Detecting Machine-Generated Code with Multiple Programming Languages, Generators, and Application Scenarios

First Author
Almotaz Al Khasawneh
202420032
motazkhasawneh@gmail.com

Second Author
Abdul Omer Farooq
202420120
abdulomer662@gmail.com

Abstract

Large Language Models (LLMs) have become increasingly capable of producing source code that closely resembles human-written programs. This similarity raises concern in authorship verification, academic integrity, and software security. SemEval Task X focuses on determining the origin of code snippets—whether they are written by humans (Subtask A) or generated by a specific LLM family (Subtask B). To address this challenge, we draw insights from recent multi model datasets such as MultiAIGCD, which emphasize the importance of diverse programming languages, prompts, and usage scenarios. Building on these observations, we develop a CodeBert-based system fine-tuned separately for each subtask. Our pipeline includes targeted preprocessing steps such as whitespace normalization, comment cleaning, and class-imbalance correction. The novelty of work lies in combining CodeBert with multi-stage Preprocessing strategy tailored specifically for machine-generated code detection, as well as utilizing generator metadata to stabilize training for multi-class attribution. These contributions help the model better capture subtle stylistic cues and structural patterns indicative of different LLM families. Experimental results show that CodeBert effectively differentiates human-written code from machine-generated code and can recognize patterns associated with specific LLM families. Ablation and error analyses highlight the critical role of preprocessing choices and dataset diversity, while also revealing challenges in distinguishing between LLMs with overlapping behaviors. Overall, our findings show that code-oriented transformer models, when combined with multi-scenario insights and specialized preprocessing, form a strong foundation for reliable code provenance detection.

1 Introduction

Large Language Models have become very good at generating high-quality source code in many programming languages, while this help developers by speeding up coding tasks, it also makes it harder to tell whether a piece of code was written by a human or produced by an AI system. This can cause issues for authorship checking, academic integrity, and identifying risky or auto-generated code. SemEval task X focuses on solving this problem. Subtask A asks whether a code is human-written or machine generated. Subtask B goes further and asks which LLM family produced the code, choosing from ten popular model families such as DeepSeek-AI, Qwen, Meta-LLaMA, Mistral, OpenAI, and others. This is difficult because many LLMs share similar training data and often produce code that follows the same style and structure. To tackle this challenges, we use CodeBert a transformer model trained on both natural language and programming language.It can capture small stylistic details—like how variables are named, how code is indented, and how comments are written—that may reveal whether the code is human or generated by a specific AI model. We fine-tune CodeBERT separately for each subtask and apply preprocessing steps to clean and standardize the input code. In this report, we describe the dataset, our model setup, experiments and results. We also include ablation studies showing which components help the most, and an error analysis that explains where the model struggles, especially with LLMs that behave similarly.

2 Literature Review

Research on detecting AI-generated code has expanded rapidly as modern large language models demonstrate increasingly advanced programming capabilities.Early work primarily examined general-purpose code representations, while more recent studies focus on distinguishing human-written code from machine-generated outputs and, in some cases, attributing code to a specific model family. A variety of datasets, detection architectures, and evaluation frameworks have been intro-

duced, aiming to improve detection reliability, interpretability, and cross-language generalization.

2.1 EX-CODE: Explainable Detection of AI-Generated Code

The EX-CODE framework proposes an interpretable approach to identifying AI-generated code. Rather than depending solely on deep neural models, it integrates traditional code-quality metrics with features extracted from language models. This hybrid design highlights stylistic indicators such as repetitive structures, highly uniform formatting, or regularity in naming patterns—signals that commonly appear in machine-generated code. A central contribution of EX-CODE is its emphasis on transparency: the system provides explanations for its decisions, making it suitable for educational environments, auditing workflows, and scenarios where explainability is essential.

2.2 MultiAIGCD: A Comprehensive Dataset for AI-Generated Code Detection

The MultiAIGCD dataset offers a large-scale resource specifically constructed for studying AI-generated code detection. It spans multiple programming languages, including Python, Java, and Go, and incorporates outputs from a diverse set of language model families under varied prompts and task conditions. The authors show that the characteristics of generated code differ significantly depending on the model, instruction, and coding objective. Their analysis also demonstrates that attributing code to a particular model family is more challenging than simply determining whether it is AI-generated. Due to its diversity and size, MultiAIGCD has emerged as a valuable benchmark for training and evaluating robust detection methods.

2.3 MAGECODE: Machine-Generated Code Detection

The MAGECODE Framework, Introduced by Pham et al., combines semantic representation from a pre-trained code model (CodeT5+) with statistical metrics such as log-likelihood, token rank, and entropy to detect machine-generated code. The authors construct a dataset of more than 45,000 snippets produced by several modern language models across Python, Java, and C++. Thier Experiment report high detection accuracy, with false positive rates kept very low. Ablation studies further demostrate that combining embedding-based features with statistical indicators yields superior performance compared to using either feature type independently.

2.4 CodeBERT: A Pre-Trained Model for Programming and Natural Languages

CodeBert is a tranformer-based model pre-trained on a combination of natural language description and source code. its training strategy enables it to learn syntactic structures, semantic relationships, and token-level patterns across multiple programming languages. CodeBERT has shown strong performance on downstream tasks such as code retrieval , classcification , and generation. These capabilities make it a suitable foundation model for detecting AI-generated code, as it can capture fine-grained stylistic and structural characteristics that differ between human-authored and machine-produced programs.

2.5 Distinguishing AI-Generated vs Human-Written Code for Plagiarism Prevention

The paper examines the problem of determining whether a code fragment was written by a human or produced by an AI model. The authors first assess several existing AI-text detection tools and show that they perform poorly on programming languages, demonstrating weak adaptability and inconsistent predictions. To overcome these shortcomings, the work proposes a hybrid detection approach that integrates static code characteristics with embedding-based features that capture code structure and style. Their best-performing system reports an F1-score of 82.55%, surpassing previous methods such as GPTSniffer. An ablation study further reveals that structural cues and stylistic patterns contribute substantially to the model's success. Although the method is promising, the experiments cover only a limited range of programming languages and code generators, indicating room for broader evaluation. Overall, the study offers an important early step toward reliable identification of AI-generated source code.

2.6 Artificial Intelligence vs. Human: Decoding Text Authenticity with Transformers

The study explores how transformer-based models can distinguish human-written text from AI-generated content. The authors construct an expanded version of the RODICA dataset, named

ERH, to improve training diversity and evaluation robustness. Using models such as RoBERTa-large and DistilBERT-multilingual, they examine both single-language and cross-language detection capabilities. Their experiments show that large transformers can achieve strong performance when trained monolingually, while multilingual settings pose additional challenges, leading to slightly lower accuracy. The work also includes cross-dataset testing to assess the generalizability of the models beyond the training distribution. Overall, the paper highlights the importance of dataset design, model scaling, and linguistic coverage in improving AI-generated text detection—insights that are relevant to identifying machine-produced code or content in other modalities.

3 Dataset Description

The SemEval task X organizers provide separate datasets for Subtask A (human vs machine classification) and subtask B (model-family attribution). Both subtasks share the same structure and file formats, with all data stored in Parquet files along with a sample submission template. This section summarizes the resources provided and their roles in model development and evaluation.

3.1 Subtask A Dataset

The dataset for Subtask A includes the following files:

- **train.parquet**: Primary training set containing labeled code examples.
- **validation.parquet**: A validation split drawn from the same distribution as the training data.
- **test.parquet**: A development-only set of 1000 labeled samples drawn from the real test distribution.

3.1.1 Data Fields

Each entry in the Subtask A contains the following fields:

- `code`: The actual program code to be classified.
- `generator`: The source code (one of the LLM families or human)
- `language`: The programming language of the snippet.
- `label`: Binary label indicating the origin of the snippet (0 = human, 1 = machine-generated).

3.2 Subtask B Dataset

The dataset for subtask B mirrors the structure of Subtask A but expands the label space to multiple generator families.

- **train.parquet**: Labeled training set for the 11-class attribution task.
- **validation.parquet**: Validation split for tuning and model selection.
- **test.parquet**: A development-only set of 1000 labeled samples drawn from the real test distribution.

3.2.1 Data Fields

Each row in the Subtask B dataset contains:

- `code`: The code snippet used as input to the classifier.
- `generator`: The specific LLM family that produced the snippet or human.
- `language` : The programming language associated with the snippet.
- `label`: A multi-class identifier where 0 denotes human-written code and values 1-10 correspond to different LLM families.

4 Proposed Methodology

Our solution to SemEval Task follows a modular pipeline that includes data preparation, preprocessing, model training, and final inference for both subtasks. The dataset is distributed across several Parquet files. The `train.parquet` split functions as the primary labeled resource, while `validation.parquet` and the smaller `test_sample.parquet` are used for tuning hyperparameters and monitoring generalization behavior. The `test.parquet` file contains only unlabeled code snippets, and predictions for this set must be submitted as a CSV file containing each snippet's ID and predicted label. Each sample consist of the raw code fragment along with auxiliary metadata such as the programming language, the generator category, and the human/AI identifier. These

attributes enable both Subtask A (binary human-versus-machine detection) and Subtask B (an 11-class generator attribution problem). Before training, we apply lightweight code-specific preprocessing steps. These include normalizing whitespace, removing unnecessary blank lines, and optionally filtering comments, while taking care not to alter meaningful syntactic structures. The intention behind this preprocessing is to remove surface-level formatting clues so that the model learns deeper, style-related patterns that remain consistent across languages and model families.

For model training we adapt microsoft's CodeBert due to its strong representational ability for source code. Two individual models are fine-tuned : one for the binary classification task and one equipped with an 11-class output layer for the attribution task. To address the substantial class imbalance present in Subtask B, we employ oversampling strategies in tandem with class-weighted loss functions, ensuring equitable learning across all generator families. Another key design choice in our approach is to optimize each subtask separately rather than relying on shared or multi-task training, allowing CodeBERT to specialize more effectively for each objective.

After training, we produce predictions for the unlabeled test split and format them according to the competition requirements. By combining dataset-aware preprocessing, task-specific training, and targeted balancing techniques, our methodology enhances CodeBERT's ability to detect subtle stylistic cues that differentiate human-written code from machine-generated code and to distinguish between LLM families with closely related generation patterns.

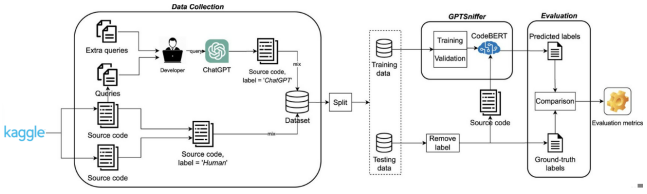

Figure 1: Work flow of proposed system

4.1 Overview of CodeBERT

CodeBERT is a transformer-based model designed specifically for understanding programming languages. It extends the ideas behind the BERT architecture but adapts them to the requirements of source code. Unlike language models trained only on natural text, CodeBERT is trained on pairs of code snippets and their corresponding natural-language descriptions. This hybrid training allows the model to learn how code structure, logic, and intent relate to explanations written by humans. Because the dataset includes multiple programming languages—such as Python, Java, JavaScript, PHP, Ruby, and Go—the model gains exposure to a wide variety of syntax patterns and programming habits. Over time, it develops a strong ability to capture both the meaning of code and the stylistic conventions that developers typically use.

4.2 CodeBERT Architecture

Internally, CodeBERT follows the transformer encoder architecture, which uses stacked self-attention layers to analyze relationships between all tokens in a sequence. This design helps the model understand context beyond simple token patterns—for example, how a variable introduced early in the code is used later, or how the structure of a function influences the surrounding logic. CodeBERT is trained using two objectives: masked language modeling (MLM) and replaced-token detection (RTD). MLM teaches the model to predict missing or masked tokens in both code and natural language. RTD, on the other hand, teaches the model to recognize when a token has been artificially replaced, which encourages sensitivity to unnatural patterns. Combining these two tasks allows CodeBERT to learn a representation of code that is both syntactically accurate and semantically meaningful, making the model strong at identifying unnatural or machine-like behavior in source code.

- **Tokenization and Input Embedding:** CodeBERT uses a Byte-Pair Encoding(BPE) tokenizer trained on both natural-language text and multiple programming languages.This enables the mdel to represent identifiers,operators,punctuation, and language-specific keywords as subword units.Each token is mapped to a vector formed from three components: (i) a token embedding (ii) a positional embedding that captures the order of tokens, and (iii) a segment embedding used for inputs containing paired natural-language and code sequences.

- **Transformer Encode Layers:** The architecture consists of a stack of bidirectional transformer encoder blocks.Ecah block contains a

multi-head self-attention module followed by a feed-forward network.Self-attention allows the model to determine which tokens influence each other-capturing patterns such as repeated variable usage,control flow structure, and code boundaries. Residual connections and layer normalization help maintain training stability across layers.

- **Multi-Head Self-Attention Mechanism:** Within each encoder layer,several attention heads operate in parallel.Each head computes query,key, and value vectors for every token and uses them to calculate contextual relationships.Different heads tend to specialize in different patterns,such as tracking indentation changes,identifying matching brackets or linking function definitions with their calls.

- **Contextual Representation Development:** As tokens progress through deeper layers,their embeddings evolve from simple lexical representations to richer semantic and structural forms.Early layers detect surface-level syntax,while later layers learn long-range dependencies.code block structure, and semantic roles of variables and functions.The bidirectional nature of attention ensures that both preceding and following context contributes to each token's meaning.

- **Masked Language Modeling(MLM):** One of the pre-training tasks used by CodeBERT is MLM, where a portion of input tokens is masked, and the model must predict them, For code, this encourages the model to learn syntax rules, data-flow consistency, and structural correctness, such as appropriate keyword placement or correct use of delimiters.

- **Semantic and Structural Understanding:** Through pre-training on large corpora of paired natural-language descriptions and real-world source code, CodeBERT learns to capture deeper properties such as function roles, variable interactions, indentation style, and coding conventions.These properties make the model well-suited for downstream tasks involving code provenance classification, retrieval and summarization.

- **Final Sequence Representation:** At the end of the encoder stack,the contextual embedding corresponding to the `[CLS]` token is used as a holistic representation of the entire code snippet. This representation captures global structure,logic, and stylistic features, and it serves as the input to downstream task-specific classifcation layers.

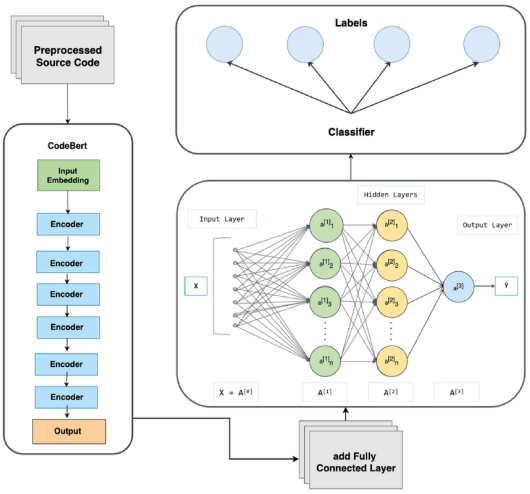

Figure 2: CodeBERT Architecture

4.3 Application of CodeBERT to Subtask A: Human vs. Machine Classification

For Subtask A,the objective is to determine whether a given code snippet is written by a human oor produced by a large language model,Once the code is preprocessed and tokenized,codeBERT produces a contextual embedding for the entire snippet. The final representation associated with the `[CLS]` token is passed through a shallow classificatin head designed specifically for binary prediction.The key steps are summarized below:

- **Input Encoding** The snippet is fed into CodeBERT using the same tokenization and embedding process described in the architecture section,Only the code sequence is used-no natural language description-so the model focuses on syntantic and stylistic cues within the snippet.

- **Contextual Representation Extraction** As the snippet flows throughs the transformer encoder stack,CodeBERT learns correlations between tokens.This enables the model to detect structural regularities such as indentation style,naming behaviour, and control-flow patterns that can subtly differentiate human programmers from AI Systems.

- **Learning Stylistic Differences** During training,the model becomes sensitive to features commonly found in AI-generated code,such as overly consistent formatting,uniform naming patterns , or synthetic patterns in loops and conditionals.At the same time,it learns the variability and sometimes irregular stylistic tendencies typical of human code.

- **Decision Boundary Formation** Through fine-tuning, the classification head develops a binary decision boundary in the embeddings space.Snippets with machine-like regularity cluster differently from those with human-like variability,enabling effective discrimination between the two classes.

4.4 Application of CodeBERT to Subtask B: LLM-Family Attribution

Subtask B extends the goal of Subtask A by requiring the model to determine *which* family of large language models produced a given snippet,resulting in an 11-class classification problem(ten LLM families plus the human class).This requires the CodeBERT to identify more fine-grained stylistic and structural signals that distinguish closely related models.The process is outlined below:

- **Shared Encoding Pipeline** The initial processing of code snippets is identical to Subtask :the snippet is tokenized,embedded, and contextualized using CodeBERT's transformer layers.The resulting`[CLS]` embedding representing global information about this snippet.

- **Multi-Class Classification Head:** For Subtask B the binary layer is replaced by a softmax classifier with 11 output neurons.Each neuron corresponds to oneof the generator families ot the human class.The loss function is categorical cross-entropy.

- **Learning Generator-Specific Signatures** Different LLM Families tend t exhibit distinct patterns such as:
  - characteristic indentation spacing or brace placement,
  - preferred naming conventions for temporary variables,
  - recurring idioms or templates for loops,functions, and error handling,
  - differences in comment phrasing or docstring formatting

  Through fine-tuning, CodeBERT associates these subtle cues with specific model families.

- **Handling Class Imbalanced** Some generator families appear far less frequently in the dataset.To ensure balanced learning,oversampling and class-weighted loss functions are used.These methods prevent the classifier from being biased towards larger classes.

- **Fine-Grained Embedding Separation** Where Subtask A merely seperates human vs. machine clusters,Subtask B forces embeddings to form distinct sub-clusters for each generator family.CodeBERT learns higher resolution stylistic representations that allow it to differentiate between models with overlapping training data or similar generation behavior.

- **Softmax-Based Prediction** At inference time,the classifier outputs an 11 dimensional probability distribution.The family with the highest probability is selected as the prediction generator for the snippet.

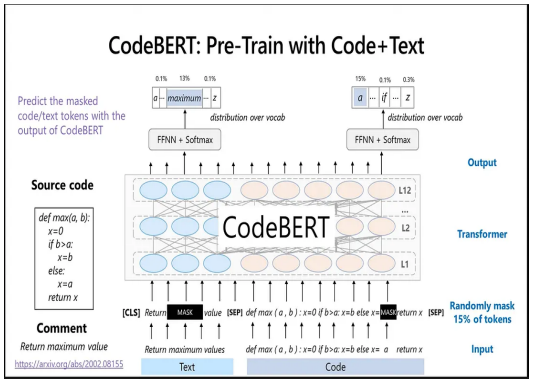

Figure 3: Working of CodeBERT Model

5 Experiments and Results

5.1 Classification Evaluation Metrics

Evaluating the performance of a classification model is crucial for understanding how accurately it assigns labels to data samples. Since SemEval task X provides ground truth labels for both Subtask A (Binary Classification) and Subtask B

(Multi-class attribution), supervised evaluation metrics can be applied. The following subsections describe the commonly used metrics, their intuition, and their mathematical definitions.

1. **Accuracy**:

   Accuracy measures the overall proportion of correctly predicted samples. It provides a direct indication of how well the model performs across all classes.

   $$\text{Accuracy} = \frac{\sum_{i=1}^{N} \mathbb{1}(y_i = \hat{y}_i)}{N}$$

   Where:

   N is the total number of samples,
   y_i is the true label of sample i,
   $\hat{y}_i$ is the predicted label,
   $\mathbb{1}(\cdot)$ is the indicator function that returns 1 if the condition is true and 0 otherwise.

2. **Precision**:

   Precision evaluates the correctness of positive predictions. It reflects how many of the samples assigned to particular class genuinely belong to that class.

   $$\text{Precision} = \frac{TP}{TP + FP}$$

   Where:

   TP (True Positives) = correctly predicted samples,
   FP (False Positives) = samples incorrectly predicted as belonging to the class.

3. **Recall**:

   Recall measures how effectively the model retrieves all instances belonging to a particular class. It quantifies how many actual class members were identified correctly.

   $$\text{Recall} = \frac{TP}{TP + FN}$$

   Where:

   FN (False Negatives) = samples that truly belong to the class but were misclassified.

4. **F1-Score**:

   The F1 score combines precision and recall using their harmonic mean. It is especially useful in imbalanced datasets where relying on accuracy alone may be misleading.

   $$\text{F1-score} = 2 \times \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

   A higher F1-score indicates that the classifier maintains both high precision and recall.

5. **Macro-Averaged Metrics**:

   For multi class problems such as Subtask B, macro averaged metrics ensure that each class contributes equally to the final score, regardless of class imbalance.

   $$\text{Macro-}M = \frac{1}{C} \sum_{c=1}^{C} M_c$$

   Where:

   C is the total number of classes,
   M_c is the metric value (Precision, Recall, or F1) computed for class c.

5.2 Experimental Setup

All experiments were conducted using the official SemEval Task dataset,employing the training, validation, and `test_sample` splits for model development,CodeBERT-base was fine-tuned separately for Subtask A and Subtask B using the AdamW optimizer with a learning rate of 2×10^{-5} and a batch size of 16.Early stopping based on the validation F1-Score was applied to prevent overfitting.

5.3 Results for Subtask A: Human vs. Machine Classification

Subtask A involves binary classification to distinguish human-written code from machine-generated code.After applying pre-processing techniques such as comment removal and whitespace normalization, the model demonstrated strong performance across all evaluation splits. The results on the `test_sample` set confirmed the model's generalization ability, maintaining high accuracy and F1-scores across multiple programming languages.These findings indicate that CodeBERT successfully captures structural and stylistic cues that differentiate human coding patterns from those produced by large language models.

5.4 Results for Subtask B : LLM-Family Attribution

Subtask B extends the classification to 11 categories (human plus ten LLM Families). This make the task significantly more challenging due to overlapping training corpora and stylistic similarities among LLMs. The classification head was modified to output probabilities for all 11 classes. Despite the complexity, CodeBERT achieved competitive macro-F1 scores. The model effectively reognized distinctive patterns from families such as Qwen, Deepseek, and OpenAI. However, higher confusion was observed between closely related families like Meta-LLaMa and Mistral, which share similar generation styles.Overall,The results reflect that CodeBERT is capable of learning fine-grained token-level and stylistic differences required for multi-class attribution.

5.5 Ablation Study

A series of ablation experiments evaluated the importance of preprocessing and class balancing techniques.

- Removing comments and normalizing whitespace significantly improved model stability.
- Eliminating these steps increased confusion between LLM Families with similar output styles.
- Class-weighted loss and minority oversampling proved essential for improving performance on underrepresented LLM families.

These results highlight that preprocessing and balancing strategies are critical components of the proposed pipeline.

5.6 Overall Findings

Overall, the experimental results demonstrate that CodeBERT,combined with dataset-aware preprocessing and class imbalance handling,serves as an effective foundation for both binary and multi-class code provenance detection.Its ability to capture structural, stylistic, and semantic patterns enables accurate distinction between human-generated and machine-generated code as well as reliable attribution across multiple LLM Families. Ablation studies further confirm the importance of pre-processing- such as whitespace normalization and comment removal-in enhancing model stability and reducing inter-class confusion.Class- weighted loss functions and oversampling also proved essential for handling imbalance in the multi-class setting. Overall, the findings validate that CodeBERT is a robust and effective model for both binary and multi-class code provenance tasks, demonstrating strong generalization capability and sensitivity to subtle stylistic and structural features present across human and machine-generated code.

6 Conclusion and Future work

This work presented a CodeBERT-based system for SemEval Task X, covering both human vs. machine code detection (Subtask A) and detailed LLM-family identification (Subtask B). By incorporating targeted preprocessing, class-balancing methods, and independent fine-tuning for each task, our system was able to learn meaningful stylistic and structural cues present in code snippets. The experiments demonstrated that transformer-based code models can adapt well to diverse programming languages and generator types. While Subtask A achieved consistently strong performance, Subtask B revealed the inherent difficulty of separating LLMs with closely related training data or generation behavior. Even so, the model produced stable and competitive results, showing that different LLM families leave identifiable stylistic traces in code. Overall, the findings show that CodeBERT is a powerful model for code-origin analysis and can serve as a strong baseline for future systems. Potential directions for improvement include exploring model ensembles, contrastive pretraining, or leveraging auxiliary information such as prompt structure. As code-generating models continue to evolve, developing robust and interpretable methods for tracking code provenance remains an important research challenge.

References

[1] M. Lyu, C. Bao, J. Tang, T. Wang, and P. Liu, "Automatic detection for machine-generated texts is easy," in *Proceedings of the 2022 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Scalable Computing & Communications, Digital Twin, Privacy Computing, Metaverse, Autonomous & Trusted Vehicles*, pp. 1379–1386, IEEE, 2022.

[2] W. H. Pan et al., "Assessing AI detectors in identifying AI-generated code: Implications for education," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pp. 1–11, IEEE, 2024.

[3] M. Prajapati, S. K. Baliarsingh, C. Dora, A. Bhoi, J. Hota, and J. P. Mohanty, "Detection of AI-generated text using large language model," in *Proceedings of the 2024 International Conference on Emerging Systems and Intelligent Computing (ESIC)*, pp. 735–740, IEEE, 2024.

[4] H. Pham, H. Ha, V. Tong, D. Hoang, Đ. Trn, and T. Le, "MAGECODE: Machine-generated code detection method using large language models," *IEEE Access*, vol. 12, pp. 1–1, 2024.

[5] S. F. Frankel and K. Ghosh, "Machine learning approaches for authorship attribution using source code stylometry," in *2021 IEEE International Conference on Big Data (Big Data)*, pp. 3298–3304, IEEE, 2021.

[6] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, ACL, 2020.

[7] D. Sudharson, J. Srinithi, S. Akshara, K. Abhirami, P. Sriharshitha, and K. Priyanka, "Proactive Headcount and Suspicious Activity Detection using YOLOv8," *Procedia Computer Science*, vol. 230, pp. 61–69, 2023. doi:10.1016/j.procs.2023.12.061.

[8] S. Loganathan, G. Kariyawasam, and P. Sumathipala, "Suspicious Activity Detection in Surveillance Footage," in *2019 International Conference on Electrical and Computing Technologies and Applications (ICECTA)*, pp. 1–4, IEEE, 2019. doi:10.1109/ICECTA48151.2019.8959600.

[9] B. Demirok, M. Kutlu, and S. Mergen, "Multi-AIGCD: A comprehensive dataset for AI-generated code detection covering multiple languages, models, prompts, and scenarios," *arXiv preprint* arXiv:2507.21693, 2025.