

ARTICLE

Agentic AI Code Generation vs. Human Authorship: Stylometric Detection in Firmware Written for Arduino Platform

Rajani Navoda Gunawardhana* and Thareendhra Keerthi Wijayasiriwardhane

Department of Industrial Management, Faculty of Science, University of Kelaniya, Kelaniya, Sri Lanka

*Corresponding Author: Rajani Navoda Gunawardhana. Email: hrrngun251@kln.ac.lk

Received: 26 April 2026; Accepted: 09 June 2026; Published: 22 October 2026

ABSTRACT: In modern education, learning Science, Technology, Engineering, and Mathematics (STEM) has become an essential requirement. Physical computing, embedded systems, and Internet of Things (IoT) are widely used in STEM education, and the Arduino platform has emerged as a *de facto* standard for teaching and learning these technologies. However, with recent advancements in Artificial Intelligence (AI), students are tempted to over-rely on Large Language Models (LLMs). As a result, the auto-generation of software source codes by students using LLMs poses a significant threat to achieving the Intended Learning Outcomes (ILOs) of their course of study. Unlike conventional software source code developed for general-purpose computing, firmware written for the Arduino platform directly interacts with hardware components and follows domain-specific syntactic and structural patterns. As a result, existing AI-based auto code detectors often underperform when detecting auto-generated firmware for the Arduino platform. This paper presents an approach for detecting LLM-generated firmware source code for the Arduino platform using code stylometry features. The proposed method evaluates lexical and syntactic features to distinguish LLM-generated and human-written firmware for the Arduino platform. The experimental results demonstrate an accuracy of 0.9413, indicating that the proposed approach can effectively classify whether firmware is authored by a human or generated by an LLM.

KEYWORDS: AI-generated firmware; Arduino; STEM education; large language models; auto code detection

1 Introduction

In recent years, there has been a significant global shift toward Science, Technology, Engineering, and Mathematics (STEM) education [1]. STEM education has become a strategic priority for many countries seeking to maintain competitiveness in the global economy [2]. With the advent of the 4th Industrial Revolution, societies have been on a new quest to be able to cope with global developments [2]. Consequently, there has been a growing demand for individuals who can innovate and apply interdisciplinary knowledge to real-world problems [2]. STEM education enables students to acquire the most in-demand skills, such as critical thinking, creativity, information literacy, digital literacy, collaboration, etc. [3]. Therefore, almost all educational systems in the world have started to include science, technology, engineering, and mathematics in their curricula with the intention of offering a promising future for their students [2].

Acquiring practical command of programming skills in embedded computing systems, cyberphysical systems, and the IoT through project-based learning can significantly enhance STEM education and better prepare students for technology-driven careers [4]. In STEM education, the Arduino platform is used as a *de facto* standard for teaching and learning IoT, physical computing, and embedded systems [5].

In developing 21st-century skills like problem-solving, creativity, critical thinking, and entrepreneurial skills, Arduino plays a prominent role in STEM education [5]. For teaching and learning, from simple electronic applications to complex robotic systems, the Arduino platform provides invaluable, easy-to-use, open-source, free toolchain support for both students and teachers [6]. As an example, Arduino is used in low-cost circuit emulators, digital filter algorithms, and radar systems for signal processing [6].

Arduino and Tinkercad online simulators are significantly improving and deepening programming competencies, such as data-oriented and object-oriented programming, recursion, and data structures [7]. From the electrical engineering student's perspective, Arduino is used to develop electronic theory and practical knowledge [7]. Furthermore, in primary education, Arduino is effectively used to teach students basic programming and robotics in different computational skills such as abstraction, algorithmic thinking, decomposition, and generalization [8]. In secondary education, Arduino is used in technology and physics subjects, and in the development of interdisciplinary STEM subjects [9]. Therefore, Arduino is used to learn programming languages and as a tool for scientific experiments [9]. Thus, STEM-focused Arduino practical activities are significant for adopting an interdisciplinary approach to education.

Today, with the advancements in Artificial Intelligence (AI), Large Language Models (LLMs) based software source-code assistance has become an unavoidable part of modern software development, including domains such as IoT, embedded systems, and physical computing [10]. This assistance offers many new features, such as snippet generation, partial code completion, and translation of the entire code into other programming languages [11]. Such capabilities make the system multilingual, allowing developers to work smoothly between different programming environments with the utmost flexibility and efficiency.

In the context of education, this auto-code generation capability can enhance the learning experience as students can solve programming challenges easily within a short time [12]. The source codes generated by the LLMs provide students with an easy way to complete their assignments. However, over-relying on LLMs to generate firmware source code for the Arduino platform by students poses a significant threat to achieving the Intended Learning Outcomes (ILOs) of their course of study [13]. This over-reliance on LLMs negatively affects students' critical thinking, problem-solving, and computational thinking abilities, and their creativity [13]. With this overreliance on AI-generated codes, students may become dependent on LLMs, and they might not understand the AI-generated code, may not know how to modify it if needed, and may not know how to best express their intentions in order to generate their code [12]. Teachers are also concerned about academic integrity and plagiarism. Though AI code generators positively impact the enhancement of the learning experience by causing less stress and reducing coding errors, there is a potential risk that students may not develop a deep understanding of programming fundamentals and the ability to write code without AI assistance [13]. This ultimately affects computer programming education [13].

There are several AI-based auto-code detectors available. However, most of them are focused on text detection, not on program source code detection [14]. Recently, several content detectors like DetectGPT [15], GPTZero [16], and CopyLeaks [17] have been developed for commercial use, and they are good at distinguishing AI-generated from human-written texts in natural languages. However, they exhibit significant shortcomings when applied to detect AI-generated program source codes [18].

DetectGPT [15] is a zero-shot detection model that classifies AI-generated content by analyzing log probability curvatures. The AIGC detector by Xu and Sheng [19], has optimized the perturbation and scoring processes in the DetectGPT model for source code detection. The Zero-Shot Synthetic Code Detector [20] compares original and rewritten source code, boosting the Area Under the Receiver Operating Characteristic (AUROC) by up to 0.29 in benchmarks. DetectCodeGPT [21] avoids resorting to external LLMs by perturbing code with spacing. It raises the AUROC by 0.076 over baseline techniques.

Caliskan-Islam et al. [22] developed a machine-learning model for source code authorship attribution, using code stylometry features built from Abstract Syntax Trees (ASTs), achieving 0.98 accuracy in identifying the authors by coding style. Bukhari et al. [23] extended this method to detect AI-generated code by adding lexical and syntactic features. For the classification, Random Forest Classifier (RFC), k-Nearest Neighbors (KNN), Support Vector Machine (SVM), and XGBoost (XGB) were used and achieved up to 0.92 accuracy in the classification for source codes written in the C programming language. Idialu et al. [24] applied XGB to 140 code stylometry features, such as line length and function usage, to identify AI-generated Python code, with an F1 score of 0.91. Li et al. [25] extended this method by introducing lexical, structural, and semantic features. This model outperformed with 0.9 accuracy.

Oedingen et al. [26] developed an embedding-based model using Deep Neural Networks (DNN), RFC, and XGB. This model shows accuracies of up to 0.98. GPTSniffer [27] has been developed based on the CodeBERT classifier, which extracts, tokenizes, and then classifies the code with high accuracy to detect AI-generated Python and Java source codes. Hoq et al. [28] proposed a traditional machine learning and neural network technique for classifying the origins of source code, using Term Frequency-Inverse Document Frequency (TF IDF) embedding, Code2Vec, and AST-based Neural Network (ASTNN). According to the authors, all the above-discussed models achieve accuracies between 0.90 and 0.95.

Commercially available AI-generated text detectors are built upon classical machine learning models and zero-shot detection techniques [15–17]. These detectors are primarily designed to identify patterns, structure, and linguistic characteristics of natural language content such as essays, articles, or chatbot responses. However, when applied to AI-generated program source code detection, these models show low accuracy with high false-positive and false-negative rates [18]. These general-purpose models struggle to effectively analyse and classify program source codes due to the structural and syntactic differences between natural language and program source codes and the complexity of programming languages.

Existing auto-code detection models mainly focus on detecting LLM-generated source codes written in Java, Python, JavaScript, C, and other traditional high-level programming languages for general-purpose computing [29]. However, Arduino focuses on developing firmware mainly for microcontroller platforms [30]. The existing auto-code detection models trained for detecting LLM-generated program source code written in traditional high-level programming languages for general-purpose computing underperform when used for detecting firmware source code, written with unique syntaxes for the Arduino platform.

This paper presents a novel approach for detecting LLM-generated firmware source code for the Arduino platform, using code stylometry features. The proposed classifier has been trained and evaluated on a dataset consisting of 204 firmware source code snippets, including 102 human-written programs from official Arduino libraries and 102 AI-generated source code produced by ChatGPT-4o, Perplexity AI, and Gemini 2.5 Flash. This appears to be the first such attempt to construct a classifier for detecting LLM-generated firmware for the Arduino platform. The previously developed classifiers for detecting AI-generated program source code written in programming languages like C, C++, Java, and Python using specific code stylometry features [22] have been extended by introducing novel lexical and syntactic feature sets.

2 Methodology

This research uses code stylometry features introduced by Caliskan-Islam et al. [22] to identify code authorship. By extending the prior work by Bukhari et al. [23], this study introduces a novel feature set focused on the Arduino platform. Code stylometry features were selected because prior research [22,23] demonstrates that lexical and syntactic characteristics extracted from Abstract Syntax Trees (ASTs) capture

programmers' stylistic patterns and enable highly accurate authorship attribution and AI-generated code detection. In this approach, code snippets are represented as ASTs, where paths between leaf nodes can be transformed into fixed-length vector representations [28]. These vectors allow machine learning models to analyze structural code patterns [28]. This process highlights important programming structures and relationships, aiding in a deeper understanding of code behavior and stylistic patterns. Studies by Caliskan-Islam et al. [22] and Bukhari et al. [23] show that such features can achieve accuracy levels above 0.90 in identifying code authors and distinguishing AI-generated code from human-written code.

Zero-shot detection and embedding-based models [14,18,19,25–27] were not applicable because these models typically rely on probabilistic language patterns or require large-scale datasets for effective training. Given the relatively small dataset and the structural nature of firmware source code, stylometric features derived from ASTs provide a more interpretable and computationally efficient representation of programming style. As shown in Fig. 1, the proposed approach is based on lexical and syntactic feature generation, focusing on the unique functionalities and characteristics of the Arduino platform.

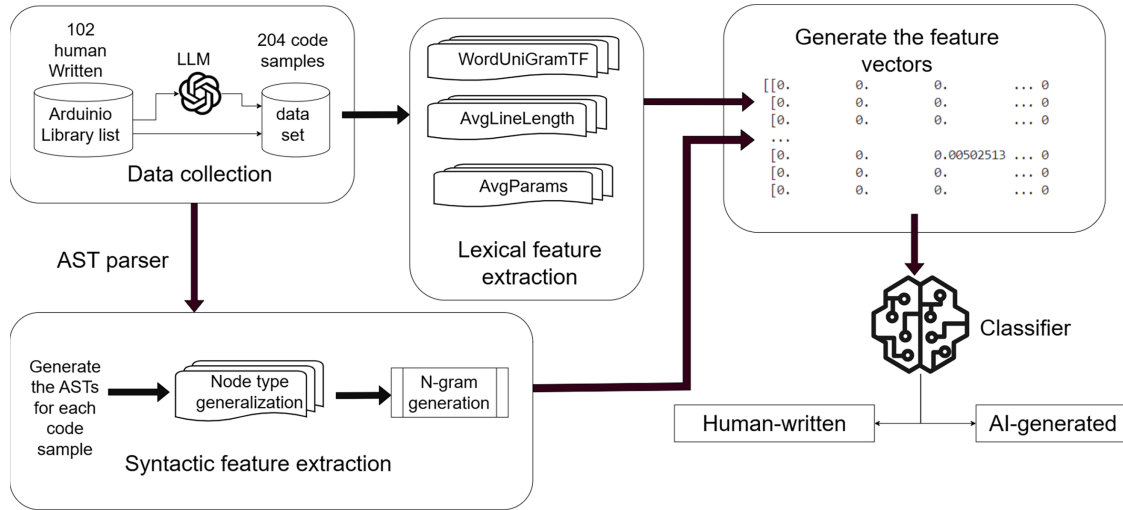


Figure 1: Illustration of the overview of the research. The process begins with data collection. Then, the lexical and syntactic feature vectors are extracted using AST node types and textual patterns. These vectors feed classification models that identify the origin of each code sample.

2.1 Data Collection

The dataset consists of 204 firmware samples written/generated for the Arduino platform, of which 102 are human-written, whereas the rest are AI-generated. Human-written firmware were collected from the Arduino Library List [31] which contains many repositories, including Arduino libraries and examples. Source code samples committed prior to 2019 were extracted as human-written program codes, as the capability to generate source codes using LLMs, such as OpenAI GPT-3, Codex, etc., were not available then. During data extraction, source code samples were collected from 35 different Arduino libraries. For each extracted human-written code, a corresponding source code sample was generated using an LLM. 102 AI-generated samples were collected, maintaining a balanced dataset. ChatGPT-4o, Gemini 2.5 Flash, and Perplexity AI models were prompted to generate source code for the Arduino platform based on the same project descriptions as the human-written code samples. To reduce prompt bias and avoid dependence on a single standardized prompt style, five different prompt styles were used. Although the dataset size is limited, 5-fold StratifiedGroupKFold Cross Validation (CV) and balanced sampling were employed to

mitigate overfitting and provide reliable performance estimation. CV was performed using a fixed random seed of 42 to ensure reproducibility. A grouped CV protocol was implemented, where all samples belonging to the same project were assigned to the same fold using unique Group_IDs derived from file paths. This ensured that related samples, including paired human-written and AI-generated code from the same task description, did not appear in both the training and testing sets, thereby reducing the risk of data leakage. For each fold, the training set contained 163–164 samples, while the testing set contained 40–41 samples, corresponding to 80% training and 20% testing split. The class distribution was kept nearly balanced in every fold, with around 81–82 human-written samples and 81–82 AI-generated samples in the training set, and 20–21 samples from each class in the testing set. Across all five folds, the group overlap between the training and testing sets was verified as zero, confirming that no group leakage occurred during model evaluation. Future work will expand the dataset across multiple LLMs and real student submissions.

2.2 Lexical Features

Lexical features help capture the textual characteristics of code [22]. These characteristics would be indicative of identifying whether a code was written by a human or generated by AI. In this study, 11 lexical features were extracted to represent stylistic patterns present in Arduino firmware. Table 1 illustrates the lexical features used in the proposed approach. These features are easy to calculate and provide the first level of categorization using only text analysis. This study uses a subset of the lexical feature set for code stylometry suggested by Caliskan-Islam et al. [22].

Table 1: Lexical features used in methodology.

Feature	Definition
AverageLoL	Average length of lines in the source code, calculated as the mean number of characters per line.
StdDevLineLength	The standard deviation of the character lengths of each line.
NoOfWords	Logarithm of the total number of words in the source code, delimited by whitespace.
NoOfComments	Logarithm of the number of single-line comments divided by the total number of characters.
if-Occured	Log-normalized frequency of the keyword if, normalized by total file length in characters.
else-Occured	Log-normalized frequency of the keyword else, normalized by total file length in characters.
elseif-Occured	Log-normalized frequency of the keyword else if, normalized by total file length in characters.
do-Occured	Log-normalized frequency of the keyword do, normalized by total file length in characters.
while-Occured	Log-normalized frequency of the keyword while, normalized by total file length in characters.
for-Occured	Log-normalized frequency of the keyword for, normalized by total file length in characters.
switch-Occured	Log-normalized frequency of the keyword switch, normalized by total file length in characters.

Lexical feature vectors are created based on the calculated features from the code. A feature vector is a structured numerical representation of a single piece of code, where each element corresponds to a specific lexical characteristic that has been extracted during the analysis. After computing these features for a given code sample, the values are arranged in a fixed and consistent order to form the vector. When all code samples are processed in this way, the resulting vectors are combined into a feature matrix, where each row represents one code sample, and each column represents a particular lexical feature. This structure ensures that all samples are represented uniformly, making it possible to directly compare them based on their textual style. These standardized feature vectors can then be used by machine learning models to identify patterns and classify the code. Table 2 shows a sample of the extracted feature matrix.

RFC, KNN, SVM, and XGB were used to evaluate the model's performance. These models were considered for evaluation based on the following reasons. SVM and KNN are non-tree-based algorithms with good performance in simple classification tasks, while RFC and XGB are ensemble tree-based classifier

variants that provide high performance even in complex tasks [23]. In the implementation, the KNN classifier was configured with five nearest neighbors, while the RFC and SVM classifiers were implemented with random state values of 32 and 23, respectively, to ensure reproducibility. The SVM classifier used the default Radial Basis Function (RBF) kernel configuration, and the XGB classifier was evaluated using its default hyperparameter settings. Five-fold StratifiedGroupKFold CV was performed for each classifier on the dataset, and performance was evaluated using accuracy, F1 score, precision, and recall.

Table 2: Lexical feature matrix sample.

Sample	Avg	Std	Words	Coms	else	if	else if	do	while	for	switch
Sample1	26.68	26.70	4.31	0	−5.57	0.0	0.0	0.0	−6.26	0.0	0.0
Sample2	31.14	34.40	5.22	0	−7.95	0.0	0.0	0.0	−7.95	−6.85	0.0
Sample3	36.52	37.85	5.70	0	−8.38	0.0	0.0	0.0	−8.38	−6.77	0.0
Sample4	30.58	36.42	5.98	0	−6.38	0.0	0.0	−7.48	−8.58	−7.48	0.0

2.3 Syntactic Features

Syntactic features are used to analyze the structure of the code. Syntactic features are mainly focused on how different components of code, such as statements, loops, and function calls, are organized and related to each other. Syntactic features can be used to identify the overall structure and flow of the code. ASTs were extracted for each source code sample. ASTs are a tree-based representation of the code, mainly used in code stylometry. At a high level, the proposed approach generates the ASTs for each code sample and generalizes the ASTs, computes the n-gram counts for bigrams and trigrams, and trains the classifier based on these syntactic features.

Generate and generalize the ASTs: By using a parser for C/C++, code samples can be converted to ASTs. This method is appropriate since Arduino firmware is built upon C and C++ languages. Consequently, C/C++ parsers can effectively capture the syntactic structure and control-flow constructs present in Arduino programs, enabling accurate extraction of AST-based stylometric features. Fig. 2a shows the sample Arduino code, and Fig. 2b shows the corresponding AST. This research focuses on void setup() and void loop() functions, which are specific to Arduino boards. The parsing process needs to be capable of generating an AST even from incomplete code snippets, which are common in Arduino sketches. An AST is constructed while parsing the code, and each node of it corresponds to a certain syntactic element, like control structures, functions, or variables. This tree structure reflects the program's control flow and organizational hierarchy, which are essential for analyzing its structural characteristics.

When the AST is generated for the source file, this method extracts the subtree representing each function. From the generated AST, it is possible to identify specific functions that are relevant for Arduino operations, mainly void setup() and void loop(). These functions are important as they specify setup configurations and the main operational loop of an Arduino sketch. The heterogeneousness of node types in an AST may lead to a feature space with a high dimension in terms of feature extraction. Consider two different Arduino code snippets, where one AST has a node named function_call: Serial.print and another has function_call: Serial.println. If this were not standardized, these nodes would be treated as different, while conceptually they are performing similar tasks as function calls to print output within the Arduino environment. This level of granularity will introduce unnecessary complexity and inconsistency in comparing similar code structures across samples. To manage this complexity and improve the efficiency of subsequent classification algorithms, node types are generalized to broader categories. For instance,

conditional keywords “else”, “for”, “if”, and “while” were grouped into a generic node type “Condition-Keyword,” even though the key parsing produced these phrases as individual node types. Old ASTs were converted into new ASTs with fewer syntactical entities while maintaining the same syntactic meaning and workflow using the proposed newly defined generalized categories. Using this generalization strategy, the original AST node types were mapped into 45 generalized node categories, reducing feature dimensionality while preserving the syntactic meaning and workflow of the program. The original ASTs were therefore converted into new AST representations with fewer syntactic entities while maintaining their structural semantics.

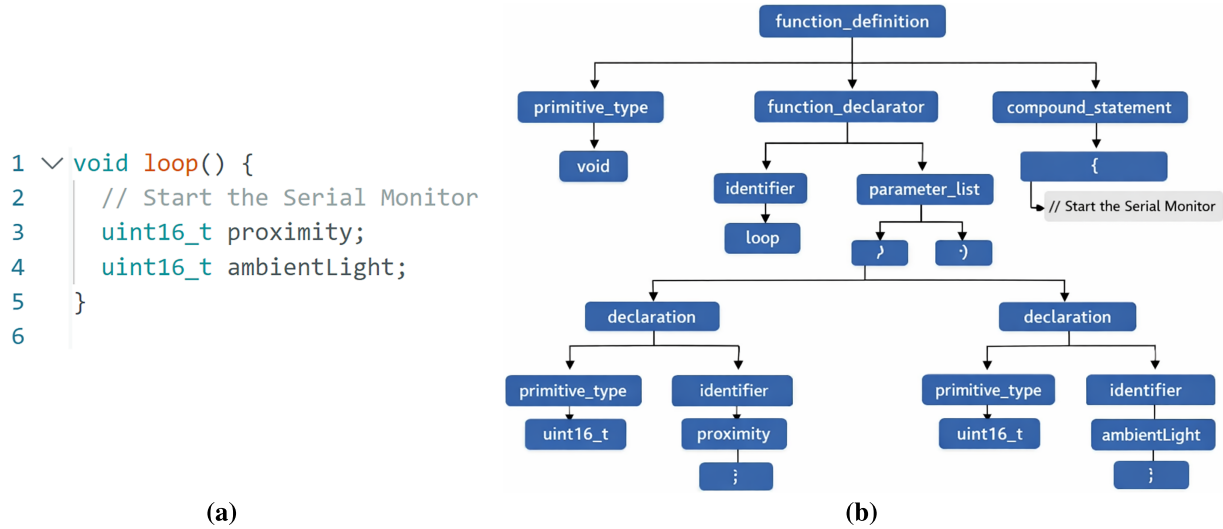


Figure 2: Illustration of syntactic feature extraction using ASTs from Arduino firmware code. (a) Sample Arduino code snippet. (b) Generated AST representation of the code showing hierarchical syntactic elements.

n-gram feature vectors: As the traditional classification algorithms depend on feature vectors, they cannot be provided with structured data in ASTs, and vectorizing ASTs is required. However, it must be done properly as it will always lead to information loss. A popular approach, *n*-gram-based features, was used for the AST vectorization process. Generating *n*-grams requires serializing the AST into a linear sequence. This is done using Depth-First Traversal (DFT) and listing all encountered nodes, where each node in the tree represents a syntactic element from the code.

When the AST is serialized into the linear series of the nodes, an *n*-gram is formed with the use of a sliding window approach, depending on the number of nodes. The sliding window means looking at small groups of nodes (like 2 or 3 nodes at a time) in the AST, one after another. Each group represents a short, continuous part of the code’s structure. For this method, *n* values of 2 and 3 were used for bigram and trigram. Once the *n*-grams were generated, the frequency of each unique *n*-gram in the code was calculated.

Since it is more reasonable to compare the frequencies of different *n*-grams in different source code samples, the frequency of each *n*-gram is normalized by the total number of *n*-grams in the sequence. See Eq. (1). For example, if the total number of bigrams is 10, and the bigram (‘FunctionCall’, ‘Condition’) appears 2 times, its normalized frequency would be 0.2.

$$\text{Normalized_Frequency} = n_gramFrequency \div total_n_gramFrequencies \quad (1)$$

Every AST is then converted into a feature vector form, wherein each element is the normalized frequency of occurrence of an n-gram. Dimensionality reduction was subsequently applied to refine the feature set. Features with a value of 0.0 normalized frequency across all samples were eliminated, as such, all-zero features contributed no discriminative power in distinguishing between human-written and AI-generated firmware. Initially, the AST bigram feature matrix contained 2116 features, which were reduced to 199 informative features after removing zero-frequency columns. Similarly, the AST trigram feature matrix originally consisted of 97,336 features, which were reduced to 520 informative features after dimensionality reduction. Removing these redundant features reduced the dimensionality of the dataset and enhanced computational efficiency. This process not only accelerated both the training and testing phases but also mitigated the risk of overfitting by reducing noise and enabling the model to concentrate on informative feature variations across samples. After generating the feature vectors, the same classifiers used for lexical feature classification were used to evaluate the model's performance. Five-fold StratifiedGroupKfold CV was followed for the classification. The dataset containing 204 firmware samples was divided into five folds while maintaining the class distribution between human-written and AI-generated code samples.

3 Results

The syntactic and lexical feature extraction included the extraction of features specific to the Arduino platform, such as hardware-dependent commands, and structured programming patterns that are specific to embedded systems. The experiments were done for lexical, syntactic, and a combination of lexical and syntactic feature vectors. In each experiment, accuracy, F1 score, precision, and recall were measured. Five-fold StratifiedGroupKfold CV was performed for each model to mitigate the risk of over-fitting and data leakage. For the experiments, RFC, SVM, KNN, and XGB were used as classification models.

3.1 Experiments Using Combined Features

Experiments for the combination of lexical and syntactic features (E1 and E2) were done on both bigram and trigram vectors. This feature set captures both lexical and syntactic features to detect AI-generated and human-written code samples. The evaluation results for the bigram vectors are presented in [Table 3](#), which details the performance of various models using combined features. Similarly, the results for the trigram vectors are shown in [Table 4](#). According to the results for the bigram vectors, the XGB model achieved the highest accuracy with 0.9413 and F1 score of 0.9423. The KNN model demonstrated a slightly lower performance with 0.7209 accuracy. For the trigram vectors, the highest value of accuracy of 0.9316 is for the XGB model, which then shows that the XGB model had better results at both detecting and correctly classifying the samples of code. KNN performed with the lowest accuracy, with 0.7305.

Table 3: Model performance comparison with combined features for bigram vectors (E1).

Model	Performance Metrics			
	Accuracy	F1 Score	Precision	Recall
KNN	0.7209	0.7685	0.6550	0.9310
RFC	0.8777	0.8706	0.9059	0.8429
SVM	0.8483	0.8592	0.8075	0.9214
XGB	0.9413	0.9423	0.9357	0.9519

Table 4: Model performance comparison with combined features for trigram vectors (E2).

Model	Performance Metrics			
	Accuracy	F1 Score	Precision	Recall
KNN	0.7305	0.7769	0.6622	0.9412
RFC	0.8728	0.8686	0.8903	0.8529
SVM	0.8337	0.8392	0.8180	0.8629
XGB	0.9316	0.9320	0.9262	0.9419

3.2 Experiments Using Syntactic Features

The next experiments used only syntactic features (E3 and E4). Syntactic features extract the structural patterns of the code. The same experimental design of methodology was used and tested 4 classifiers on the syntactic feature vectors for bigrams and trigrams. The evaluation results for bigram vectors are shown in [Table 5](#), and the results for trigram vectors are shown in [Table 6](#). The XGB model achieved the highest performance with an accuracy and F1 score of 0.9167 for the bigram vectors, and for the trigram vectors RFC model achieved the highest performance, with a 0.9117 accuracy and F1 score of 0.9094. According to the E3 and E4 results, non-tree-based classifiers such as KNN show lower accuracies than tree-based classifiers XGB and RFC, indicating that tree-based models are more effective at capturing complex and discriminative stylometric patterns for detecting AI-generated firmware source code.

Table 5: Model performance comparison with syntactic features for bigram vectors (E3).

Model	Performance Metrics			
	Accuracy	F1 Score	Precision	Recall
KNN	0.7845	0.8177	0.7146	0.9600
RFC	0.9071	0.9093	0.8916	0.9314
SVM	0.8923	0.8997	0.8478	0.9605
XGB	0.9167	0.9167	0.9151	0.9224

Table 6: Model performance comparison with syntactic features for trigram vectors (E4).

Model	Performance Metrics			
	Accuracy	F1 Score	Precision	Recall
KNN	0.7643	0.7949	0.6964	0.9440
RFC	0.9117	0.9094	0.8989	0.9221
SVM	0.8922	0.8964	0.8544	0.9472
XGB	0.9022	0.8994	0.9061	0.8956

3.3 Experiments Using Lexical Features

In this experiment (E5), the classification performance was evaluated using Lexical features which capture the basic textual elements of the codes. The results for classification are presented in [Table 7](#).

Table 7: Model performance comparison with lexical features (E5).

Model	Performance Metrics			
	Accuracy	F1 Score	Precision	Recall
KNN	0.8188	0.8132	0.8022	0.8285
RFC	0.8826	0.8785	0.9033	0.8587
SVM	0.8728	0.8629	0.9015	0.8336
XGB	0.8580	0.8525	0.8637	0.8431

Tables 8–11 summarize all experimental results, showing that the combined bigram features for XGB model provide the highest classification performance with highest accuracy of 0.9413 and an F1-score of 0.9423. This indicates that XGB is the most effective classifier for detecting AI-generated Arduino firmware when both lexical and syntactic characteristics were considered together. Table 8 evaluates the performance of the KNN model across different feature types with Confident Interval (CI), while Table 9 presents the results obtained using the RFC. Similarly, Table 10 evaluates the SVM model, and Table 11 summarizes the performance of the XGB classifier. The results show that tree-based classifiers are more effective in capturing syntactic patterns, whereas non-tree-based classifiers show stronger performance in detecting lexical-level features. However, when lexical and syntactic features were combined, the performance of the non-tree-based classifiers decreased, indicating that these models may struggle to handle the increased feature complexity and interactions introduced by the combined feature representation. In contrast, tree-based classifiers demonstrated better capability in managing such complex feature spaces.

Table 8: KNN performance comparison for different feature types with 95% CI.

Feature	KNN			
	Accuracy	F1 Score	Precision	Recall
Combined Bigram	0.7209 (0.65, 0.78)	0.7685 (0.70, 0.83)	0.6550 (0.60, 0.70)	0.9310 (0.84, 1.00)
Syntactic Bigram	0.7845 (0.71, 0.85)	0.8177 (0.76, 0.87)	0.7146 (0.63, 0.79)	0.9600 (0.90, 1.00)
Lexical	0.8188 (0.75, 0.88)	0.8132 (0.72, 0.90)	0.8022 (0.75, 0.85)	0.8285 (0.68, 0.96)

Table 9: RFC performance comparison for different feature types with 95% CI.

Feature	RFC			
	Accuracy	F1 Score	Precision	Recall
Combined Bigram	0.8777 (0.78, 0.96)	0.8706 (0.76, 0.97)	0.9000 (0.83, 0.98)	0.8429 (0.70, 0.98)
Syntactic Bigram	0.9071 (0.84, 0.96)	0.9093 (0.84, 0.96)	0.8916 (0.80, 0.97)	0.9314 (0.85, 1.00)
Lexical	0.8826 (0.84, 0.92)	0.8785 (0.83, 0.91)	0.9033 (0.83, 0.97)	0.8587 (0.78, 0.93)

XGB is a gradient-boosted tree-based model that differs from KNN, SVM, and even RFC in its sequential learning strategy. In XGB, trees are constructed iteratively, with each new tree attempting to correct the errors made by the previous trees. This enables XGB to capture complex nonlinear relationships between lexical and syntactic features. Moreover, XGB is well-suited to high-dimensional and heterogeneous feature spaces, making it effective when different types of stylometric features are combined. Therefore,

when lexical and syntactic features are integrated, XGB can identify meaningful interactions between textual coding style and AST-based structural patterns. In contrast, other models may interpret the additional features as noise or may be less effective in separating human-written and AI-generated code.

Table 10: SVM performance comparison for different feature types with 95% CI.

Feature	SVM			
	Accuracy	F1 Score	Precision	Recall
Combined Bigram	0.8483 (0.79, 0.90)	0.8592 (0.81, 0.90)	0.8075 (0.73, 0.88)	0.9214 (0.86, 0.97)
Syntactic Bigram	0.8923 (0.85, 0.93)	0.8997 (0.86, 0.93)	0.8478 (0.79, 0.90)	0.9605 (0.90, 1.00)
Lexical	0.8728 (0.81, 0.92)	0.8629 (0.79, 0.93)	0.9015 (0.84, 0.96)	0.8336 (0.70, 0.95)

Table 11: XGB performance comparison for different feature types with 95% CI.

Feature	XGB			
	Accuracy	F1 Score	Precision	Recall
Combined Bigram	0.9413 (0.89, 0.98)	0.9423 (0.89, 0.98)	0.9357 (0.87, 0.99)	0.9519 (0.87, 1.00)
Syntactic Bigram	0.9167 (0.86, 0.97)	0.9167 (0.86, 0.97)	0.9151 (0.84, 0.98)	0.9224 (0.83, 1.00)
Lexical	0.8580 (0.80, 0.91)	0.8525 (0.78, 0.92)	0.8637 (0.78, 0.94)	0.8431 (0.76, 0.91)

Among all evaluated models, XGB obtained the highest accuracy and F1 score using the combined syntactic and lexical feature representation, demonstrating that the integration of structural and textual stylometric features improves the detection of AI-generated Arduino firmware source code. Syntactic features also performed well for XGB model, achieving bigram accuracy of 0.9167 and an F1 score of 0.9167, which is not surprising given the fact that syntactic features encapsulate structural properties of code more pertinent to programming than the simple textual attributes represented by lexical features. As expected, the lexical features alone yielded a lower accuracy and F1 score of 0.8580 and 0.8525. This is because they capture superficial features, which contribute less toward the task-specific distinctions required in this classification. For the other models, the results indicate that they perform well when using feature types that align with their individual learning characteristics. However, when lexical and syntactic features are combined, their performance decreases, suggesting that these models are less effective at handling the increased feature complexity and interactions introduced by the combined feature space.

The XGB classifier, with the highest accuracy of 0.9413, shows that tree-based classifiers are more suitable for program source code classification tasks because of their unbounded ability to adapt to complex data boundaries and handle structural feature sets efficiently. This may indicate that models like XGB, and other tree-based methods, are especially helpful for the proper classification of code into AI-generated vs. human-written code in domain-specific use cases such as firmware programming in the Arduino platform.

4 Discussion

This study investigated whether code stylometry techniques based on lexical and syntactic features can effectively distinguish between human-written and AI-generated firmware for the Arduino platform. The experimental results show that stylometric patterns embedded in firmware code can provide reliable signals for identifying the origin of code samples. Across all experiments, the combination of lexical and syntactic

features consistently produced a high classification performance, confirming that both textual and structural characteristics of code contribute to distinguishing AI-generated programs from human-authored firmware.

The results indicate that tree-based classifiers demonstrate stronger performance in Arduino AI-generated code detection, particularly when syntactic and combined feature representations are used. Among the evaluated models, RFC and XGB consistently achieved higher accuracies, suggesting that tree-based classifiers are more effective at capturing complex coding patterns than non-tree-based approaches. Lexical features mainly capture surface-level textual characteristics, such as naming style, token usage, and formatting tendencies. Therefore, they produced relatively lower accuracies when used alone. When lexical and syntactic features are combined, the feature representation becomes more comprehensive because it includes both textual coding characteristics and structural code patterns. This richer representation can improve detection accuracy, especially for models such as XGB, which are capable of handling high-dimensional and heterogeneous feature spaces. However, the increase in feature dimensionality may negatively affect some classifiers, causing them to underperform compared with using syntactic features alone.

The results show that feature representation plays a critical role in classification performance. After dimensional reduction, 199 syntactic bigram features, 520 syntactic trigram features, 970 combined bigram features, 1291 combined trigram features and 11 lexical features were retained for classifier training and evaluation. The results indicate that when lexical and syntactic features were combined, the classifiers generally achieved higher accuracy compared with using either feature set individually for the XGB. For example, the combined bigram feature experiment achieved an accuracy value of 0.9413, while the combined trigram feature experiment achieved an accuracy of 0.9316. However, the increase in the number of features resulted in a high-dimensional feature space, which negatively affected the performance of some classifiers. For example, the RFC achieved an accuracy of 0.9071 using syntactic bigram features consisting of 199 feature representations. However, when the combined bigram feature set was used, the number of features increased to 970, and the RFC accuracy decreased to 0.8777. A similar trend was observed for SVM, where the syntactic bigram features achieved an accuracy of 0.8923, while the combined bigram features produced a lower accuracy of 0.8483. These results indicate that increasing the feature dimensionality can reduce model performance, particularly when the classifier is less effective at handling complex and high-dimensional feature spaces. In contrast, XGB benefited from the richer feature representation, achieving its highest accuracy of 0.9413 with the combined bigram feature set despite the increased dimensionality.

To further analyze the classification performance of the proposed approach, confusion matrices were examined for the evaluated classifiers. Confusion matrices provide a detailed representation of the classification behavior by illustrating the number of correctly and incorrectly predicted samples for each class. As shown in Fig. 3a, the confusion matrix for the XGB classifier using combined lexical and syntactic features with the bigram representation demonstrates that the majority of both human-written and AI-generated firmware samples were correctly classified. Specifically, the model correctly identified 95 human-written samples and 97 AI-generated samples, while only 7 human-written samples were misclassified as AI-generated and 5 AI-generated samples were misclassified as human-written. These results indicate that the stylometric features derived from the combination of lexical characteristics and AST-based structural patterns effectively capture programming style differences between human and AI-generated firmware.

Similarly, the trigram representation exhibits comparable classification performance, as shown in Fig. 3b. The XGB classifier correctly classified 94 human-written samples and 96 AI-generated samples, with only 8 human-written samples and 6 AI-generated samples incorrectly predicted. The improvement in correctly classified human-written samples suggests that the bigram representation captures slightly richer structural relationships between AST nodes compared to the trigram representation. However, the overall

performance difference between bigram and trigram representations remains relatively small, indicating that most discriminative stylistic information is already captured within shorter AST node sequences.

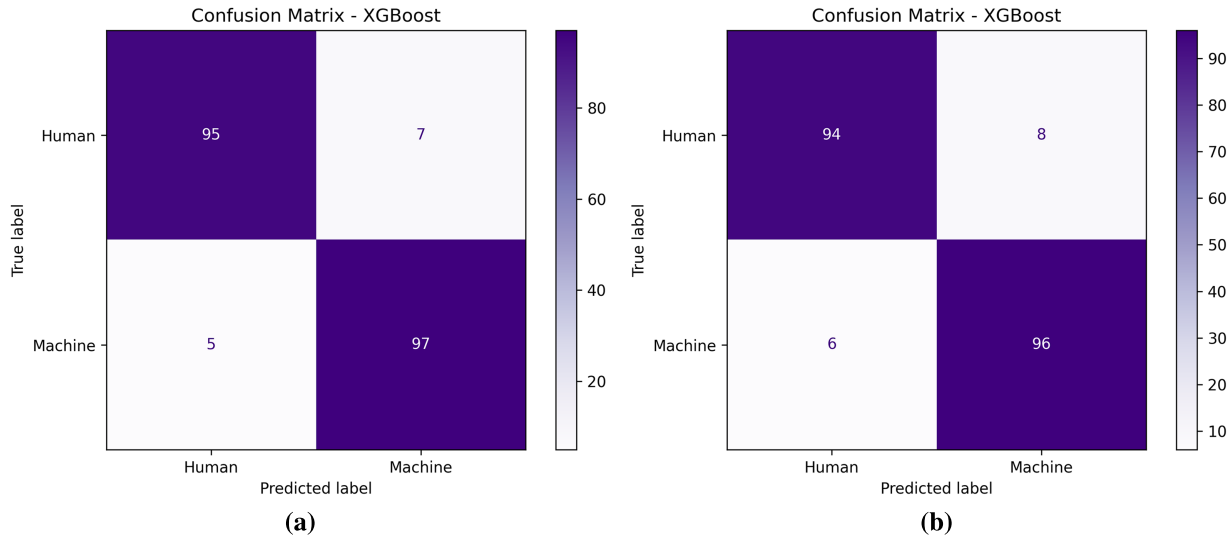


Figure 3: Confusion matrix for the bigram (a) and trigram (b) vector classifications with combined features.

Another important observation from the experimental results is the difference between bigram and trigram feature representations. Although trigram vectors capture longer structural relationships between AST nodes, the experimental results show that bigram features achieved higher accuracy than trigram features. This suggests that the most discriminative stylistic patterns for distinguishing human-written and AI-generated Arduino code are better captured through shorter AST node sequences. In contrast, trigram features may introduce additional dimensionality and complexity, which can reduce classification performance by adding less informative or noisy feature interactions. Therefore, bigram features provide a more effective and computationally efficient representation for AI-generated Arduino code detection while maintaining stronger classification accuracy than trigram-based features.

The results also highlight the importance of structural features in firmware source code analysis. Arduino programs follow a specific programming paradigm centered around the `setup()` and `loop()` functions. Because of this structure, stylistic differences between human programmers and AI-generated implementations often appear in the arrangement of control-flow constructs, function calls, and hardware interaction routines. AST-based syntactic features capture these structural characteristics effectively, enabling machine learning models to identify patterns associated with AI-generated code.

In addition, the dimensionality reduction process applied to AST n-gram features contributed significantly to improving classification efficiency. The original AST bigram feature space consisted of 2116 features, which was reduced to 199 informative features after removing zero-frequency columns. Similarly, the trigram feature space was reduced from 97,336 features to 520 informative features. This reduction not only decreased computational complexity but also helped reduce noise and potential overfitting by eliminating irrelevant features that do not appear in the dataset. As a result, the classifiers could focus on meaningful stylistic patterns present across the firmware samples.

Despite the encouraging results, several limitations should be acknowledged. Although the dataset was expanded to include 204 firmware samples from 35 Arduino libraries and multiple LLMs, it remains relatively small compared with datasets used in large-scale code-classification studies. Furthermore, the

human-authored firmware was obtained from publicly available Arduino libraries rather than authentic student submissions. Student-developed code often contains inconsistent formatting, incomplete abstractions, and debugging artifacts that may influence stylometric characteristics. In addition, the present study focused on stylometric feature engineering and conventional machine-learning classifiers. Future work should therefore evaluate the proposed approach on larger and more diverse datasets, including real student assignments, and compare its performance against transformer-based approaches, embedding-based methods, and other code-specific AI-detection techniques to establish its relative effectiveness.

4.1 Implication for STEM Education

The capability to differentiate AI-generated firmware source codes from human-authored ones for the Arduino platform is significantly important in the context of education, particularly in STEM education, in which the Arduino platform is used widely in teaching and learning computer programming and firmware development. This research shows high accuracy, precision, and recall across four classifiers for combined and syntactic feature vectors. These findings show that the proposed model can be a valuable tool for maintaining academic integrity and evaluating the students' real performance. The effectiveness of the model in promoting authentic learning experiences discourages the dependency on AI-based solutions such as LLMs, thereby encouraging students to develop essential coding skills. This helps improve the problem-solving and critical-thinking skills of the students. The use of this tool further resolves some broader issues in the education sector, in that all students are graded based on actual skill and not on the availability of sophisticated AI tools. These findings might lead to educational institutions toward the creation or refinement of policies regarding the use of AI tools in homework and project submissions to maximize the benefits of AI in education while steering clear of its potential pitfalls.

However, the proposed approach is intended to support educators rather than replace academic judgment. Detection results should not be treated as conclusive evidence of academic misconduct and should be considered alongside other assessment methods such as code walkthroughs, oral questioning, project documentation, version history analysis, and instructor evaluation.

4.2 Comparative Study

LLMs for code detection: Another experiment was conducted for available LLMs, such as ChatGPT and Gemini, to evaluate the performance of detecting AI-generated firmware source codes for the Arduino platform. Both human-written and AI-generated firmware source code samples were fed into these LLMs. For the result, every code sample submitted to ChatGPT and Gemini was classified as human written. ChatGPT has shown better performance in text detection and annotation tasks. These models handle only the human language [32]. However, with respect to the detection of programming languages, they perform with very low accuracy. The pure text-based focus of LLMs can only go so far in recognizing structural patterns unique to the source code, as opposed to the natural language patterns.

Models for detecting other programming languages when used for detecting Arduino firmware: Models to detect other programming languages like Python, C, JavaScript, and Java have been implemented, but these models are not applicable to detect Arduino firmware because of Arduino's low-level, hardware-focused nature. Python models are not applicable because Python relies on Python-specific syntaxes and conventions are not available in Arduino [33]. C, C++, and Java source code detection models are also not applicable because Arduino is used for microcontroller programming, and its unique functions are not present in other programming languages. While C++ and Java are object-oriented, Arduino programming is mainly procedural. Even though Arduino supports C++ syntax, most projects for Arduino avoid complicated object-oriented structures in favor of the simplicity and efficiency of function-oriented direct procedural code.

Models based on class-based architecture, with inheritance and polymorphism, as would be normal in Java and C++ applications, do not find these structures in Arduino code. This seriously restricts their effectiveness in its analysis and classification.

4.3 Baseline Comparison

The proposed Arduino firmware detection approach was compared with several baseline methods representing different categories of AI-generated code detection. These baselines included simple statistical classifiers, TF-IDF-based classifiers, a Code2Vec-style AST path embedding baseline, and a CodeBERT-style classifier. The purpose of this comparison was to examine whether the proposed Arduino-specific stylometric feature representation provides better classification performance than general textual, statistical, embedding-based, and transformer-based code representations. Table 12 summarizes the results from each baseline category and compares them with the proposed model.

Table 12: Model performance with baselines.

Model	Performance Metrics			
	Accuracy	F1 Score	Precision	Recall
Code2Vec	0.4846	0.1263	0.0923	0.2000
CodeBERT	0.8879	0.8999	0.8509	0.9600
Statistical_SVM	0.8654	0.8717	0.8631	0.8827
Statistical_RFC	0.9077	0.9053	0.9328	0.8827
Statistical_XGB	0.9126	0.9113	0.9385	0.8928
TF-TDF Combined	0.9105	0.9172	0.9186	0.9210
TF-IDF Linear	0.9155	0.9172	0.9186	0.9210

The TF-IDF baselines were included to evaluate the effectiveness of surface-level textual representations. Character-level TF-IDF, token-level TF-IDF, and combined TF-IDF representations were tested with Logistic Regression and Linear SVM classifiers. Among these models, the token-level TF-IDF with Linear SVM achieved the best performance, with an accuracy of 0.9155, precision of 0.9186, recall of 0.9210, and F1 score of 0.9172. This result shows that token distribution patterns are useful for distinguishing AI-generated Arduino firmware from human-written firmware. However, TF-IDF features mainly capture lexical-level characteristics and do not explicitly represent the structural organization of the code.

Simple statistical baselines were also evaluated using basic code-level measurements, such as line-length statistics, comment-related features, word counts, and keyword-frequency patterns. These features were tested with Logistic Regression, Linear SVM, RFC, and XGBoost classifiers. Among these models, Statistical_XGB achieved the highest performance, with an accuracy of 0.9126, precision of 0.9385, recall of 0.8928, and F1 score of 0.9113. Statistical_RFC also performed competitively, achieving an accuracy of 0.9078 and F1 score of 0.9053. These results indicate that simple stylometric measurements can capture useful differences between human-written and AI-generated firmware. Nevertheless, their performance remained lower than that of the proposed feature representation because statistical features alone do not fully capture the syntactic structure of Arduino programs.

A Code2Vec-style AST path embedding baseline was implemented to evaluate the effectiveness of neural AST path representations. In this baseline, AST path contexts were extracted from each Arduino firmware sample using start tokens, AST paths, and end tokens. The learned code vectors were then used as input to

a Logistic Regression classifier. This baseline produced weak classification performance, with an accuracy of 0.4846, precision of 0.0923, recall of 0.2000, F1 score of 0.1263, and AUROC of 0.5000. This result suggests that the Code2Vec-style representation was not suitable for the current dataset. A possible reason is that neural AST path embedding models generally require larger datasets to learn stable and generalizable code representations. In contrast, the proposed AST n-gram representation directly captures normalized syntactic patterns and is more suitable for a smaller domain-specific firmware dataset.

The CodeBERT-style classifier was included as a transformer-based code representation baseline. This model achieved an accuracy of 0.8879, precision of 0.8509, recall of 0.9600, F1 score of 0.8988, and AUROC of 0.9498. The high recall indicates that the model was able to identify many AI-generated samples. However, the comparatively lower precision suggests that it also misclassified some human-written firmware as AI-generated. This may be because CodeBERT-style models are commonly trained on general-purpose source code and may not fully capture the domain-specific characteristics of Arduino firmware, such as setup-loop structure, hardware-control functions, and embedded-system programming patterns.

4.4 Threats to Validity

Several factors may affect the validity and generalizability of the reported results. First, although the dataset includes firmware from multiple Arduino libraries and AI-generated code from several LLMs, its size remains relatively limited and may not fully represent the diversity of real-world Arduino development. Second, the AI-generated samples were created using predefined prompt variations. Consequently, alternative prompt engineering strategies not represented in the dataset may produce different stylometric characteristics. Third, only code generated by ChatGPT-4o, Gemini 2.5 Flash, and Perplexity AI were evaluated. However, code generated by other LLMs and AI coding assistants, such as Claude and GitHub Copilot, may exhibit different coding patterns that are not represented in the current dataset.

Another limitation is robustness against adversarial or post-processed AI-generated code. The present study evaluates firmware generated directly by multiple LLMs under diverse prompt styles but does not explicitly investigate scenarios in which AI-generated code is subsequently modified by human users, refined through iterative prompting, or combined with human-written code segments. Such hybrid and edited artifacts are increasingly common in educational settings and may reduce the discriminative stylometric signals captured by the proposed feature set. Future work should therefore evaluate detector robustness against manually edited AI code, hybrid human-AI firmware, and iterative prompt-refinement workflows to better reflect authentic student usage patterns.

A further limitation relates to the evaluation of general-purpose LLM-based detection. The comparison with ChatGPT and Gemini was intended as an exploratory assessment of widely available tools rather than a comprehensive benchmarking exercise. The study did not systematically investigate the effects of alternative prompting strategies, repeated queries, model parameter settings, or future model updates, which may influence classification behavior. Therefore, the reported findings should be interpreted within the specific evaluation conditions employed in this work and should not be generalized.

To mitigate potential data leakage, grouped stratified cross-validation was employed to ensure that samples derived from the same project remained within the same fold. However, external validation on independent datasets is required to confirm the robustness of the proposed approach. Furthermore, as false positives may occur, the proposed methodology should be considered a decision-support approach rather than a definitive indicator of AI-assisted authorship. Detection results should always be interpreted in conjunction with human judgment and other forms of assessment.

5 Conclusion

In conclusion, this research demonstrates the feasibility of distinguishing between AI-generated and human-authored firmware for the Arduino platform using code stylometry features, including lexical and syntactic measures, achieving high classification accuracy. The proposed approach was evaluated on a dataset of 204 firmware samples collected from 35 Arduino libraries, consisting of 102 human-written and 102 AI-generated programs. Experimental results show that combining lexical and syntactic stylometric features provides the most effective representation for distinguishing between AI-generated and human-authored firmware. The combined feature experiments achieved the highest classification accuracy of up to 0.9413 and F1 score of 0.9423 using XGB classifier with bigram vectors. Among the evaluated models, the XGB classifier achieved the best overall performance with the highest accuracy of 0.9413, demonstrating the effectiveness of tree-based ensemble learning methods for stylometric source code classification.

The study also highlights the importance of structural code representations in firmware detection tasks. Syntactic features derived from ASTs achieved high accuracy of 0.9167, outperforming lexical features alone, which achieved an accuracy of 0.8580. These findings confirm that structural code characteristics provide stronger discriminative signals than purely textual features.

Limitations of this study include the relatively small dataset size and the evaluation of firmware generated by only three LLMs. Further, higher-order n-grams and semantic feature representations were not investigated due to computational constraints. These factors may limit the generalizability of the results.

This study opens several fruitful avenues for future research. One of the key areas for future exploration is the improvement of model performance through advanced machine learning algorithms and ensemble learning techniques. These methods have significant potential to enhance performance. Furthermore, future work could focus on increasing the dataset coverage of more diverse Arduino libraries to enhance model generalizability. Using larger n-grams with cloud-based resources can better capture syntactic features, and the addition of more metrics for capturing the complexity and semantic features could allow for finer separation between AI-generated and human-written source codes. While the proposed approach demonstrates strong performance across multiple LLMs and prompt styles, further validation using real student submissions, additional coding assistants such as Claude and GitHub Copilot, and robustness testing against edited or hybrid AI-generated firmware remains an important direction for future research.

Acknowledgement: The authors would like to express their sincere gratitude to the University of Kelaniya, for providing the necessary academic environment and resources to conduct this research. During the preparation of this manuscript, the authors utilized ChatGPT-4o, Gemini 2.5 Flash, and Perplexity AI for generating AI-based firmware samples for the dataset. The authors have carefully reviewed and revised the output and accepted full responsibility for all content.

Funding Statement: The authors received no specific funding for this study.

Author Contributions: The authors confirm contribution to the paper as follows: Conceptualization, Thareendhra Wijayasiriwardhane; methodology, Rajani Gunawardhana and Thareendhra Wijayasiriwardhane; software, Rajani Gunawardhana; validation, Rajani Gunawardhana and Thareendhra Wijayasiriwardhane; formal analysis, Rajani Gunawardhana and Thareendhra Wijayasiriwardhane; investigation, Rajani Gunawardhana and Thareendhra Wijayasiriwardhane; data curation, Rajani Gunawardhana; writing—original draft preparation, Rajani Gunawardhana; writing—review and editing, Thareendhra Wijayasiriwardhane; visualization, Rajani Gunawardhana; supervision, Thareendhra Wijayasiriwardhane; project administration, Thareendhra Wijayasiriwardhane. All authors reviewed and approved the final version of the manuscript.

Availability of Data and Materials: The data that support the findings of this study are openly available in AI-Generated-vs-Human-Arduino-Firmware-code-detection at <https://github.com/Rajani-Navoda/AI-Generated-vs-Human-Arduino-Firmware-code-detection.git>.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Lukychova NS, Osypova NV, Yuzbasheva GS. ICT and current trends as a path to STEM education: implementation and prospects. CTE-2021; 2022 [cited 2026 Jan 1]. Available from: <http://academy.ks.ua/?p=17743>.
2. Türk N, Kalaycı N, Yamak H. New trends in higher education in the globalizing world: stem in teacher education. *Univers J Educ Res*. 2018;6(6):1286–304. doi:10.13189/ujer.2018.060620.
3. Widya, Rifandi R, Laila Rahmi Y. STEM education to fulfil the 21st century demand: a literature review. *J Phys Conf Ser*. 2019;1317(1):012208. doi:10.1088/1742-6596/1317/1/012208.
4. Zalewski J, Guo D, Kenny R, Wang X. From embedded systems to cyberphysical systems to the Internet of Things: consequences for STEM education. *Int J Comput*. 2017;11:48–53.
5. Sari U, Çelik H, Pektaş HM, Yalçın S. Effects of STEM-focused Arduino practical activities on problem-solving and entrepreneurship skills. *Australas J Educ Technol*. 2022;135–49. doi:10.14742/ajet.7293.
6. Martín-Ramos P, da Silva MML, Lopes MJ, Silva MR. Student2student: Arduino project-based learning. In: *Proceedings of the Fourth International Conference on Technological Ecosystems for Enhancing Multiculturality*; 2016 Nov 2–4; Salamanca, Spain. p. 79–84. doi:10.1145/3012430.3012500.
7. Tupac-Yupanqui M, Vidal-Silva C, Pavesi-Farriol L, Sánchez Ortiz A, Cardenas-Cobo J, Pereira F. Exploiting Arduino features to develop programming competencies. *IEEE Access*. 2022;10(1):20602–15. doi:10.1109/ACCESS.2022.3150101.
8. García-Tudela PA, Marín-Marín JA. Use of Arduino in primary education: a systematic review. *Educ Sci*. 2023;13(2):134. doi:10.3390/educsci13020134.
9. Marín-Marín JA, García-Tudela PA, Duo-Terrón P. Computational thinking and programming with Arduino in education: a systematic review for secondary education. *Heliyon*. 2024;10(8):e29177. doi:10.1016/j.heliyon.2024.e29177.
10. Petrović N, Konićanin S, Suljović S. ChatGPT in IoT systems: Arduino case studies. In: *2023 IEEE 33rd International Conference on Microelectronics (MIEL)*; 2023 Oct 16–18; Nis, Serbia. p. 1–4. doi:10.1109/MIEL58498.2023.10315791.
11. Aarti. Generative AI in software development: an overview and evaluation of modern coding tools. *Int J Fut Manag Res*. 2024;6(3):23271. doi:10.36948/ijfmr.2024.v06i03.23271.
12. Kazemitabaar M, Chow J, Ma CKT, Ericson BJ, Weintrop D, Grossman T. Studying the effect of AI code generators on supporting novice learners in introductory programming. In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*; 2023 Apr 23–28; Hamburg, Germany. p. 1–23. doi:10.1145/3544548.3580919.
13. Alier M, García Peñalvo FJ, Camba JD. Generative artificial intelligence in education: from deceptive to disruptive. *Int J Interact Multimed Artif Intell*. 2024;8(5):5–14. doi:10.9781/ijimai.2024.02.011.
14. Akram A. An empirical study of AI generated text detection tools. arXiv:2310.01423. 2023.
15. Mitchell E, Lee Y, Khazatsky A, Manning CD, Finn C. DetectGPT: zero-shot machine-generated text detection using probability curvature. In: *Proceedings of the 40th International Conference on Machine Learning*; 2023 Jul 23–29; Honolulu, HI, USA. p. 24950–62.
16. AI Detector-free AI checker for ChatGPT, GPT-5 & Gemini. [cited 2025 Dec 21]. Available from: <https://gptzero.me/>.
17. AI Content & Text Authenticity Detection | Copyleaks [Internet]. [cited 2025 Dec 21]. Available from: <https://copyleaks.com/>.

18. Pan WH, Chok MJ, Wong JLS, Shin YX, Poon YS, Yang Z, et al. Assessing AI detectors in identifying AI-generated code: implications for education. In: Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training; 2024 Apr 14–20; Lisbon, Portugal. doi:10.1145/3639474.3640068.
19. Xu Z, Sheng VS. Detecting AI-generated code assignments using perplexity of large language models. In: Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence. Washington, DC, USA: AAAI Press; 2024. p. 23155–62. doi:10.1609/aaai.v38i21.30361.
20. Ye T, Du Y, Ma T, Wu L, Zhang X, Ji S, et al. Uncovering LLM-generated code: a zero-shot synthetic code detector via code rewriting. AAAI. 2025;39(1):968–76. doi:10.1609/aaai.v39i1.32082.
21. Shi Y, Zhang H, Wan C, Gu X. Between lines of code: unraveling the distinct patterns of machine and human programmers. In: Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE); 2025 Apr 26–May 6; Ottawa, ON, Canada. p. 1628–39. doi:10.1109/ICSE55347.2025.00005.
22. Caliskan-Islam A, Harang R, Liu A, Narayanan A, Voss C, Yamaguchi F, et al. De-anonymizing programmers via code stylometry. In: Proceedings of the 24th USENIX Security Symposium; 2015 Aug 12–14; Washington, DC, USA. p. 255–70.
23. Bukhari S, Tan B, De Carli L. Distinguishing AI- and human-generated code: a case study. In: Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses; 2023 Nov 30; Copenhagen, Denmark. p. 17–25. doi:10.1145/3605770.3625215.
24. Idialu OJ, Mathews NS, Maipradit R, Atlee JM, Nagappan M. Whodunit: classifying code as human authored or GPT-4 generated—a case study on CodeChef problems. In: Proceedings of the 21st International Conference on Mining Software Repositories; 2024 Apr 15–16; Lisbon, Portugal. p. 394–406. doi:10.1145/3643991.3644926.
25. Li K, Hong S, Fu C, Zhang Y, Liu M. Discriminating human-authored from ChatGPT-generated code via discernable feature analysis. In: Proceedings of the 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW); 2023 Oct 9–12; Florence, Italy. p. 120–7. doi:10.1109/ISSREW60843.2023.00059.
26. Oedingen M, Engelhardt RC, Denz R, Hammer M, Konen W. ChatGPT code detection: techniques for uncovering the source of code. AI. 2024;5(3):1066–94. doi:10.3390/ai5030053.
27. Nguyen PT, Di Rocco J, Di Sipio C, Rubei R, Di Ruscio D, Di Penta M. GPTSniffer: a CodeBERT-based classifier to detect source code written by ChatGPT. J Syst Softw. 2024;214(1):112059. doi:10.1016/j.jss.2024.112059.
28. Hoq M, Shi Y, Leinonen J, Babalola D, Lynch C, Price T, et al. Detecting ChatGPT-generated code submissions in a CS1 course using machine learning models. In: Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1; 2024 Mar 20–23; Portland, OR, USA. p. 526–32. doi:10.1145/3626252.3630826.
29. Suh H, Tafreshipour M, Li J, Bhattiprolu A, Ahmed I. An empirical study on automatically detecting AI-generated source code: how far are we? arXiv:2411.04299. 2024.
30. Recktenwald GW, Hall DE. Using Arduino as a platform for programming, design and measurement in a freshman engineering course. In: Proceedings of the 118th American Society of Engineering Education Annual Conference & Exposition; 2011 Jun 26–29; Vancouver, BC, Canada.
31. Arduino Libraries. All Libraries. [cited 2026 Jan 1]. Available from: <https://www.arduino-libraries.info/libraries>.
32. Reiss MV. Testing the reliability of ChatGPT for text annotation and classification: a cautionary remark. arXiv:2304.11085. 2023. doi:10.48550/arXiv.2304.11085.
33. Python vs. Arduino | What are the differences? | StackShare. [cited 2026 Jan 18]. Available from: <https://stackshare.io/stackups/arduino-vs-python>.