR**, an **AI**-Generated Code **D**etection framework based on **A**ST st**R**uctures. To reduce reliance on surface textual features, we first parse each code snippet into an AST and provide only the AST to AIDAR. Thus, AIDAR receives a structural representation of each C/C++, Python, or Java snippet rather than its raw source text. In the basic reference-based framework, a query program is compared with a pool of human-written and AI-generated reference programs and assigned the label of its nearest reference. The central question is how to measure the distance between a query and a reference. We first introduce **AST-TED**, a baseline that measures structural proximity using tree edit distance (TED) between ASTs and assigns each query the label of its nearest reference. AST-TED uses a deterministic algorithm with no learned parameters. Its well-defined structural distance makes predictions transparent and the method straightforward to apply, although a model trained on a particular dataset may achieve higher accuracy. We then extend this framework in two ways. First, to make fuller use of AST topology, we identify subtrees that provide strong signals for distinguishing AI-generated from human-written code. A dynamic programming procedure selects discriminative subtrees, focusing the comparison on the most informative regions. Second, we distill reference structures into a compact set of shared prototypes and represent the query and each reference class as distributions over these prototypes. The final prediction is based on their distributional similarity.

In summary, we make the following contributions in this paper:

- We propose an AST-based framework for AI-generated code detection that explicitly compares program structures beyond surface text. Its basic version, AST-TED, uses normalized tree edit distance to provide a well-defined, non-neural structural baseline.
- We develop AIDAR by extending this framework with Discriminative Subtree Selection and Prototype Distribution Matching. The former selects informative local subtrees alongside the complete AST, while the latter replaces direct tree edit distance comparison with distribution matching over shared structural prototypes.
- Experiments on CodeNet100 and a historical contest (C1) validate AIDAR's effectiveness, while case studies demonstrate its interpretability through selected subtrees and learned prototypes. We further deploy AIDAR in a subsequent live contest (C2) programming contest with 4,316 participants. The results demonstrate strong ranking performance, with all eight top-ranked submissions subsequently identified by human experts as AI-generated.

2 Related Work

AI-Generated Text Detection. Existing detectors broadly use statistical signals or learned representations. GLTR uses token likelihoods and ranks to reveal generation artifacts (Gehrmann et al.,

2019). DetectGPT estimates probability curvature from perturbed passages (Mitchell et al., 2023), whereas Fast-DetectGPT replaces the perturbation step with conditional sampling to estimate conditional probability curvature (Bao et al., 2024). Other approaches insert detectable watermarks during generation (Kirchenbauer et al., 2023). Representation-based approaches learn to distinguish human- from machine-written text and, in some cases, attribute a passage to a particular generator (Uchendu et al., 2020; Guo et al., 2024). Their reliability under rewriting remains a concern: paraphrasing attacks can substantially reduce detection accuracy (Sadasivan et al., 2025).

AI-Generated Code Detection. One line of work adapts token-level signals from text detection to source code. Xu and Sheng combine targeted masking with perplexity-based scores (Xu & Sheng, 2024); DetectCodeGPT evaluates code under semantics-preserving whitespace perturbations (Shi et al., 2025); and CodeRipple uses wavelet analysis to capture local fluctuations in token perplexity (Yao et al., 2026). SWEET restricts watermarking to higher-entropy code tokens to better preserve code quality (Lee et al., 2024). Another line learns code representations. GPTSniffer uses CodeBERT to distinguish human-written from ChatGPT-generated code (Nguyen et al., 2024). CodeGPT-Sensor addresses the same task but fine-tunes UniXcoder with contrastive learning, bringing same-class representations closer and separating different-class representations (Xu et al., 2025). Droid contributes a diverse benchmark and encoder-based detectors, highlighting generalization challenges across languages, domains, generators, and humanized code (Orel et al., 2025). Suh et al. incorporate abstract syntax trees (ASTs) by serializing them for embedding and classification (Suh et al., 2025). This sequence-based treatment leaves tree topology and local structural correspondence between programs implicit, motivating direct structural comparison.

3 Preliminaries and Baseline framework

Abstract syntax tree. An abstract syntax tree (AST) represents the syntactic structure of a program as a tree, while removing surface details such as formatting. Each node corresponds to a syntactic construct in the program, such as a function definition, loop, conditional statement, assignment, operator, or identifier. An edge connects a parent construct to one of its direct syntactic components; for example, an *if* node is connected to its condition and body.

We represent an AST as $T = (V, E, \mathbf{X})$, where V is the set of syntax nodes , E is the set of parent–child edges, and $\mathbf{X}$ contains node features describing properties such as syntax type and operator structure information without any semantic information.

Graph encoder. Given a graph $G = (V, E, \mathbf{X})$, a graph neural network updates each node representation by aggregating information from its neighboring nodes. We denote the resulting node embeddings as

$$\mathbf{H} = \mathrm{GE}(G, \mathbf{X}; \Theta), \tag{1}$$

where θ contains the encoder parameters and each row $\mathbf{h}_v \in \mathbb{R}^d$ of $\mathbf{H}$ represents a node v.

A readout function further aggregates the node embeddings into a graph-level representation:

$$\mathbf{h}_G = \mathrm{Readout}(\mathbf{H}), \tag{2}$$

where $\mathbf{h}_G$ denotes the embedding of graph G.

Reference-based detection. Given a query program x_q, we want to know whether the query is human-written or AI-generated. For the same programming problem, we prepare two reference sets: Human references $\mathcal{R}_H$, collected from real human submissions, and AI references $\mathcal{R}_A$, generated by LLMs. The query is then compared with these two reference sets to determine which source it is more similar to. In this work, the query and reference programs are parsed into ASTs before structural modeling.

AST-TED baseline framework. We introduce AST-TED, our basic AST-based detection framework. It directly measures structural differences between complete ASTs using tree edit distance, without a learned encoder. For query AST T_q and reference AST T_r, the normalized distance is

$$d_{\mathrm{NTED}}(T_q, T_r) = \frac{\mathrm{TED}(T_q, T_r)}{|T_q| + |T_r|}, \tag{3}$$

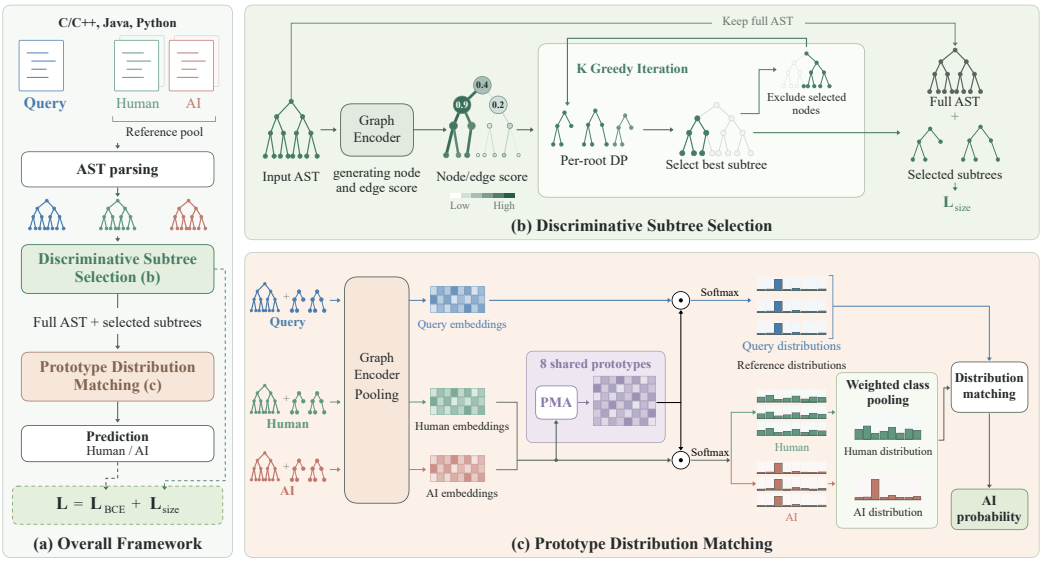


Figure 1: Framework of AIDAR.

where $|T_q|$ and $|T_r|$ are their node counts. The query takes the label of its nearest reference in $\mathcal{R}_H \cup \mathcal{R}_A$ under this distance. AST-TED thus offers a well-defined, non-neural foundation for detecting AI-generated code through tree structure. However, complete-AST comparison does not learn which local structures are most informative for detection. **AIDAR** extends this framework by selecting discriminative local subtrees alongside the complete AST and replacing direct TED comparison with learned structural distribution matching.

4 AIDAR

In this section, we present AIDAR, beginning with an overview and then detailing its two main components: *Discriminative Subtree Selection (DSS)* and *Prototype Distribution Matching (PDM)*.

4.1 Overall Framework

Figure 1 illustrates the overall framework of AIDAR.

First, given a query program and its Human and AI references, we parse all programs into ASTs. Discriminative Subtree Selection (DSS) encodes each AST and assigns scores to candidate subtree roots and parent–child connections. It then selects subtrees sequentially: at each iteration, dynamic programming finds the highest-scoring feasible subtree, and its nodes are excluded from subsequent selections. The complete AST is retained alongside the selected subtrees.

Second, Prototype Distribution Matching (PDM) uses a separate graph encoder, shared by the query and references, to map the complete ASTs and selected subtrees into a common structural embedding space. It constructs shared structural prototypes from the Human and AI references and represents each complete AST or subtree as a distribution over these prototypes. The reference distributions are aggregated separately for the Human and AI classes. Finally, PDM compares each query distribution with the two class distributions and combines the evidence from the complete AST and selected subtrees to produce the final prediction.

4.2 Discriminative Subtree Selection

Given the query and reference ASTs, DSS identifies the local structures that are most informative for distinguishing Human and AI code.

Structural encoding and scoring. We first encode each AST with the DSS graph encoder:

$$\mathbf{H} = \mathrm{GE}(G, \mathbf{X}; \Theta_D), \tag{4}$$

where Θ_D denotes the DSS encoder parameters and $\mathbf{h}_v \in \mathbb{R}^d$ is the representation of node v in $\mathbf{H}$. For each node v, we compute a root score

$$r_v = \mathbf{w}_r^\top \mathbf{h}_v + b_r. \tag{5}$$

where $\mathbf{w}_r$ and b_r are learnable parameters, and a larger r_v indicates a more informative subtree root.

We further evaluate each parent–child connection (u, v). Since different child positions play different syntactic roles, we also encode their relation, such as `condition`, `body`, `left`, or `right`. We maintain a trainable embedding table

$$\mathbf{E}_{\mathrm{rel}} \in \mathbb{R}^{|\mathcal{F}| \times d}, \tag{6}$$

where $\mathcal{F}$ is the set of syntactic relations.

Let $\mathbf{e}_{uv}$ denote the learned embedding of the syntactic relation between u and v. We compute their connection score from the parent, child, and relation representations:

$$\tilde{a}_{uv} = \mathrm{MLP}\left([\mathbf{h}_u \| \mathbf{h}_v \| \mathbf{e}_{uv}]\right). \tag{7}$$

We center the connection scores within each AST and apply a softsign transformation:

$$a_{uv} = \mathrm{softsign}\left(\tilde{a}_{uv} - \frac{1}{|E|} \sum_{(i,j)\in E} \tilde{a}_{ij}\right), \tag{8}$$

where $\mathrm{softsign}(x) = x/(1+|x|)$. The resulting a_{uv} measures the contribution of retaining connection (u, v) in a selected subtree.

Sequential subtree selection. For a candidate connected subtree C rooted at node v, we define its structural score as

$$S(C) = r_v + \sum_{(u,w)\in E(C)} a_{uw}, \tag{9}$$

where $E(C)$ denotes the parent–child connections retained in C. Connection scores can be positive or negative, so adding a branch does not necessarily increase $S(C)$. A larger subtree therefore does not always receive a higher score.

DSS selects subtrees sequentially using a greedy procedure. At each iteration, dynamic programming finds the highest-scoring feasible subtree among the currently available nodes. We select this subtree, exclude its nodes from subsequent iterations, and repeat the process. The resulting subtrees are node-disjoint: no node appears in more than one selected subtree. Each subtree is connected but may omit some descendants of its root. The detailed selection procedure is provided in Appendix B.1.

We denote the K selected query subtrees by $\mathcal{C}_q$ and those of reference R by $\mathcal{C}_R$. The complete AST is retained alongside the selected subtrees for both the query and each reference. Reference subtree selection is independent of the current query and is fixed after training.

Subtree weights. For each reference, let $C_1, \ldots, C_K$ denote its selected subtrees. We normalize their DSS scores to obtain relative weights:

$$\alpha_j = \frac{\exp(S(C_j))}{\sum_{\ell=1}^{K} \exp(S(C_\ell))}, \qquad j = 1, \ldots, K. \tag{10}$$

where α_j measures the relative importance of subtree C_j within the same reference. These weights are used to aggregate reference distributions in Sect. 4.3 and to define the subtree-size penalty in Sect. 4.4.

4.3 PROTOTYPE DISTRIBUTION MATCHING

Given the complete ASTs and selected subtrees of the query and references, PDM determines whether the query is structurally closer to Human or AI references by comparing their distributions over shared structural prototypes.

Structural encoding. For each complete AST or selected subtree, we use the PDM graph encoder to obtain node representations:

$$\mathbf{H} = \mathrm{GE}(G, \mathbf{X}; \Theta_P), \tag{11}$$

where Θ_P denotes the PDM encoder parameters. We then aggregate $\mathbf{H}$ through a pooling operation to obtain a graph-level structural representation $\mathbf{z}$. The encoder is shared by the query and references, while its parameters are separate from the DSS encoder.

Shared structural prototypes. We collect the representations of the complete ASTs and selected subtrees from both Human and AI references into $\mathbf{Z}_{\mathrm{ref}}$. We then construct M shared structural prototypes through PMA (Lee et al., 2019):

$$\mathbf{U} = \mathrm{PMA}(\mathbf{Z}_{\mathrm{ref}}), \tag{12}$$

where $\mathbf{U} = \{\mathbf{u}_1, \ldots, \mathbf{u}_M\}$ denotes the resulting prototypes. The prototypes are shared by the two reference classes and are constructed without using the query.

For each structural representation $\mathbf{z}$, we map it to a distribution over the shared prototypes:

$$p(m) = \frac{\exp(\mathbf{z}^\top \mathbf{u}_m / \tau)}{\sum_{n=1}^{M} \exp(\mathbf{z}^\top \mathbf{u}_n / \tau)}, \tag{13}$$

where $\mathbf{z}$ and $\mathbf{u}_m$ are ℓ_2-normalized, such that $\mathbf{z}^\top \mathbf{u}_m$ measures their cosine similarity, and τ is a temperature parameter. Thus, each complete AST or selected subtree is represented by a distribution over the same M structural prototypes.

Reference distributions. Using the subtree weights α_j defined in Eq. 10, we aggregate the prototype distributions of Human and AI references separately to obtain the class distributions P_H and P_A. Each reference contributes both its complete-AST distribution and its selected-subtree distributions. The detailed weighting, aggregation, and smoothing procedures are provided in Appendix B.2.

Distribution matching. Let p_0 denote the prototype distribution of the complete query AST, and let p_k denote that of its k-th selected subtree for $k = 1, \ldots, K$. We compare how well each query distribution matches the two class distributions:

$$e_k = \sum_{m=1}^{M} p_k(m) \log \frac{P_A(m)}{P_H(m)}, \qquad k = 0, \ldots, K. \tag{14}$$

This score equals the KL divergence from p_k to P_H minus the KL divergence from p_k to P_A. A positive e_k therefore indicates that p_k is closer to the AI reference distribution in KL divergence, while a negative value indicates that it is closer to the Human reference distribution.

We average the evidence from the complete query AST and its selected subtrees:

$$e = \frac{1}{K+1} \sum_{k=0}^{K} e_k. \tag{15}$$

The final AI probability is

$$\hat{p} = \sigma(ae), \tag{16}$$

where σ is the sigmoid function and $a = \mathrm{softplus}(s) > 0$ is a scaling factor parameterized by a learnable scalar s. The query is classified as AI when $\hat{p} \geq 0.5$ and Human otherwise.

4.4 MODEL TRAINING

For each labeled training program, let $y \in \{0, 1\}$ denote its ground-truth label, where $y = 1$ indicates AI and $y = 0$ indicates Human. Given the predicted AI probability $\hat{p}$, we use the binary cross-entropy loss

$$\mathcal{L}_{\mathrm{cls}} = -y \log \hat{p} - (1 - y) \log(1 - \hat{p}). \tag{17}$$

To discourage DSS from selecting unnecessarily large subtrees, we introduce a subtree-size penalty using the subtree weights defined in Eq. 10:

$$\mathcal{L}_{\text{size}} = \text{mean}\left[\sum_{j=1}^{K} \alpha_j |C_j|\right]. \tag{18}$$

where $|C_j|$ denotes the number of nodes in subtree C_j.

The final objective is

$$\mathcal{L} = \mathcal{L}_{\text{cls}} + \lambda_{\text{size}} \mathcal{L}_{\text{size}}, \tag{19}$$

where λ_{size} controls the strength of the subtree-size penalty. We jointly optimize DSS and PDM with this objective in a single training stage. To train the DSS scoring network, we use a straight-through estimator (STE) over feasible candidate subtrees, preserving discrete selections in the forward pass while using soft candidate weights for backpropagation. Details are provided in Appendix B.3.

5 Experiments

5.1 Experimental Setup

Datasets. We construct two benchmarks for AI-generated code detection. CodeNet100 is derived from IBM Project CodeNet (Puri et al., 2021) and contains 4,200 Python solutions across 100 programming problems. C1 is constructed from a single problem in a large-scale programming contest and contains 900 solutions written in C, C++, Java, and Python. Both datasets contain human-written submissions and LLM-generated solutions. To further increase the detection difficulty, we additionally include polished versions of human submissions and AI solutions generated in a human-like coding style. Labels follow the original source of each solution: polished human submissions remain labeled Human, while human-like AI solutions remain labeled AI. Detailed construction procedures are provided in Appendix C.

Competitors. All competitors are from the following three categories. (1) *Representation-based detectors*: GPTSniffer (Nguyen et al., 2024), CodeGPTSensor (Xu et al., 2025), Suh AST Embedding (Suh et al., 2025), DroidDetect (Orel et al., 2025), and DeTeCtive (Guo et al., 2024) learn representations for distinguishing human-written and AI-generated content. (2) *Probability- and perturbation-based detectors*: Fast-DetectGPT (Bao et al., 2024), DetectCodeGPT (Shi et al., 2025), and Dual-CodeDetect (Li et al., 2026) detect AI-generated content using token probabilities or changes in model scores after perturbation. Further descriptions are provided in Appendix D. (3) *Ours*: **AST-TED** and **AIDAR** are our proposed methods. AST-TED directly compares program structures using tree edit distance. AIDAR combines Discriminative Subtree Selection (DSS) with Prototype Distribution Matching (PDM) to compare global and local program structures.

Task and Evaluation. Task and Evaluation. For each programming problem, we split the solutions into a reference (training and validation) set and a query (testing) set. The task is to classify each query program as Human or AI. All methods are evaluated on the same query set. For reference-based methods, the reference set provides labeled Human and AI programs for comparison, and each query is compared only with references from the same programming problem. The reference set is also used for training and validation by learning-based methods. CodeNet100 contains 2,140 references and 2,060 queries, while C1 contains 600 references and 300 queries. Both sets are balanced between Human and AI programs. We report Accuracy, Human Recall, and AI Recall. Human Recall is the accuracy on human-written programs, while AI Recall is the accuracy on AI-generated programs.

5.2 Implementation Details

Baseline settings. We follow the settings recommended in the original papers or official implementations of all baselines. Detailed configurations are provided in Appendix E.

AIDAR settings. We use two separately parameterized 3-step GGNN encoders for DSS and PDM, each with a hidden dimension of 128. Unless otherwise stated, DSS uses sequential greedy selection

Table 1: Performance (%) on CodeNet100 and C1. AST-TED is our non-neural structural baseline.

Category	Method	CodeNet100			C1		
		Accuracy	Human Recall	AI Recall	Accuracy	Human Recall	AI Recall
Representation	GPTSniffer	62.04	74.85	49.22	<u>97.67</u>	<u>95.33</u>	**100.00**
	CodeGPTSensor	<u>92.96</u>	<u>92.23</u>	93.69	92.33	89.33	95.33
	Suh AST Embedding	87.23	83.40	91.07	86.33	77.33	95.33
	DroidDetect	89.56	83.79	<u>95.34</u>	92.00	86.67	97.33
	DeTeCtive	84.71	75.92	93.50	94.67	90.67	98.67
Prob./Perturb.	Fast-DetectGPT	66.07	83.69	48.45	60.33	36.00	84.67
	DetectCodeGPT	55.19	26.02	84.37	72.67	60.67	84.67
	DualCodeDetect	51.07	4.76	**97.38**	52.67	16.67	88.67
Ours	AST-TED	89.08	84.47	93.69	87.67	76.67	98.67
	AIDAR	**95.05**	**96.60**	93.50	**99.00**	**98.67**	<u>99.33</u>

Best results in each column are shown in **bold**, and second-best results are <u>underlined</u>.

with $K = 2$ on both CodeNet100 and C1. Each query is therefore represented by its complete AST and two selected, node-disjoint subtrees. Within each selection round, dynamic programming finds the highest-scoring feasible subtree among the remaining nodes. PDM constructs eight shared structural prototypes using PMA (Lee et al., 2019). We train AIDAR with $\lambda_{\text{size}} = 0.01$ unless otherwise stated. Further implementation details are provided in Appendix E.

5.3 Performance Evaluation

Table 1 presents the results on CodeNet100 and C1. First, AIDAR achieves the highest accuracy on both datasets, indicating our structure-based detection framework that combines discriminative subtree selection with prototype-based distribution matching. We further examine the contributions of these components in Sect. 5.4. Second, AIDAR achieves the highest Human Recall while maintaining high AI Recall, indicating that its overall performance does not rely on favoring one class. This demonstrates the effectiveness of reference-based structural comparison in distinguishing both human-written and AI-generated programs. Lastly, AIDAR consistently outperforms both structure-aware baselines. Suh AST Embedding incorporates AST information but still relies on a serialized representation processed by a pretrained code encoder, without explicitly modeling structural relations. Our AST-TED baseline directly compares AST structures and achieves accuracies of 89.08% and 87.67% without learned detector parameters. It therefore provides an explicit, non-neural structural baseline. AIDAR improves on both approaches by learning to select local structures and compare global and local evidence through shared prototype distributions. The ablation study provides a more direct assessment of these design choices.

5.4 Ablation Study

Variant 1 removes both DSS and PDM and directly applies Normalized AST-TED to the complete AST. Variant 2 removes DSS and applies PDM using only the complete AST. Variant 3 retains DSS but replaces PDM with Normalized TED over the selected structural regions.

Table 2 yields three observations. First, Variant 2 outperforms Variant 1 in accuracy on both datasets, improving from 89.08% to 94.51% on CodeNet100 and from 87.67% to 98.33% on C1. This supports the effectiveness of learned prototype-based comparison over direct complete-AST edit-distance matching in the evaluated settings.

Second, DSS also benefits TED-based comparison. Variant 3 achieves accuracies of 90.73% and 92.33%, exceeding Variant 1 on both datasets. Thus, selected local structures provide useful information even without PDM, but their inclusion does not improve both class-wise recalls uniformly.

Finally, the full AIDAR achieves the highest accuracy among the variants, reaching 95.05% on CodeNet100 and 99.00% on C1. Compared with Variant 2, it provides further gains of 0.54 and 0.67 percentage points, respectively. Overall, PDM alone provides a larger accuracy improvement over AST-TED than DSS alone, while combining both components yields the best accuracy in the reported configurations.

Table 2: Ablation study of DSS and PDM (%).

Variant	DSS	PDM	CodeNet100			C1		
			Accuracy	Human Recall	AI Recall	Accuracy	Human Recall	AI Recall
Variant 1	×	×	89.08	84.47	<u>93.69</u>	87.67	76.67	<u>98.67</u>
Variant 2	×	✓	<u>94.51</u>	<u>94.95</u>	**94.08**	<u>98.33</u>	<u>98.00</u>	<u>98.67</u>
Variant 3	✓	×	90.73	90.19	91.26	92.33	86.67	98.00
AIDAR	✓	✓	**95.05**	**96.60**	93.50	**99.00**	**98.67**	**99.33**

DSS: Discriminative Subtree Selection. PDM: Prototype Distribution Matching. Variants without PDM use normalized TED matching.

5.5 Real-World Deployment on C2

Deployment setting. We deployed AIDAR during a programming contest with 4,316 participants. C2 contains 3,950 anonymized submissions to Question 2 E.1. The reference pool contains 430 generated AI programs and 430 Human references as detailed in Appendix C. We screen 1,176 full-score submissions using AIDAR, AST-TED and DeTeCtive. A submission is reported for human review only when all three methods classify it as AI.

This conservative design reflects the constraints of real-world detection. Human and AI code differ statistically, but their styles can overlap. For example, if only 1% of 10,000 candidates independently write code resembling AI output, this still amounts to 100 candidates. AI-like code therefore cannot establish misconduct: final determinations require contest-site checks and other physical evidence, not model predictions alone. Because such verification is costly, the practical bottleneck is producing a small set of credible leads. Strong benchmark performance alone is insufficient if a detector produces more alerts than humans can reasonably investigate.

The three methods combine semantic and structural information: DeTeCtive provides text-based representations, while AST-TED and AIDAR provide direct and learned structural comparisons. Requiring agreement across these perspectives restricts reports to submissions that appear AI-like in multiple ways, keeping the manual-review workload manageable.

Results. The joint screening procedure reports 17 of the 1,176 submissions (1.45%) for manual review, limiting immediate inspection to a small candidate set. Notably, the eight highest-ranked submissions were exactly the eight cases flagged by human experts as suspected AI-generated code. These results demonstrate the practical effectiveness of our method.

5.6 Additional Experiments

We provide additional experiments and analyses in the appendix. First, we present two intuitive case studies to illustrate the interpretability of AIDAR. The DSS case study shows which local AST structures are selected and how they correspond to concrete code fragments (Appendix A.1), while the PDM case study shows how a learned structural prototype relates the query to Human and AI references and contributes to the final prediction (Appendix A.2). Second, we examine the effects of the number of selected subtrees K and the subtree-size penalty λ_{size} to better understand the sensitivity of AIDAR to these hyperparameters (Appendix A.3).

6 Conclusions

We presented an AST-based framework for AI-generated code detection, comprising the non-neural AST-TED baseline and AIDAR, which combines Discriminative Subtree Selection with Prototype Distribution Matching. AIDAR achieves 95.05% and 99.00% accuracy on CodeNet100 and C1, outperforming all evaluated baselines. Ablations support both components, while case studies connect predictions to concrete code patterns. In C2, AIDAR, AST-TED, and DeTeCtive jointly provided leads for human review. Detection results assist investigation but do not establish misconduct.

AI USE STATEMENT

Generative AI tools were used to improve the clarity and language of the manuscript and to assist with code writing and editing. They were also used to generate some of the code samples included in the experimental datasets, as described in the dataset construction section. All research ideas, hypotheses, methodological choices, experimental designs, and interpretations of the results were developed by the authors. All AI-assisted manuscript revisions and code were reviewed and verified by the authors. The authors take full responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

We provide detailed descriptions of the proposed method and training procedure in Appendix B, dataset construction and data splits in Appendix C, baseline methods in Appendix D, and implementation and hyperparameter settings in Appendix E. We also provide anonymous source code with the submission to facilitate reproduction of the reported results.

REFERENCES

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Guangsheng Bao, Yanbin Zhao, Zhiyang Teng, Linyi Yang, and Yue Zhang. Fast-detectgpt: Efficient zero-shot detection of machine-generated text via conditional probability curvature. In *ICLR*, volume 2024, pp. 24814–24836, 2024.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of machine learning research*, 24(240):1–113, 2023.

Werner Dinkelbach. On nonlinear fractional programming. *Management science*, 13(7):492–498, 1967.

Sebastian Gehrmann, Hendrik Strobelt, and Alexander Rush. GLTR: Statistical detection and visualization of generated text. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pp. 111–116, 2019. doi: 10.18653/v1/P19-3019.

Xun Guo, Shan Zhang, Yongxin He, Ting Zhang, Wanquan Feng, Haibin Huang, and Chongyang Ma. Detective: Detecting ai-generated text via multi-level contrastive learning. *NeurIPS*, 37: 88320–88347, 2024.

John Kirchenbauer, Jonas Geiping, Yuxin Wen, Jonathan Katz, Ian Miers, and Tom Goldstein. A watermark for large language models. In *Proceedings of the 40th International Conference on Machine Learning*, pp. 17061–17084, 2023.

Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International conference on machine learning*, pp. 3744–3753. PMLR, 2019.

Taehyun Lee, Seokhee Hong, Jaewoo Ahn, Ilgee Hong, Hwaran Lee, Sangdoo Yun, Jamin Shin, and Gunhee Kim. Who wrote this code? watermarking for code generation. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pp. 4890–4911, 2024. doi: 10.18653/v1/2024.acl-long.268.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Zhengdao Li, Xiuwei Shang, Zhenkan Fu, Shikai Guo, Weiming Zhang, NengHai Yu, and Kejiang Chen. Dualcodedetect: Zero-shot llm-generated code detection via dual-channel perturbation. *Proceedings of the ACM on Software Engineering*, 3(FSE):3604–3626, 2026.

Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D. Manning, and Chelsea Finn. DetectGPT: Zero-shot machine-generated text detection using probability curvature. In *Proceedings of the 40th International Conference on Machine Learning*, pp. 24950–24962, 2023.

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

Daniil Orel, Indraneil Paul, Iryna Gurevych, and Preslav Nakov. Droid: A resource suite for ai-generated code detection. In *EMNLP*, pp. 31251–31277, 2025.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*, 2021.

Vinu Sankar Sadasivan, Aounon Kumar, Sriram Balasubramanian, Wenxiao Wang, and Soheil Feizi. Can AI-generated text be reliably detected? stress testing AI text detectors under various attacks. *Transactions on Machine Learning Research*, 2025. URL https://mlanthology.org/tmlr/2025/sadasivan2025tmlr-aigenerated/.

Yuling Shi, Hongyu Zhang, Chengcheng Wan, and Xiaodong Gu. Between lines of code: Unraveling the distinct patterns of machine and human programmers. In *ICSE*, pp. 1628–1639. IEEE, 2025.

Hyunjae Suh, Mahan Tafreshipour, Jiawei Li, Adithya Bhattiprolu, and Iftekhar Ahmed. An empirical study on automatically detecting ai-generated source code: How far are we? In *ICSE*, pp. 859–871. IEEE, 2025. doi: 10.1109/ICSE55347.2025.00064.

Adaku Uchendu, Thai Le, Kai Shu, and Dongwon Lee. Authorship attribution for neural text generation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, pp. 8384–8395, 2020. doi: 10.18653/v1/2020.emnlp-main.673.

Xiaodan Xu, Chao Ni, Xinrong Guo, Shaoxuan Liu, Xiaoya Wang, Kui Liu, and Xiaohu Yang. Distinguishing llm-generated from human-written code by contrastive learning. *ACM Transactions on Software Engineering and Methodology*, 34(4):1–31, 2025.

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*, volume 38, pp. 23155–23162, 2024. doi: 10.1609/aaai.v38i21.30361.

Xingyu Yao, Zhendong Mao, and Quan Wang. CodeRipple: Wavelet-based detection of LLM-generated code. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics*, pp. 38351–38364, 2026. doi: 10.18653/v1/2026.acl-long.1777.

A Additional Experiments

A.1 Case Study of DSS

Figure 2 compares the two subtrees selected for the same query with $K = 2$ under $\lambda_{\text{size}} = 0.01$ and maximum-density selection. Both settings select structures from the same respective AST branches and identify the same underlying computations in the same order. The first corresponds to the midpoint calculation `left + (right - left) / 2`, and the second corresponds to the arithmetic expression `(apples * k + 99) / 100`. Thus, despite using different size-control objectives, the two settings focus on the same local code fragments.

The difference lies in how much surrounding structure is retained. Our method selects the arithmetic expressions themselves, whereas maximum-density selection extends outward to include the enclosing variable definitions, `long long mid = ...;` and `long long spoiled = ...;`. The additional structure therefore comes from the outer declarations rather than a different computation. This comparison gives the selected subtrees a direct source-level interpretation: our method isolates the computational expressions, while maximum-density selection also retains their declaration context. A reader can trace each selected subtree to a meaningful code fragment and inspect which surrounding structure is included or omitted.

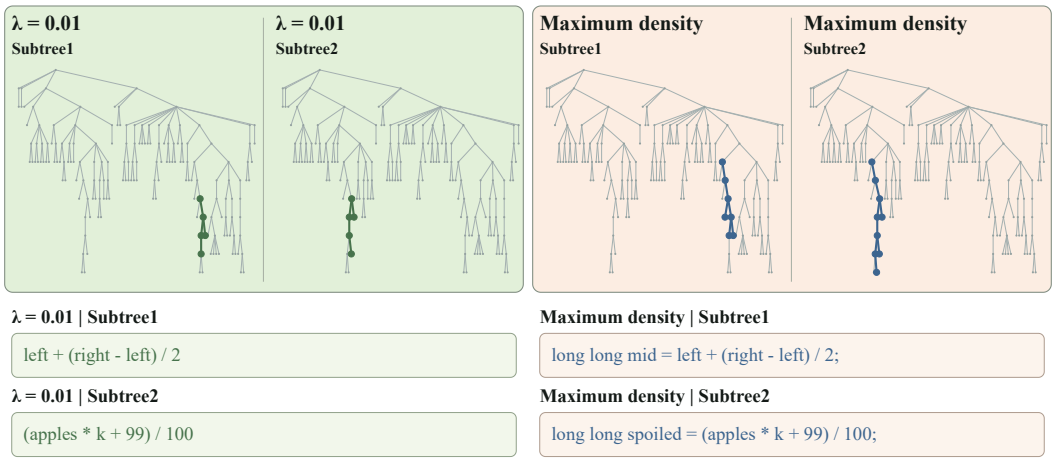


Figure 2: Case study of DSS

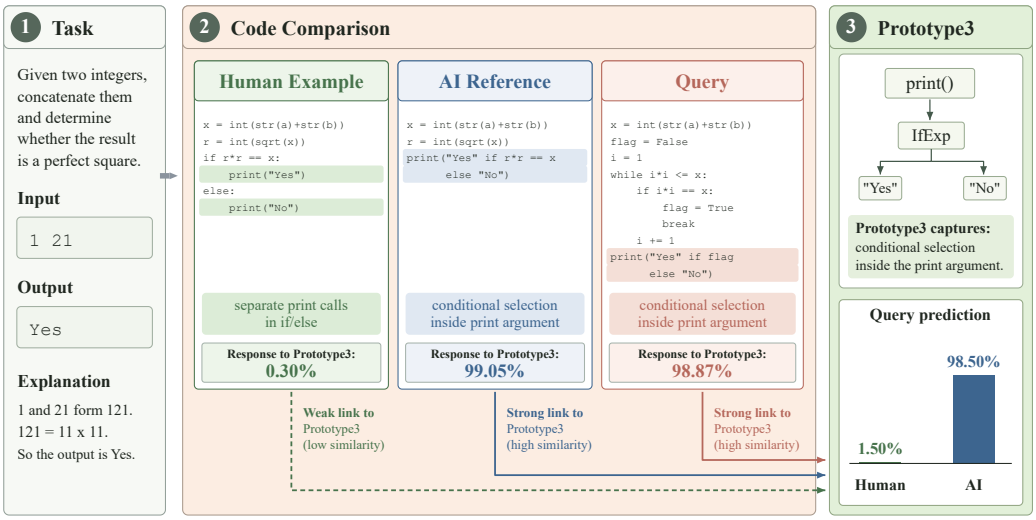


Figure 3: Case study of PDM

A.2 Case Study of Prototype

Figure 3 illustrates how a learned prototype provides an interpretable structural basis for the model's prediction. The task is to concatenate two integers and determine whether the result is a perfect square. PDM represents query and reference structures as distributions over shared prototypes constructed from the references. In this example, Prototype 3 is associated with a concrete coding pattern: a conditional expression placed inside the argument of `print` to select between `"Yes"` and `"No"`.

The AI reference uses this pattern, whereas the Human example places separate `print` calls in thebranches of an `if/else` statement. Their responses to Prototype 3 are 99.05% and 0.30%, respectively. The query also places a conditional expression inside the `print` argument and has a similarly strong response of 98.87%. Although the query checks the square condition using a loop rather than the square-root computation used by the AI reference, their local output structures match. Prototype 3 therefore captures a recognizable structural pattern shared by these two programs despite differences in their surrounding computations.

Table 3: Effect of the number of selected subtrees K (%).

Selection	K	CodeNet100			C1		
		Accuracy	Human Recall	AI Recall	Accuracy	Human Recall	AI Recall
Sequential greedy	0	94.51	94.95	**94.08**	98.33	98.00	98.67
	1	94.61	95.83	93.40	**99.33**	**98.67**	**100.00**
	2	**95.05**	**96.60**	93.50	99.00	**98.67**	99.33
	3	94.81	95.53	**94.08**	97.33	97.33	97.33
	4	93.20	96.12	90.29	99.00	**98.67**	99.33
	5	94.08	94.17	93.98	97.00	95.33	98.67
Joint DP	0	94.51	94.95	**94.08**	98.33	98.00	**98.67**
	1	94.32	95.15	93.50	97.67	**98.67**	96.67
	2	94.81	95.92	93.69	**98.67**	**98.67**	**98.67**
	3	**94.95**	**96.12**	93.79	97.00	97.33	96.67
	4	93.45	95.15	91.75	98.00	97.33	**98.67**
	5	93.45	93.50	93.40	98.33	**98.67**	98.00

For prediction, PDM compares each query component's prototype distribution with the aggregated Human and AI reference distributions, combining evidence from the complete AST and its two selected subtrees. The full model assigns probabilities of 98.50% to AI and 1.50% to Human, classifying the query as AI. The strong response to Prototype 3 provides an interpretable local cue consistent with this prediction: the query shares the conditional-print structure of the AI reference rather than the separate-print structure of the Human example. Importantly, the 98.87% prototype response is an intermediate structural association, while the 98.50% AI probability is the output of the full model. This case connects an otherwise abstract prototype vector to a concrete coding pattern, making the structural basis of the prediction understandable without treating that pattern as a standalone authorship rule.

A.3 Hyperparameter Analysis

Impact of the number of selected subtrees. We vary K from 0 to 5 and compare sequential greedy selection with joint dynamic programming (joint DP). Sequential greedy selection chooses subtrees one at a time, excluding previously selected nodes, while joint DP jointly selects K non-overlapping subtrees to maximize their total DSS score. Each configuration with $K \geq 1$ is trained separately, while $K = 0$ uses only the complete AST. We selected $K = 2$ using the training data, and use the query set here only to examine sensitivity to K.

Table 3 yields three observations. First, sequential greedy selection offers greater stability as K increases: under fixed structural scores, it preserves earlier selections, whereas joint DP may revise the selected set. Although greedy selection does not guarantee the globally optimal total subtree score, it achieves detection performance comparable to joint DP. These results suggest that global joint optimization is not necessary for effective detection in our experiments, supporting our choice of sequential greedy selection. Second, selecting a small number of subtrees can improve detection, but increasing K does not consistently yield further gains. This indicates that local structures provide useful information, while including more subtrees is not necessarily beneficial because these will bring more noise. Lastly, no single K performs best across both datasets. For example, sequential greedy selection achieves its highest accuracy at $K = 2$ on CodeNet100 and $K = 1$ on C1. Automatically determining an appropriate number of subtrees for each programming problem remains an interesting direction for future work.

Impact of the subtree-size penalty. We vary the subtree-size penalty coefficient λ, denoted λ_{size} in the training objective, over $\{0, 0.001, 0.01, 0.1, 1\}$. For each value of λ, we retrain the full AIDAR model using the corresponding training objective. We selected $\lambda_{\text{size}} = 0.01$ using the training data, and use the query set here only to examine sensitivity to the size penalty. We also compare with

Table 4: Detection performance and subtree size under different size-control settings.

Setting	Accuracy	Human Recall	AI Recall	Mean query subtree size
$\lambda = 0$	97.33	97.33	97.33	14.92
$\lambda = 0.001$	<u>98.33</u>	97.33	<u>99.33</u>	10.13
$\lambda = 0.01$	**99.00**	**98.67**	<u>99.33</u>	5.60
$\lambda = 0.1$	97.33	97.33	97.33	4.08
$\lambda = 1$	97.33	97.33	97.33	4.01
Maximum density	**99.00**	<u>98.00</u>	**100.00**	8.61

maximum-density selection, which maximizes the structural score per selected node:

$$C^\star \in \arg\max_{C \in \mathcal{F}(T)} \frac{S(C)}{|C|}, \tag{20}$$

where $\mathcal{F}(T)$ contains the currently feasible connected subtrees with 4–24 nodes, $S(C)$ is the DSS structural score, and $|C|$ is the number of nodes. We solve each single-subtree problem using the classical algorithm (Dinkelbach, 1967). Multiple subtrees are selected sequentially, excluding previously selected nodes.

Table 4 yields three observations. First, increasing λ consistently reduces the mean subtree size, from 14.92 nodes at $\lambda = 0$ to approximately four nodes at $\lambda = 1$. This confirms that the penalty controls the size of the selected structures. Second, detection accuracy peaks at $\lambda = 0.01$ and decreases when a larger penalty is applied. A moderate penalty therefore provides a better balance between subtree compactness and detection performance in this experiment. Lastly, maximum-density selection adaptively determines the score–size trade-off and matches the best accuracy achieved with a fixed λ (99.00%). However, its iterative implementation requires repeated DP solves for each subtree, whereas our selector requires only one. Our approach therefore achieves the same accuracy without this iterative selection overhead.

B Further More Details of Proposed Approach

B.1 Additional Details of Subtree Selection

DSS uses sequential greedy selection, with a single-subtree dynamic program at each round. Nodes selected in earlier rounds are excluded from subsequent selections.

For each available node v, the dynamic program stores the largest sum of retained connection scores for a connected subtree rooted at v, for each size up to the upper size bound. The one-node state contains only v and has a connection score of zero; all other states are initially infeasible. Children are processed one at a time. For an available child u, the dynamic program either omits its branch or attaches a connected subtree rooted at u, adding the connection score a_{vu} and the retained connection scores of that subtree. Only combinations within the upper size bound are retained. Connections through previously selected nodes are not allowed.

After processing all children, we add the root score r_v to obtain the structural score $S(C)$ of each candidate rooted at v. We compare candidates across all available roots and all sizes satisfying the lower and upper bounds. The highest-scoring feasible subtree is recovered by backtracking through the recorded choices.

We then exclude its nodes and repeat the single-subtree optimization for the next greedy round. After K rounds, the selected subtrees are pairwise node-disjoint. This procedure is applied separately to the query and each reference. The complete AST is retained as an additional component and is not subject to the node-disjointness constraint on the selected subtrees.

B.2 Additional Details of Prototype Distribution Matching

Reference distribution construction. For each reference, let p_0 denote the prototype distribution of its complete AST and p_j that of its j-th selected subtree, where $j = 1, \ldots, K$. We assign the base

weights

$$\mu_0 = \frac{1}{K+1}, \qquad \mu_j = \frac{K}{K+1}\alpha_j, \quad j = 1, \ldots, K. \tag{21}$$

These weights satisfy

$$\mu_0 + \sum_{j=1}^{K} \mu_j = 1. \tag{22}$$

The complete AST therefore contributes one share, while the remaining weight is distributed among the selected subtrees according to their DSS importance.

We further measure how confidently each component is assigned to the prototypes:

$$\gamma_j = 1 - \frac{H(p_j)}{2 \log M}, \qquad j = 0, \ldots, K. \tag{23}$$

where $H(\cdot)$ denotes entropy. The final aggregation weight is then $w_j = \mu_j \gamma_j$.

We then collect all components from references of the same class. For class $c \in \{H, A\}$, let p_i and w_i denote the prototype distribution and aggregation weight of its i-th reference component. The class distribution is

$$\bar{P}_c = \frac{\sum_i w_i p_i}{\sum_i w_i}. \tag{24}$$

We finally smooth the class distribution to avoid zero probabilities:

$$P_c = (1 - \epsilon)\bar{P}_c + \frac{\epsilon}{M}\mathbf{1}, \tag{25}$$

where ϵ is a small smoothing coefficient.

B.3 Training through discrete subtree selection

DP and the outer greedy procedure are discrete and are not differentiated. During training, detached copies of the DSS root and connection scores are passed to the selector. The forward pass therefore uses the actual subtree returned by DP and greedy selection. To train the scoring network, we use a straight-through estimator over a retained set of feasible candidate subtrees.

Candidate scores. At greedy round k, let $\mathcal{B}_k = \{C_{k,1}, \ldots, C_{k,B_k}\}$ denote the retained candidates. This set includes the selected subtree $C_k^\star$ and may include additional feasible alternatives. All candidates satisfy the selection constraints for that round, including exclusion of nodes selected in earlier rounds. Candidate identities and the selected index are treated as fixed during backpropagation.

For each candidate, we recompute its structural score using the original, non-detached DSS outputs:

$$s_{k,j} = r_{\mathrm{root}(C_{k,j})} + \sum_{(u,v) \in E(C_{k,j})} a_{uv}. \tag{26}$$

Thus, the candidate structures are obtained through discrete search, while their scores remain connected to the DSS scoring network.

We normalize these scores over the candidates retained in the current round:

$$\pi_{k,j} = \frac{\exp(s_{k,j}/\tau_{\mathrm{STE}})}{\sum_{\ell=1}^{B_k} \exp(s_{k,\ell}/\tau_{\mathrm{STE}})}, \tag{27}$$

where $\tau_{\mathrm{STE}} > 0$ controls the sharpness of the soft candidate weights.

Straight-through weights. Let $d_{k,j} = 1$ when $C_{k,j} = C_k^\star$ and $d_{k,j} = 0$ otherwise. We construct the straight-through weights as

$$\omega_{k,j} = d_{k,j} + \pi_{k,j} - \mathrm{sg}(\pi_{k,j}), \tag{28}$$

where $\mathrm{sg}(\cdot)$ is the stop-gradient operator: it preserves its input value in the forward pass and has zero derivative during backpropagation.

In the forward pass, the two soft terms cancel, giving $\omega_{k,j} = d_{k,j}$. The selected candidate therefore has weight one and all other candidates have weight zero. In the backward pass, the gradient through these weights follows the softmax:

$$\left.\frac{\partial \omega_{k,j}}{\partial s_{k,\ell}}\right|_{\text{backward}} = \frac{1}{\tau_{\text{STE}}} \pi_{k,j} \left(\mathbb{I}[j = \ell] - \pi_{k,\ell}\right). \tag{29}$$

This provides a surrogate gradient to the DSS scoring network through the candidate weights, without differentiating through search, backtracking, or the exclusion of previously selected nodes. The soft weights define the backward surrogate; they do not replace the discrete forward selection with a mixture of subtrees.

The candidate weights $\pi_{k,j}$ should be distinguished from the reference subtree weights α_j in Sect. 4.2. The former compare alternative candidates within one selection round, whereas the latter weight the subtrees already selected for a reference.

C Further Descriptions of Datasets

CodeNet100. We construct CodeNet100 from IBM Project CodeNet Puri et al. (2021). We select 100 Python programming problems, including 50 medium-easy and 50 medium-hard problems. For each problem, we collect accepted human submissions and generate AI solutions. The AI solutions are obtained through six generation sources: DeepSeek V4 Flash, DeepSeek V4 Pro, DeepSeek Web, Doubao Web, Qwen 3.7 Plus API, and Qwen 3.7 Plus Web. We additionally construct two types of samples to augment the dataset and increase the difficulty of detection. *Human-polish* samples are polished versions of human-written programs and therefore retain the Human label. *AI-humanlike* samples are generated by an LLM in a more human-like coding style and retain the AI label.

Thus, labels indicate the source of a program rather than its apparent coding style. The final reference set contains 1,070 Human and 1,070 AI programs, while the query set contains 1,030 Human and 1,030 AI programs. No source program is shared between the reference and query sets. Generated programs pass the recorded parsing and visible-test checks. These checks verify basic validity but do not imply acceptance by CodeNet's hidden online judge.

C1. C1 is constructed from anonymized submissions to a single problem in a large-scale programming contest with 5,778 participants.. We focus on Question 2 because its difficulty level and large number of submissions make it the most suitable problem for detecting potential unauthorized AI-assisted coding. The original Human corpus contains 5,303 submissions written in C, C++, Java, and Python. AI programs are generated through 14 model routes: Claude Opus 5, Claude Sonnet 5, Gemini 3.6 Flash, GPT-5.4 Mini, GPT-5.6 Luna, GPT-5.6 Terra, Grok 4.5, Grok 4.6, K3, GLM-5.2, Qwen 3.7 Plus, Doubao, DeepSeek V4 Flash, and DeepSeek V4 Pro.

Before adding the Human-polish and AI-humanlike samples, the reference set contains 200 Human and 200 AI programs and the query set contains 100 Human and 100 AI programs. We then add 100 Human-polish and 100 AI-humanlike programs to the reference set and 50 samples of each type to the query set. The final reference set therefore contains 300 Human and 300 AI programs, while the query set contains 150 Human and 150 AI programs.

C2. C2 is constructed from 3,950 anonymized submissions to Question 2 of a separate large programming contest with 4,316 participants. We construct the Human reference set through a preliminary round of consensus screening. First, randomly sampled contest submissions serve as provisional Human references for the initial screening pass. We then select 430 submissions from those

Table 5: Summary of the two detection benchmarks.

Dataset	# Problems	Human		AI		# References	# Queries	Languages
		Original	Polished	Original	Human-like			
CodeNet100	100	2,000	100	2,000	100	2,140	2,060	Python
C1	1	300	150	300	150	600	300	C/C++, Java, Python

classified as Human by all three methods AIDAR AST-TED, and DeTeCtive and use them as the final Human reference set. To maintain a balanced reference pool, we pair these 430 submissions with 430 AI-generated programs. The AI references comprise 280 API-generated programs from 14 model routes: GPT-5.6 Luna, GPT-5.6 Terra, Claude Opus 5, Claude Sonnet 5, DeepSeek V4 Flash, DeepSeek V4 Pro, Qwen 3.7 Max, Gemini 3.6 Flash, Grok 4.6, GLM-5.2, GLM-5.3 Flash, Kimi K3, Hunyuan Hy3, and Hunyuan Hy4 Preview. An additional 150 AI programs are collected from 11 web-platform and generation-mode categories provided by Doubao, DeepSeek, Qwen, Yuanbao, and Wenxin Yiyan.

After excluding the 430 Human references, the remaining 3,520 Question 2 submissions form the query pool. Each query is compared only with references for the same problem, with different programming languages sharing a common reference pool. Unlike C1, C2 serves as a real-world screening corpus rather than a balanced benchmark.

D Further Descriptions of Baselines

We provide additional descriptions of the baseline methods used in our experiments.

D.1 Representation-Based Detectors

- **GPTSniffer** Nguyen et al. (2024). GPTSniffer is a supervised AI-code detector built on CodeBERT, which fine-tunes pre-trained code representations to distinguish human-written and AI-generated programs. Its prediction is produced by a classification head trained on top of the CodeBERT representation.
- **CodeGPTSensor** Xu et al. (2025). CodeGPTSensor uses UniXcoder to encode source code and introduces contrastive learning to make representations of human-written and AI-generated programs more distinguishable. The learned representations are subsequently used by a supervised classifier for AI-code detection.
- **Suh AST Embedding** Suh et al. (2025). Suh et al. investigate whether structural information can improve AI-generated code detection by representing programs through their ASTs. In the AST-based setting, the AST is serialized and encoded with CodeT5+, and the resulting representation is used for downstream classification.
- **DroidDetect** Orel et al. (2025). DroidDetect is a learned detector designed specifically for distinguishing AI-generated from human-written source code. It uses a ModernBERT-based code encoder together with supervised and representation-learning objectives to obtain discriminative code representations for classification.
- **DeTeCtive** Guo et al. (2024). DeTeCtive learns representations using multi-level contrastive learning, encouraging human- and machine-generated contents to occupy distinguishable regions in the representation space. Instead of using a conventional classification head, it performs detection by retrieving labeled examples that are nearest to the input representation.

D.2 Structure-Based Methods

- **Normalized AST-TED** : Normalized AST-TED measures structural similarity using tree edit distance between program ASTs, normalized by the total number of nodes in the two trees. Each query is classified according to its nearest reference.

D.3 Probability- and Perturbation-Based Detectors

- **Fast-DetectGPT** Bao et al. (2024): Fast-DetectGPT is a zero-shot detector based on conditional probability curvature, exploiting differences between the token-probability patterns of human- and machine-generated content. It produces a detection score directly from a language model and therefore does not require detector-specific supervised training.
- **DetectCodeGPT** Shi et al. (2025): DetectCodeGPT detects AI-generated programs by applying small formatting perturbations, such as inserting spaces or newlines, while preserving the program semantics. It measures how the language-model score changes under

these perturbations, based on the observation that human-written and AI-generated code respond differently to such modifications.

- **DualCodeDetect** Li et al. (2026): DualCodeDetect extends perturbation-based detection by considering both lexical and structural changes to source code. It combines identifier-level perturbations with semantics-preserving structural transformations and uses the resulting score changes to distinguish AI-generated from human-written programs.

E Implementation Details

Baseline settings. We use the official implementations for open-source baselines whenever available and follow the settings recommended in their corresponding papers or released code.

For GPTSniffer Nguyen et al. (2024), we use our own reproduction following the original method, retaining its CodeBERT-base encoder and supervised classification head.

For CodeGPTSensor Xu et al. (2025), we use UniXcoder with a maximum input length of 400 tokens. The training batch size is 8, the learning rate is 2×10^{-5}, and the model is trained for up to 20 epochs.

For Suh AST Embedding Suh et al. (2025), we use the AST-based setting with the CodeT5+ 110M encoder and follow the recommended downstream classification settings in the original implementation.

For DroidDetect Orel et al. (2025), we retain its ModernBERT-based encoder and classification head and follow the recommended configuration of the released implementation.

For DeTeCtive Guo et al. (2024), we use its SimCSE-RoBERTa encoder with a maximum input length of 512 tokens. We set $K = 1$ for inference, such that each query is assigned the label of its nearest reference.

For Normalized AST-TED , we first parse each program into an AST. Given two ASTs T_i and T_j, we compute their normalized distance as

$$d_{\mathrm{NTED}}(T_i, T_j) = \frac{\mathrm{TED}(T_i, T_j)}{|T_i| + |T_j|},$$

where $|T_i|$ and $|T_j|$ denote the numbers of nodes in the two trees. For each query, we compute its normalized TED to all references from the same problem and assign the label of the nearest reference.

For Fast-DetectGPT Bao et al. (2024), we follow its original zero-shot detection setting and retain the conditional-probability-curvature score.

For DetectCodeGPT Shi et al. (2025), we generate 50 perturbed versions of each program. We set the space- and newline-selection ratios to 0.5, with $\lambda_{\mathrm{spaces}} = 3$ and $\lambda_{\mathrm{newlines}} = 2$.

For DualCodeDetect Li et al. (2026), we use the official implementation and retain its default identifier- and structure-perturbation settings.

AIDAR settings. We use two separately parameterized 3-step GGNN encoders for DSS and PDM, each with 128-dimensional hidden states and shared parameters between queries and references. The scoring, projection, and PMA feed-forward modules use 128-dimensional hidden representations, while the PMA residual feed-forward layer expands to 256 dimensions. DSS uses sequential greedy selection to extract $K = 2$ node-disjoint subtrees from the query and from every Human or AI reference. Each selected subtree satisfies the same 4–24-node size constraint. For every program, the complete AST is retained together with the two selected subtrees. Reference subtree selections remain fixed during inference. The hyperparameters K and λ_{size} were selected exclusively on the training set, while the query set was reserved for final evaluation and was not used for hyperparameter selection. PDM uses 128-dimensional graph representations and an 8-representative PMA with four attention heads of dimension 32. We set the prototype-assignment temperature and reference-distribution smoothing coefficient to 0.1 and 0.02, respectively. We jointly optimize DSS and PDM in a single training stage using AdamW with a learning rate of 3×10^{-4} and weight decay 10^{-4}

for 3 epochs. The training objective is $\mathcal{L} = \mathcal{L}_{\text{cls}} + \lambda_{\text{size}}\mathcal{L}_{\text{size}}$, where the classification loss has unit weight and $\lambda_{\text{size}} = 0.01$ unless otherwise stated. The final classification threshold is fixed at 0.5.

E.1 Problem Statement for C2

Process Planning. There are n optional manufacturing stages. The duration of stage i is an independent normally distributed random variable $T_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$, where μ_i and σ_i are its mean and standard deviation. For a selected subset S, the total duration has mean and standard deviation

$$\mu_S = \sum_{i \in S} \mu_i, \qquad \sigma_S = \sqrt{\sum_{i \in S} \sigma_i^2}.$$

The benefit of selecting S is defined as μ_S.

Given an allowable delay m relative to the mean duration and a reliability threshold q, every nonempty selected subset must satisfy

$$\frac{m}{\sigma_S} \geq q.$$

This requires the total duration to be at most $\mu_S + m$ with probability at least $\Phi(q)$, where Φ is the standard normal cumulative distribution function. The parameter q is provided directly, so no evaluation of Φ is required. The empty subset is also feasible and has benefit zero. The task is to output the maximum achievable benefit as an integer.

Input and constraints. The input gives n, m, and q, followed by the stage means $\mu_1, \ldots, \mu_n$ and standard deviations $\sigma_1, \ldots, \sigma_n$. The integer parameters satisfy $1 \leq n \leq 100$, $1 \leq m \leq 10$, and $1 \leq \mu_i \leq 100$. The real-valued parameters satisfy $0.5 \leq q \leq 1.0$ and $0.1 \leq \sigma_i \leq 3.0$. The input specifies q to exactly three decimal places and each σ_i to exactly one decimal place.