Detection of GenAI-produced and student-written C# code: A comparative study of classifier algorithms and code stylometry features

Adewuyi Adetayo Adegbite
Postdoctoral Research Fellow, Department of Computer Science and Informatics, University of the Free State, Bloemfontein, South Africa; and Lecturer, Adekunle Ajasin University, Akunba Akoko, Ondo State, Nigeria
https://orcid.org/0000-0001-8195-1382

Eduan Kotzé
Associate Professor and Head of Department, Department of Computer Science and Informatics, University of the Free State, Bloemfontein, South Africa
https://orcid.org/0000-0002-5572-4319

Abstract

The prevalence of students using generative artificial intelligence (GenAI) to produce program code is such that certain courses are rendered ineffective because students can avoid learning the required skills. Meanwhile, detecting GenAI code and differentiating between GenAI-produced and human-written code are becoming increasingly challenging. This study tested the ability of six classifier algorithms to detect GenAI C# code and to distinguish it from C# code written by students at a South African university. A large dataset of verified student-written code was collated from first-year students at South Africa's University of the Free State, and corresponding GenAI code produced by Blackbox.AI, ChatGPT and Microsoft Copilot was generated and collated. Code metric features were extracted using modified Roslyn APIs. The data was organised into four sets with an equal number of student-written and AI-generated code, and a machine-learning model was deployed with the four sets using six classifiers: extreme gradient boosting (XGBoost), k-nearest neighbors (KNN), support vector machine (SVM), AdaBoost, random forest, and soft voting (with XGBoost, KNN and SVM as inputs). It was found that the GenAI C# code produced by Blackbox.AI, ChatGPT, and Copilot could, with a high degree of accuracy, be identified and distinguished from student-written C# code through use of the classifier algorithms, with XGBoost performing strongest in detecting GenAI code and random forest performing best in identification of student-written code.

Keywords

C# code, generative AI (GenAI) code, student-written code, machine-learning, code classification, code stylometry features

DOI: https://doi.org/10.23962/ajic.i35.21309

Recommended citation

Adegbite, A.A., & Kotzé, E. (2025). Detection of GenAI-produced and student-written C# code: A comparative study of classifier algorithms and code stylometry features. *The African Journal of Information and Communication (AJIC)*, *35*, 1–20. https://doi.org/10.23962/ajic.i35.21309

1. Introduction

Artificial intelligence (AI) has recently advanced significantly in several domains, transforming businesses, industries, and academia with its power, especially with the advent of large language models (LLMs) (Makridakis, 2017). Software development is one field where AI is having a strong influence (Kuhail et al., 2024). Generative artificial intelligence (GenAI) code is rapidly progressing due to advancements in natural language processing (NLP) and deep neural language models. GenAI code is autonomously generated source code (such as Python, C++, or C#) based on high-level requirements, specifications, or samples using machine-learning methods, especially deep-learning models (Odeh et al., 2024). The algorithms learn to produce new code that satisfies predetermined standards through the use of enormous repositories of pre-existing code, computer languages and patterns (Song et al., 2019).

The proliferation of GenAI code is fuelled by a few key technologies. LLMs, such as OpenAI's GPT (generative pre-trained transformer) series, have shown impressive capacities for comprehending and producing text that resembles human-written text (Cao et al., 2023). When used in code creation, these models can transform plain-language descriptions of desired functionality into executable code. Neural architectures are a crucial element in the production of code that satisfies predetermined requirements, and architectures are created specifically for code-creation activities (Dehaerne et al., 2022). The models learn to map input–output pairings, such as code snippets and their accompanying functionality. Code semantics, syntax, and patterns can be analysed and understood by such AI models when trained on extensive code repositories (Wan et al., 2023). In this field, transformers (Vaswani et al., 2017), convolutional neural networks (CNNs), and recurrent neural networks (RNNs) are model variants that are often used.

Thus, AI models can generate code that respects coding standards and complies with best practices (White et al., 2023) in the form of patterns, to solve common problems when using LLMs. Integrated development environments (IDEs) and code editors with AI capabilities such as Microsoft Copilot (Nghiem et al., 2024) offer intelligent and instantaneous code completion, recommendations and corrections (Cao et al., 2023). Through context, user behaviour, and pre-existing code analysis, these technologies improve developer efficiency and decrease mistakes. In codebases, AI algorithms can recognise common errors, anti-patterns, and code smells (potentially problematic code), and automatically recommend optimisations, refactorings, or repairs (Zhang et al., 2022).

While GenAI code is showing great promise, several issues and concerns need to be considered. Retaining good quality, correctness, and semantic meaning in produced code is still a challenge (Krasniqi & Do, 2023). AI models trained on biased or incomplete datasets may produce unfair or undesirable results (Varona & Suárez, 2022). Also, in the educational setting, there is the problem of students presenting GenAI code as their own when submitting computer-programming assignments, and this undermines the development of efficient and effective programmers. Accordingly, for educational institutions to maintain educational standards in their computer-programming courses, it is necessary to have tools that can assist educators in detection of possible student submission of assignments comprising AI-generated code instead of code written by the student. In line with this need for detection tools, the study presented in this article tested the ability of classifier algorithms to distinguish between AI-generated C# code and C# code written by first-year students at the University of the Free State, South Africa.

2. Literature review

A branch of software engineering called "code stylometry" examines programmers' writing styles and habits by analysing their source code (ShaukatTamboli & Prasad, 2013; Zafar et al., 2020). Code stylometry assigns authorship to sections of code based on their stylistic characteristics, much like the use of text stylometry in NLP, which examines writing styles to identify authors of texts (Benzebouchi et al., 2019; Ding et al., 2019). Code stylometry uses a variety of linguistic and structural elements taken from source code to describe programmers' writing styles (Odeh et al., 2024; Tereszkowski-Kaminski et al., 2022). These code stylometry features include lexical, structural, statistical, and syntactic features. Lexical features comprise vocabulary choices, comments, and programming construct usage. Structural features describe the arrangement of control structures, loops, and function definitions. Statistical features explain token distributional properties, language construct frequencies, and code metrics. Syntactic features, which are patterns in code structure, include indentation, naming conventions, and code organisation.

Code stylometry methodologies include elements of machine-learning, statistical analysis and NLP. To find patterns in code and determine authorship, researchers use methods including authorship attribution models, clustering algorithms, and classification techniques (Kalgutkar et al., 2019). A wide range of applications for code stylometry can be found in: authorship attribution; software evolution (which analyses the evolution of code-writing styles over time to understand developer behaviour, project dynamics, and software quality); code reuse and plagiarism detection (which compares writing styles and code patterns); and security analysis, forensics, malware analysis, and cyber-attack attribution (Caliskan et al., 2018; Czibula et al., 2022). Code stylometry tools are, thus useful for determining programmers' writing styles and code authorship (Tereszkowski-Kaminski et al., 2022). Source-code plagiarism is a critical issue in programming, and several studies have been conducted to explore detection methods. Table 1 lists key successful studies of code-plagiarism detection and the detection methods used.

Table 1: Studies on detection of code plagiarism (i.e., detection of plagiarised non-GenAI code)

Study	Programming language(s)	Detection method
Ebrahim and Joy (2023)	Java and C++	Binary classification via pretrained models: UnixCoder, PLBART, and CodeBERTa
Cheers et al. (2023)	Java	Combination of three classifiers: JPlag (structural), Graph ED (semantic), and BPlag (behavioural)
Eliwa et al. (2023)	C, C++, and Java	Similarity detection strategies using JPlag embedded with LMS
Cheers et al. (2021)	Java	Analysis of program-execution behaviour
Lalitha et al. (2021)	Java and Python	Combination of three classifiers: naïve Bayes, KNN, and AdaBoost
Srivastava et al. (2021)	Java	Levenshtein algorithm using edit distance between original code and perceived plagiarised code (the difference between the two codes, and the estimated plagiarism percentage)
Maryono et al. (2019)	Pascal	Euclidean distance on data for similarity measurement (by determining term-document matrices using keywords and programming characters, and then applying hierarchical clustering)
Zheng et al. (2018)	Python and Java	Abstract syntax trees
Portillo-Dominguez et al. (2017)	C++	Combination of three plagiarism tools (JPlag, Sherlock, and SIM)

GenAI production of programming code originated in early work on symbolic AI and automated programming. Early efforts focused on rule-based systems, expert systems, and genetic programming techniques. Notable progress was then achieved with the introduction of LLMs and deep-learning architectures. Today's GenAI-coding uses a variety of techniques and methods (Odeh et al., 2024; Raiaan et al., 2024). Natural language descriptions of functionality can be interpreted by NLP models, such as OpenAI's GPT series, and converted into executable code. Symbolic AI approaches produce code by combining statistical techniques with rule-based systems (Kotsiantis et al., 2024; Raiaan et al., 2024).

In the education context, according to Idialu et al. (2024), even without AI use, programming courses already suffer from high levels of plagiarism and contract-cheating (a situation where students give their tasks and assignments to an expert to solve the given problems). The use of AI tools for code generation (e.g., GitHub Copilot, Tabnine, Gemini, ChatGPT, Blackbox.AI, Mistral, Microsoft Copilot) is now further undermining academic integrity in such courses. The ease with which GenAI tools can generate code has produced a new form of academic dishonesty, with students submitting GenAI code as their own work (Kazemitabaar et al., 2024). Thus, it has now become necessary for academic instructor to find ways to detect possible AI-based plagiarism of code. Table 2 lists studies that have succeeded in detecting GenAI-produced Python, Java and C code as produced by LLMs including ChatGPT models, GitHub Copilot and others.

Table 2: Studies on detection of GenAI code

Study	Programming language	GenAI model(s) used	Detection method(s)
Corso et al. (2024)	Java	GitHub Copilot, Tabnine, ChatGPT, Google Bard	CodeBLEU and Levenshtein similarity analysis on both the generated code and developer code
Idialu et al. (2024)	Python	ChatGPT-4	Machine-learning classifier: XGBoost
Pan et al. (2024)	Python	ChatGPT (version not indicated)	Existing AI text detectors: GPTZero, GPT-2 Detector, DetectGPT, Sapling, and giant language model test room (GLTR)
Bukhari et al. (2023)	C	Code-cushman-001, code-davinci-001, code-davinci-002 (OpenAI code model variants)	Machine-learning classifiers: random forest, SVM, KNN, XGBoost

The study set out in this article focused on detection of GenAI C# code, and on distinguishing between GenAI and student-written C# code, because, to our knowledge, no such studies had previously been carried out in the South African educational context.

3. Study design

The GenAI C# code used in the study was produced by the Blackbox.AI, ChatGPT, and Microsoft Copilot LLMs, and the student-written code was produced by university students. Table 3 lists the versions used for each of the GenAI models.

Table 3: GenAI models used

Model	Version	Data freshness
Blackbox.AI	Blackbox.AI 1.0	Up to September 2024
ChatGPT-4o-mini	Gpt-4o-mini-2024-09-31	Up to October 2023
Microsoft Copilot	Copilot 1.1	Up to February 2023

The study tested the ability of six classifiers—extreme gradient boosting (XGBoost), k-nearest neighbors (KNN), support vector machine (SVM), AdaBoost, random forest, and soft voting (with XGBoost, KNN and SVM as inputs)—to distinguish between GenAI C# code and student-written C# code. These six classifiers were selected based on their successful application in existing studies of code stylometry.

XGBoost constructs decision trees iteratively optimising an objective function to strike a balance between prediction accuracy and model simplicity (Bukhari et al., 2023; Idialu et al., 2024). KNN is an instance-based learning algorithm that classifies data points based on the majority class of their nearest neighbours, identifying the k closest points in the feature space to a new example and assigning the most common class label among them. This method relies on the assumption that similar data points exist in proximity (Bukhari et al., 2024). SVM constructs a hyperplane, or set of hyperplanes, in a high-dimensional space to separate different classes, optimising the distance between the hyperplane and the nearest points of each class, called support vectors. By maximising this margin, SVM ensures robust classification focusing on generalisation to unseen data (Bukhari et al., 2024).

AdaBoost combines multiple weak classifiers, typically decision trees, to form a strong classifier focusing on misclassified examples by adjusting their weights, thereby forcing subsequent classifiers to pay more attention to complex cases. Each classifier contributes to the final prediction with a weight proportional to its accuracy. Random forest builds an ensemble of decision trees by training multiple trees on random subsets of the training data and features. This ensemble approach reduces the risk of overfitting and enhances the model's generalisation capabilities (Bukhari et al., 2024). Soft voting is an ensemble method that combines predictions from three base models, namely XGBoost, KNN, and SVM, to improve overall performance. Prediction is based on the average probabilities assigned by the models (Lalitha et al., 2021). All classifiers were trained using their default hyperparameters without additional tuning. The study also used SHAP (SHapley Additive exPlanations) to help explain the performance of the classifiers (Lundberg & Lee, 2017).

Ethical clearance

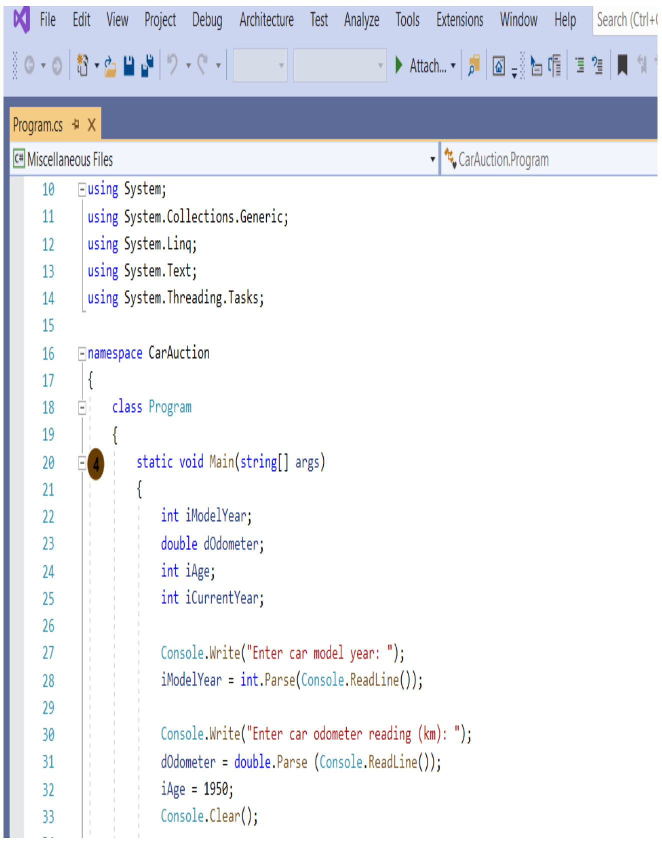
Ethical clearance for this study was granted by the General/Human Research Ethics Committee of the University of the Free State, South Africa, and the ethical clearance number is UFS-HSD2024/0601.

Data collection

During data collection, C# code files (*.cs) were extracted from Visual Studio C# solution files submitted in response to nine problems by 314 first-year Computer Science students at the Bloemfontein campus of the University of the Free State. The solution files, a sample of which is shown in Figure 1(a), were written in a controlled environment under the supervision of lecturers and student assistants, thus ensuring that the code produced was purely student-written. The same nine problems were presented (by a separate group of 219 students) to Blackbox.AI, ChatGPT, and Microsoft Copilot to solve, and this allowed for the collection of GenAI-generated C# code data, a sample of which is shown in Figure 1(b).

Figure 1: Sample C# code (for Problem 2)

(a) Student-written code


File Edit View Project Debug Architecture Test Analyze Tools Extensions Window Help
Search (Ctrl+(
Attach...
Program.cs
Miscellaneous Files
CarAuction.Program

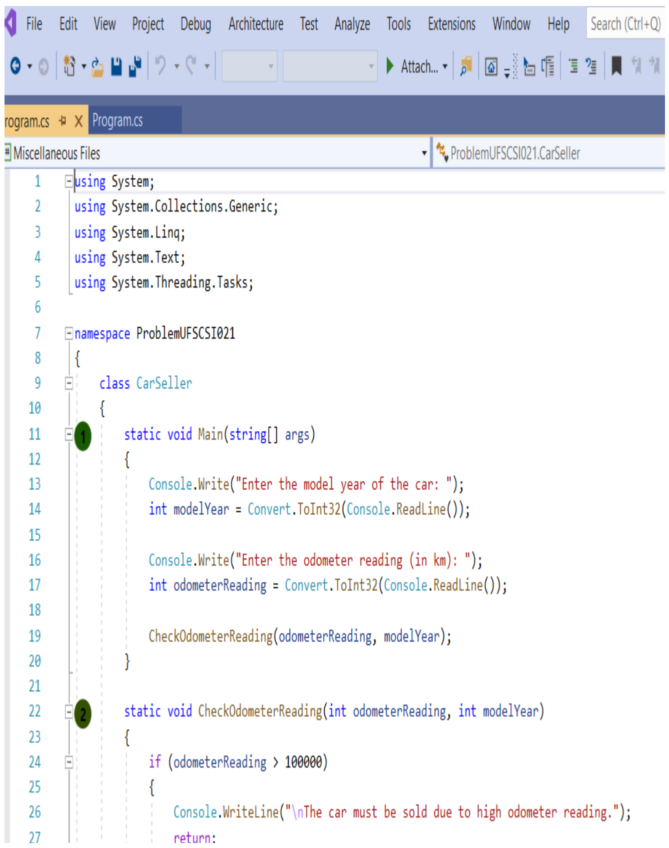
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CarAuction
{
    class Program
    {
        static void Main(string[] args)
        {
            int iModelYear;
            double dOdometer;
            int iAge;
            int iCurrentYear;

            Console.Write("Enter car model year: ");
            iModelYear = int.Parse(Console.ReadLine());

            Console.Write("Enter car odometer reading (km): ");
            dOdometer = double.Parse (Console.ReadLine());
            iAge = 1950;
            Console.Clear();
```

(b) AI-generated code


File Edit View Project Debug Architecture Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
Attach...
rogram.cs
Program.cs
Miscellaneous Files
ProblemUFSCSI021.CarSeller

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ProblemUFSCSI021
{
    class CarSeller
    {
        static void Main(string[] args)
        {
            Console.Write("Enter the model year of the car: ");
            int modelYear = Convert.ToInt32(Console.ReadLine());

            Console.Write("Enter the odometer reading (in km): ");
            int odometerReading = Convert.ToInt32(Console.ReadLine());

            CheckOdometerReading(odometerReading, modelYear);
        }

        static void CheckOdometerReading(int odometerReading, int modelYear)
        {
            if (odometerReading > 100000)
            {
                Console.WriteLine("\nThe car must be sold due to high odometer reading.");
                return;
```

The nine problems were labelled Problem 1 through 9 (Appendix C), with the ordering in ascending order of difficulty, i.e., Problem 9 was the most difficult. In the student-written C# code dataset, there were 1,043 solutions across the nine problems (Table 4). In the GenAI C# code dataset (Table 5), there were 1,120 GenAI C# categorised code solutions across the nine problems, and 195 uncategorised code solutions in total. (The uncategorised code solutions were those for which the GenAI model could not be clearly identified.)

Table 4: Student-written C# code

Problem no.	No. of student C# code solutions
Problem 1	108
Problem 2	105
Problem 3	113
Problem 4	202
Problem 5	104
Problem 6	102
Problem 7	104
Problem 8	105
Problem 9	100
Total	1,043

Table 5: GenAI C# code

Problem no.	No. of Blackbox.AI C# code solutions	No. of ChatGPT C# code solutions	No. of Copilot C# code solutions	Total categorised GenAI C# code solutions	Total uncategorised GenAI C# code solutions	Grand totals of C# code solutions
Problem 1	60	42	44	**146**	14	160
Problem 2	47	48	49	**144**	4	148
Problem 3	31	34	42	**107**	48	155
Problem 4	37	40	39	**116**	17	133
Problem 5	47	44	43	**134**	14	148
Problem 6	40	45	40	**125**	13	138
Problem 7	44	45	44	**133**	17	150
Problem 8	32	30	33	**95**	40	135
Problem 9	40	41	39	**120**	28	148
Total	378	369	373	**1,120**	195	1,315

Feature extraction

Modified Roslyn API[1] was used to extract the code metrics. For use of modified Roslyn API, the code must be written in visual code by creating a .NET console app with the addition of Microsoft.CodeAnalysis, Microsoft. CodeAnalysis.CSharp, and Microsoft.CodeAnalysis.CSharp.Syntax. The modified Roslyn code read the contents of the C# files into a string and then parsed the C# code into a syntax tree by using its syntax walk to analyse and extract features such as InterpolatedStringCount, StatementCount, MethodCount, ClassCount, VariableDeclarationCount, which are significant to code analysis as regards structure, syntax and semantics. The modified Roslyn API checked through the syntax tree to extract the code metric features, which are embedded into the modified Roslyn code. The extracted metrics were aggregated into a data structure for analysis, with the extracted features saved into an Excel file.

1 https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk

Eighty-three code stylometry features (Appendix A) extracted from the modified Roslyn API were used to train and evaluate the classifiers. The metrics fell into four categories, namely lexical, syntactic, layout, and semantic. Lexical features focus on individual tokens in the code such as keywords, identifiers, operators, and literals, reflecting the vocabulary and basic elements used. Extracted examples included UniqueIdentifiers, AverageIdentifierLength, and InterpolatedStringCount. Syntactic features capture the structural organisation of the code, including arrangement of statements, control flow constructs, and the syntax tree. The syntactic features extracted included IfStatementCount, MethodCount, NestedBlockDepth, and NamespaceCount. Layout features differentiate code based on formatting consistency, and the features extracted included NonWhitespaceLines, TotalLines, LineCount, and AverageLineLength. Semantic features capture the meaning or behaviour of the code, such as data flow, control flow, or implemented logic, and extracted features included MethodInvocationCount, CyclomaticComplexity, and ExpressionStatementCount.

CsvHelper and CsvHelper.Configuration were used to extract these code stylometry features into an Excel file for easy training and testing on the six classifier models. The command prompt was used to run the extraction command, with the directory set to the location of the modified Roslyn Visual Studio file. The "dotnet restore" command was run to check and read the .csproj in the project folder, and then the needed package from NuGet was downloaded into the solution package, and this command addressed any version conflicts. After this, the "dotnet build" command was used to compile the source code into a code that could be executed by .NET runtime and also checked for errors. Finally, the "dotnet run" command was used at the command prompt template, followed by a double quotation of the folder directory housing and saving the C# code files to extract the code stylometric. A confirmation message appeared in the command prompt, indicating the creation of the Excel file and successful writing of the code metrics.

Creation of four datasets

The experiment used 80% of the collected data for training and 20% for testing. All training was performed using group five-fold cross-validation with five splits: in each split, one fold was used for testing and the other four for training. Splitting ensured that no data point from any group appeared in both the training and testing sets. For all models, the number of estimators (n_estimators) was set manually to 100, and no hyperparameter search was performed. This default value provides a good balance between performance and computational cost.

Data was arranged into four sets:

- Set 1 comprised student-written code and a combination of Blackbox.AI, ChatGPT, and Copilot code.
- Set 2 comprised student-written code and Blackbox.AI code.
- Set 3 comprised student-written code and ChatGPT code.
- Set 4 comprised student-written code and Microsoft Copilot code.

After feature extraction, the data used ensured a balance between the student-written code and the GenAI code across each problem. The data used for the training and testing of each set included 1882, 756, 738, and 746 code solutions for Set 1, Set 2, Set 3, and Set 4, respectively. The training data for Set 1 is outlined in Table 6. The training data for Sets 2–4 is given in Table 7.

Table 6: Data in Set 1

Problem no.	**No. of student code solutions**	**No. of Blackbox.AI code solutions**	**No. of ChatGPT code solutions**	**No. of Copilot code solutions**	**Total no. of AI code solutions**	**Grand total**
Problem 1	108	**36**	36	36	**108**	**216**
Problem 2	105	**35**	35	35	**105**	**210**
Problem 3	107	**31**	34	42	**107**	**214**
Problem 4	116	**37**	40	39	**116**	**232**
Problem 5	104	**34**	35	35	**104**	**208**
Problem 6	102	**34**	34	34	**102**	**204**
Problem 7	104	**34**	35	35	**104**	**208**
Problem 8	95	**32**	30	33	**95**	**190**
Problem 9	100	**33**	33	34	**100**	**200**
Totals	**941**	306	312	323	**941**	**1,882**

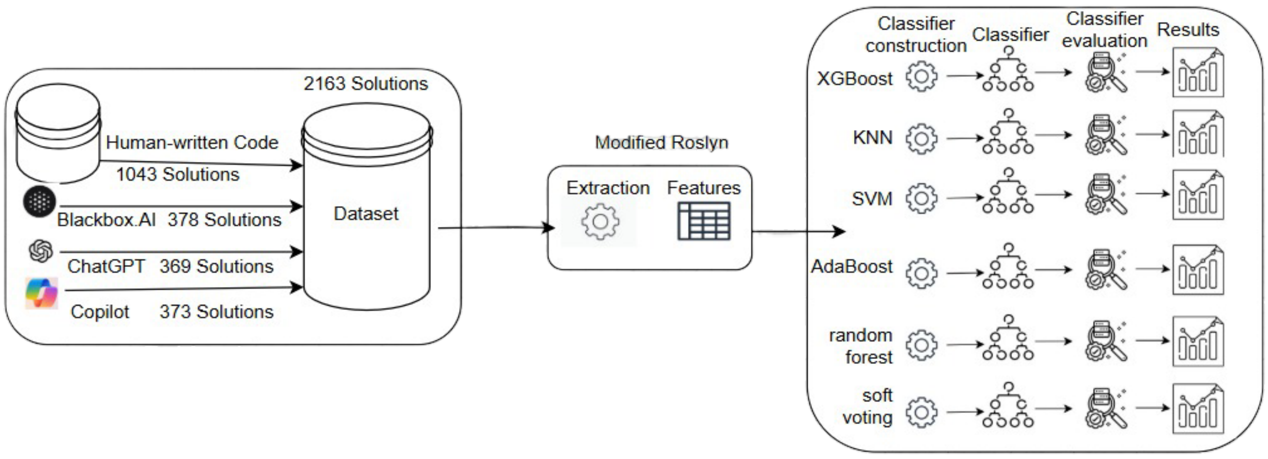
Table 7: Data in Sets 2–4

Problem no.	**Set 2 Student-written and Blackbox.AI code**			**Set 3 Student-written and ChatGPT code**			**Set 4 Student-written and Copilot code**		
	Student code	**Blackbox.AI code solutions**	**Total**	**Student code**	**ChatGPT code solutions**	**Total**	**Student code**	**Copilot code solutions**	**Total**
Problem 1	60	60	120	42	42	84	44	44	88
Problem 2	47	47	94	48	48	96	49	49	98
Problem 3	31	31	62	34	34	68	42	42	84
Problem 4	37	37	74	40	40	80	39	39	78
Problem 5	47	47	94	44	44	88	43	43	86
Problem 6	40	40	80	45	45	90	40	40	80
Problem 7	44	44	88	45	45	90	44	44	88
Problem 8	32	32	64	30	30	60	33	33	66
Problem 9	40	40	80	41	41	82	39	39	78
Totals	378	378	**756**	369	369	**738**	373	373	**746**

Testing of the classifier algorithms

In this study, no preprocessing or encoding was applied to the datasets prior to training the classifiers, because the 83 code stylometry features extracted using modified Roslyn were inherently numerical and continuous, thus representing quantitative properties of the code with no categorical variables (Appendix B). There were no missing values in the dataset, as all 83 features were successfully extracted. A machine-learning model (Figure 2) that sought to distinguish between GenAI C# code and student-written C# code was constructed, using the six aforementioned classifiers: XGBoost, KNN, SVM, AdaBoost, random forest, and soft voting (with XGBoost, KNN and SVM as inputs).

Figure 2: Model pipeline



Performance metrics

The performance of each classifier was measured using five metrics: accuracy, recall, precision, F1 score, and AUC-ROC (area under the curve-receiver operating characteristic).

Accuracy gives an overall measure of correctness and, to avoid giving misleading information, the datasets in this study were balanced with equal amounts of GenAI code and student-written code.

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN}$$

Where TP represents correctly identified GenAI code; TN represents correctly predicted human-written code; FP represents incorrectly classified GenAI code; and FN stands for incorrectly classified human-written code.

Recall measures the actual Gen AI code that is correctly identified, and a high recall indicates that GenAI code is rarely missed.

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

Precision measures the instances predicted as GenAI code that are actually GenAI. High precision indicates the low possibility of human-written code being classified and flagged as GenAI code.

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

F1 score is the harmonic mean of precision and recall, which checks the balance between detecting GenAI code and reducing the possibility of human-written code classified as GenAI code.

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

AUC-ROC valuates the model's ability to differentiate between GenAI and human-written code across different classification thresholds.

$$\text{AUC-ROC} = \int_0^1 TPR(FPR)d(FPR)$$

Where TPR is the true positive rate and the same as recall, and FPR is the false positive rate,

$$\text{FPR} = \frac{FP}{FP+TN}$$

4. Results and discussion

Set 1

In the test results for Set 1, which combined student-written code and code produced by all three LLMs (Blackbox.AI, ChatGPT, and Microsoft Copilot), it was found that random forest performed best for student-code detection with accuracy of 0.97, supported by recall of 0.97, precision of 0.88, and F1 score of 0.92 (Table 8). This strong performance indicates that random forest effectively identified nearly all the student-written code, with minimal false negatives (i.e., student code misclassified as AI-generated). For the AI-generated code, XGBoost performed best with accuracy of 0.93, recall of 0.89, precision of 0.96, and F1 score of 0.92. Both random forest and XGBoost achieved an AUC-ROC of 0.98, indicating strong discrimination between student and AI code across various thresholds.

Table 8: Classifier performance with Set 1 (student-written and AI code (from three LLMs))

Classifier	Accuracy		Recall		Precision		F1 score		AUC-ROC
	Student code	AI code	Student code	AI code	Student code	AI code	Student code	AI code	
XGBoost	**0.93**	**0.93**	0.89	0.89	**0.96**	**0.96**	**0.92**	**0.92**	**0.98**
KNN	0.78	0.70	0.78	0.70	0.75	0.76	0.75	0.70	0.81
SVM	0.79	0.75	0.79	0.75	0.81	0.83	0.77	0.75	0.92
AdaBoost	0.87	0.87	0.94	0.80	0.84	0.94	0.88	0.85	0.95
random forest	**0.97**	0.87	**0.97**	0.87	0.88	**0.97**	**0.92**	0.91	**0.98**
soft voting	0.90	0.90	0.87	**0.94**	0.93	0.87	0.90	0.90	0.97

Set 2

In the test results for Set 2, which combined student-written and Blackbox.AI-produced code, it was found that XGBoost and random forest performed best for student-code detection with accuracy of 0.92 (Table 9). Also, for the student code, XGBoost had recall of 0.88, precision of 0.95, and F1 score of 0.91, while random forest had recall of 0.92, precision of 0.88, and F1 score of 0.89. XGBoost's higher precision indicated fewer false positives, while random forest's higher recall suggested that it was slightly better at capturing all student code. For the Blackbox.AI-generated code, XGBoost was the best classifier, with accuracy of 0.92, recall of 0.88, precision of 0.95, and F1 score of 0.91. These metrics suggest that Blackbox.AI-generated code has distinct features that XGBoost effectively leverages. Both XGBoost and random forest achieved an AUC-ROC of 0.98, reinforcing their strong performance on this set.

Table 9: Classifier performance with Set 2 (student-written and Blackbox.AI code)

Model	Accuracy		Recall		Precision		F1 score		AUC-ROC
	Student code	AI code	Student code	AI code	Student code	AI code	Student code	AI code	
XGBoost	**0.92**	**0.92**	0.88	0.88	**0.95**	**0.95**	**0.91**	**0.91**	**0.98**
KNN	0.79	0.70	0.79	0.74	0.75	0.81	0.76	0.76	0.85
SVM	0.71	0.89	0.71	0.89	0.88	0.77	0.78	0.82	0.92
AdaBoost	0.89	0.89	**0.93**	0.86	0.88	0.93	0.90	0.89	0.96
random forest	**0.92**	0.86	**0.92**	0.86	0.88	0.92	0.89	0.88	**0.98**
soft voting	0.90	**0.90**	0.86	**0.94**	0.93	0.87	0.89	0.90	0.97

Set 3

In the test results for Set 3, which combined student-written and ChatGPT-produced code, it was found that random forest performed best for student-code detection with accuracy of 0.97, recall of 0.97, precision of 0.82, and an F1 score of 0.88 (Table 10). For detection of ChatGPT-generated code, AdaBoost was the best classifier with accuracy of 0.88, recall of 0.85, precision of 0.91, and F1 score of 0.87. The lower recall compared to other sets suggests that ChatGPT code is harder to detect, potentially due to student-like characteristics. XGBoost, AdaBoost, random forest, and soft voting all achieved an AUC-ROC of 0.97, indicating robust class separation despite the challenges posed by ChatGPT code.

Table 10: Classifier performance with Set 3 (student-written and ChatGPT code)

Model	Accuracy		Recall		Precision		F1 score		AUC-ROC
	Student code	AI code	Student code	AI code	Student code	AI code	Student code	AI code	
XGBoost	0.86	0.86	0.79	0.79	**0.93**	0.93	0.85	0.85	**0.97**
KNN	0.82	0.68	0.82	0.68	0.75	0.78	0.77	0.71	0.81
SVM	0.62	0.91	0.62	0.91	0.89	0.72	0.71	0.80	0.87
AdaBoost	0.88	**0.88**	0.90	0.85	0.88	0.91	**0.88**	**0.87**	**0.97**
random forest	**0.97**	0.74	**0.97**	0.74	0.82	**0.97**	**0.88**	0.82	**0.97**
soft voting	0.87	0.87	0.80	**0.93**	0.92	0.83	0.86	**0.87**	**0.97**

Set 4

In the test results for Set 4, which combined student-written and Microsoft Copilot-produced code, it was found that random forest performed best for student code detection with accuracy of 0.97, recall of 0.97, precision of 0.84, and F1 score of 0.90 (Table 11). For the Copilot-generated code, the soft voting classifier performed best with accuracy of 0.90, recall of 0.96, precision of 0.86, and F1 score of 0.90. Random forest achieved the highest AUC-ROC of 0.99, followed by XGBoost with 0.98 and AdaBoost and soft voting with 0.96.

Table 11: Classifier performance with Set 4 (student-written and Copilot code)

Model	Accuracy		Recall		Precision		F1 score		AUC-ROC
	Student code	AI code	Student code	AI code	Student code	AI code	Student code	AI code	
XGBoost	0.89	**0.89**	0.80	0.80	**0.97**	**0.97**	0.88	0.88	0.98
KNN	0.86	0.70	0.86	0.70	0.77	0.82	0.80	0.72	0.82
SVM	0.72	0.86	0.72	0.86	0.88	0.78	0.75	0.80	0.91
AdaBoost	0.88	0.88	**0.97**	0.79	0.85	0.96	**0.90**	0.85	0.96
random forest	**0.97**	0.79	**0.97**	0.79	0.84	**0.97**	**0.90**	0.86	**0.99**
soft voting	0.90	**0.90**	0.84	**0.96**	0.95	0.86	0.89	**0.90**	0.96

Results across the four sets

The results across the four sets indicated that with respect to the AI-generated code, the Blackbox.AI code was the easiest to detect, as demonstrated by the high accuracies in Set 2. ChatGPT (Set 3) and Copilot (Set 4) were more challenging, with lower detection accuracies for AI-generated code. This suggests that Blackbox.AI produces code with stylistic or structural features that are more distinct than

ChatGPT and Copilot when compared to student code. ChatGPT and Copilot apparently generate code that more closely mimics human patterns, possibly due to their advanced language-modelling capabilities. XGBoost dominated AI code detection in Sets 1 and 2 (accuracies of 0.93 and 0.92), while AdaBoost and voting classifier were better suited to Sets 3 and 4 (accuracies of 0.88 and 0.90), respectively. XGBoost's optimisation of decision trees appears to make it more attuned to optimising patterns in AI-generated code, as suggested by the results from Sets 1 and 2. Also notable was the fact that for detection of AI-generated code, the use of the soft voting classifier, which integrates inputs from three classifiers, markedly improved the recall rate.

The results also indicated that the student-written C# code was generally easier to detect than the GenAI C# code, as most classifiers demonstrated superior or equivalent accuracy in identifying the student-written code. This finding may reflect greater variability in student coding styles, making them easier to distinguish from the more uniform AI-generated code. Random forest consistently excelled in student-code detection across Sets 1, 3, and 4, and tied with XGBoost in Set 2, showing its robustness for identifying student code. Random forest's superior performance can be attributed to its algorithmic strength, which reduces overfitting and enhances generalisation, making it well suited to capturing diverse patterns in student-written code.

Feature analysis

SHAP was used to further explain the outputs of the machine-learning models, through assigning an importance value to each feature for prediction, i.e., showing the features that were most influential in identifying and classifying the AI-generated code and the student-written code.

For Set 1, the five most important features were InterpolatedStringCount, StatementCount, NamespaceCount, LineCount, and TotalLines. For Set 2, the five most important features are InterpolatedStringCount, LineCount, TotalLines, StatementCount, and ExpressionStatementCount. For Set 3, the five most important features were InterpolatedStringCount, NamespaceCount, StatementCount, NonWhitespaceLines, and ExpressionStatementCount. For Set 4, the five most important features were InterpolatedStringCount, NamespaceCount, StatementCount, NonWhitespaceLines, and ExpressionStatementCount.

InterpolatedStringCount is the number of interpolated strings such as $"Hello, {name}" in the C# code. GenAI code tends to use more of these strings for dynamic values, while human-written code tends to have more concatenation methods. StatementCount is the number of statements in the C# code. GenAI code uses more dense lines of code than the corresponding human-written code, showing structural differences. NamespaceCount is the number of namespaces used in the C# code. GenAI code tends to use a limited number of namespaces and to use an optimised relevant one, while human-written code tends to include unnecessary and even unused namespaces, thus having more namespaces than GenAI code.

NonWhitespaceLines are the lines in the C# code that contain actual code or comment. GenAI code tends to have more NonWhitespaceLines than human-written code. TotalLines measures the overall length of C# code. GenAI code tends to have fewer lines of code than its human-written counterpart. ExpressionStatementCount is the number of expression statements in the C# code, e.g., expressions such as assignments, compound, type casting, function return, and conditional statements. GenAI code tends to have more such statements than human-written code because GenAI code seeks to explicitly state each operation for clarity. LineCount is the total number of lines in the C# code, and GenAI code tends to use fewer lines than human-written code.

As seen in the SHAP feature-importance graphs below for Sets 1 and 2 (Figure 3) and Sets 3 and 4 (Figure 4), the most important features across the four sets were InterpolatedStringCount and StatementCount, which both appear in the top five features for each set.

Figure 3: SHAP feature importance for Sets 1 and 2

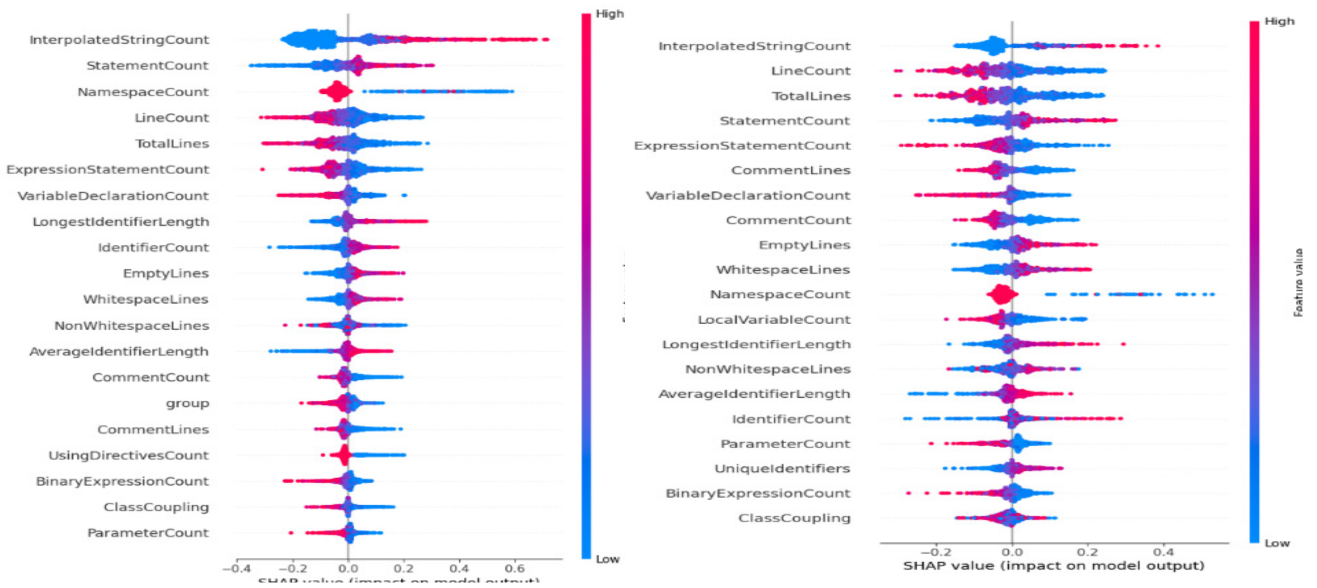
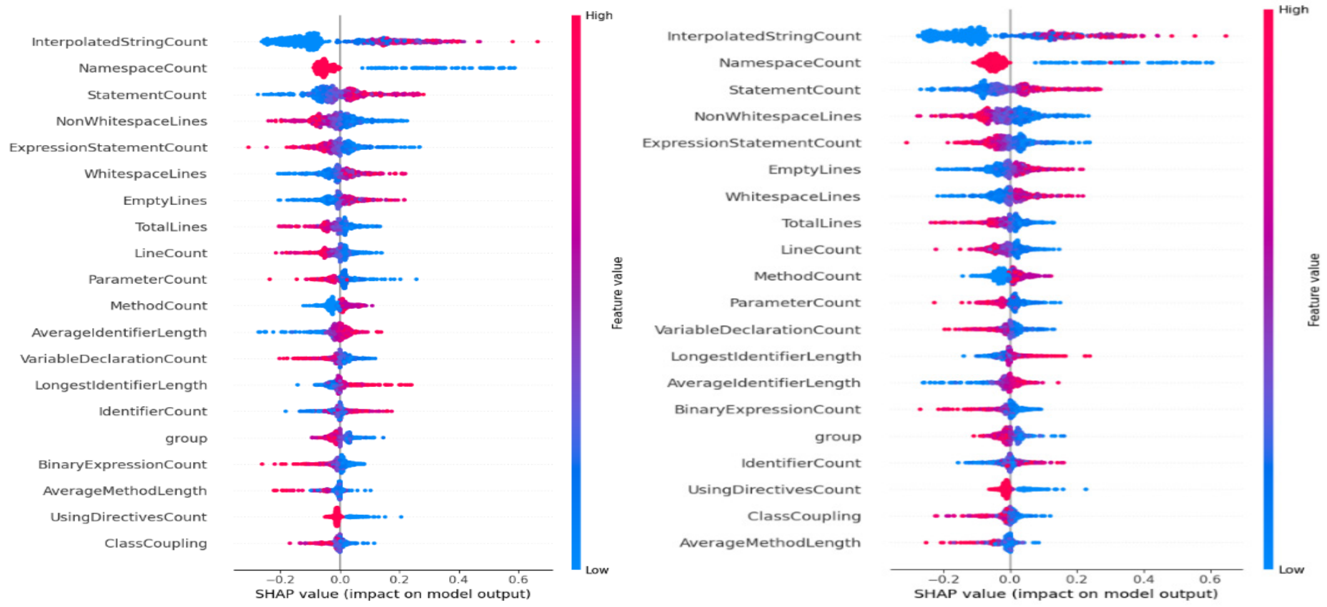


Figure 4: SHAP feature importance for Sets 3 and 4



Comparison with other similar studies

Table 12 compares the classifier accuracy, for detection of GenAI code, that was found in this C#-based study with accuracies detected in similar studies focused on different programming languages. As seen in the table, the approach in this study, which leveraged a comprehensive feature set and advanced classifiers, achieved higher accuracies than the models used in the Bukhari et al. (2023), Idialu et al. (2024), and Pan et al. (2024) studies of GenAI code detection in the Python and C programming languages.

Table 12: Comparison of GenAI code detection accuracy

Study	Programming language	GenAI model(s) used	Classifier(s) used	Classifier accuracy
This study: Adegbite and Kotzé (2025)	C#	Blackbox.AI ChatGPT Microsoft Copilot	XGBoost KNN SVM AdaBoost random forest soft voting	0.97 0.82 0.89 0.96 0.97 0.95 (highest accuracy among the accuracy figures for the 4 sets)
Bukhari et al. (2023)	C	OpenAI code-cushman-001, code-davinci-001, and code-davinci-002	XGBoost KNN SVM random forest	0.92 0.73 0.85 0.90
Idialu et al. (2024)	Python	ChatGPT-4	XGBoost	0.89
Pan et al. (2024)	Python	ChatGPT	GPT Zero GPT-2 Detector DetectGPT GLTR Sapling	0.49 0.50 0.48 0.50 0.60

The feature extraction in the Bukhari et al. (2023) study includes only lexical and syntactic features, while Idialu et al. (2024) add the layout features to the two features considered by Bukhari et al. (2023). Our study focused on a larger list of code stylometry features (comprising 83 lexical, syntactic, layout, and semantic features) than those included by Idialu et al. (2024) and Bukhari et al. (2023), and we can conclude that this wider range of features was integral to the higher classifier accuracy achieved in our study.

5. Conclusions

This study has demonstrated that GenAI C# code produced by Blackbox.AI, ChatGPT, and Copilot can, to a great extent, be identified, and distinguished from student-written C# code, through use of classifier algorithms. The random forest and XGBoost classifiers performed best, with Blackbox.AI C# code being the easiest to detect. This study's focus on the C# programming language helps to fill a research gap, as GenAI code detection in C# is a relatively unexplored area in the education sector in South Africa and globally. This study is also significant in several other respects.

Implications for educators

The study findings also have the potential to assist educational institutions and educators in developing tools for detection of potential use of GenAI code in student assignments. Student use of AI tools for programming and software course assignments can be expected to decrease if detection systems are in place, which in turn will help maintain adherence to academic standards. SHAP identification of features in student assignments can also help to reveal patterns in students' coding behaviours, enabling targeted interventions to improve foundational programming skills. Students can also be encouraged to critically evaluate the strengths and limitations of GenAI code, which will improve their critical thinking skills.

Implications for researchers

Through its use of a large list of code stylometry features (comprising 83 lexical, syntactic, layout, and semantic features), this study has highlighted certain features that were particularly important to the detection of GenAI C# code and to distinguishing between AI-generated and student-written code. Researcher identification of more important features can increase the optimisation of GenAI detection algorithms for programming tasks across varying coding styles and structures. Interdisciplinary studies can follow from this research, as GenAI code detection is at an intersection of NLP, cybersecurity, software engineering, ethics, and education. Future research could incorporate broader sets of coding problems, and broader sources of human-written C# code. The code could be sourced from different educational levels or institutions, as well as from professional developers, to improve the generalisability of the results.

Implications for software developers

Improved detection of GenAI code can help software developers to understand the structure, style and logic of AI-generated code contributions to software. With improved detection and understanding of GenAI code, developers can more easily collaborate in an environment that allows for contributions from both GenAI tools and human developers. Developers can focus more on refining AI contributions, while preserving the nuances of human creativity. Enhanced detection of GenAI code can also help developers to strengthen application security and cybersecurity, particularly with respect to malicious actors who use GenAI to produce scripts used in attacks. When GenAI code is detected early, pre-emptive measures can be put in place to reduce vulnerabilities and safeguard systems against evolving threats.

Limitations of the study

A limitation of this study was that the human-written code dataset was collated from first-year programming students from only one campus of one university: the University of the Free State, South Africa. Thus, this code does not represent the diversity of human-written C# code, which limits the generalisability of the study findings. A larger, more diverse dataset would have provided a better representation of human-written C# code. Furthermore, the nine problems in terms of which the human-written and GenAI C# code was prepared presented potential limitations. The problems could have imposed biases and are unlikely to have fully captured the nuances and complexities of software development, e.g., matters of performance optimisation, security vulnerabilities, maintainability, and real-world applicability.

Funding declaration
No funding was received for this study or for preparation of the article.

Data availability
Data will be made available upon request to the first-listed author at adewuyi.adegbite@gmail.com.

AI declaration
GenAI tools were used for data collection of GenAI C# code, with the versions stated in the article.

Competing interests declaration
The authors have no competing interests to declare.

Authors' contributions
A.A.A.: Conceptualisation; methodology; data collection; sample analysis; data analysis; validation; data curation; writing – initial draft; writing – revisions; student supervision; project management.
E.K.: Conceptualisation; methodology; data collection; writing – revisions; student supervision; project leadership; project management; funding acquisition.

References

Benzebouchi, N. E., Azizi, N., Hammami, N. E., Schwab, D., Khelaifia, M. C. E., & Aldwairi, M. (2019). Authors' writing styles based authorship identification system using the text representation vector. In *16th International Multi-Conference on Systems, Signals and Devices (SSD 2019)* (pp. 371–376). https://doi.org/10.1109/SSD.2019.8894872

Bukhari, S., Tan, B., & De Carli, L. (2023). Distinguishing AI- and human-generated code: A case study. In *SCORED 2023 – Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses* (pp. 17–25). https://doi.org/10.1145/3605770.3625215

Caliskan, A., Yamaguchi, F., Dauber, E., Harang, R., Rieck, K., Greenstadt, R., & Narayanan, A. (2018). When coding style survives compilation: De-anonymizing programmers from executable binaries. In *25th Annual Network and Distributed System Security Symposium (NDSS 2018)*. https://doi.org/10.14722/ndss.2018.23304

Cao, Y., Li, S., Liu, Y., Yan, Z., Dai, Y., Yu, P. S., & Sun, L. (2023). A comprehensive survey of AI-generated content (AIGC): A history of generative AI from GAN to ChatGPT. *Journal of the ACM*, *37*(4). http://arxiv.org/abs/2303.04226

Cheers, H., Lin, Y., & Smith, S. P. (2021). Academic source code plagiarism detection by measuring program behavioral similarity. *IEEE Access*, *9*, 50391–50412. https://doi.org/10.1109/ACCESS.2021.3069367

Cheers, H., Lin, Y., & Yan, W. (2023). Identifying plagiarised programming assignments with detection tool consensus. *Informatics in Education*, *22*(1), 1–19. https://doi.org/10.15388/infedu.2023.05

Corso, V., Mariani, L., Micucci, D., & Riganelli, O. (2024). Generating Java methods: An empirical assessment of four AI-based code assistants. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension (ICPC 2024)*. https://doi.org/10.1145/3643916.3644402

Czibula, G., Lupea, M., & Briciu, A. (2022). Enhancing the performance of software authorship attribution using an ensemble of deep autoencoders. *Mathematics*, *10*(15). https://doi.org/10.3390/math10152572

Dehaerne, E., Dey, B., Halder, S., De Gendt, S., & Meert, W. (2022). Code generation using machine learning: A systematic review. *IEEE Access*, *10*(July), 82434–82455. https://doi.org/10.1109/ACCESS.2022.3196347

Ding, S. H. H., Fung, B. C. M., Iqbal, F., & Cheung, W. K. (2019). Learning stylometric representations for authorship analysis. *IEEE Transactions on Cybernetics*, *49*(1), 107–121. https://doi.org/10.1109/TCYB.2017.2766189

Ebrahim, F., & Joy, M. (2023). Source code plagiarism detection with pre-trained model embeddings and automated machine learning. In *International Conference Recent Advances in Natural Language Processing (RANLP)* (pp. 301–309). https://doi.org/10.26615/978-954-452-092-2_034

Eliwa, E., Essam, S., Ashraf, M., & Sayed, A. (2023). Automated detection approaches for source code plagiarism in students' submissions. *Journal of Computing and Communication*, *2*(2), 8–18. https://doi.org/10.21608/jocc.2023.307054

Ghosal, S. S., Chakraborty, S., Geiping, J., Huang, F., Manocha, D., & Bedi, A. S. (2023). Towards possibilities and impossibilities of AI-generated text detection: A survey. arXiv preprint. https://doi.org/10.48550/arXiv.2310.15264

Idialu, O. J., Mathews, N. S., Maipradit, R., Atlee, J. M., & Nagappan, M. (2024). Whodunit: Classifying code as human authored or GPT-4 generated – A case study on CodeChef problems. https://doi.org/10.1145/3643991.3644926

Kalgutkar, V., Kaur, R., Gonzalez, H., Stakhanova, N., & Matyukhina, A. (2019). Code authorship attribution: Methods and challenges. *ACM Computing Surveys*, *52*(1). https://doi.org/10.1145/3292577

Kazemitabaar, M., Ye, R., Wang, X., Henley, A. Z., Denny, P., Craig, M., & Grossman, T. (2024). CodeAid: Evaluating a classroom deployment of an LLM-based programming assistant that balances student and educator needs. In *CHI '24: Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems.* https://doi.org/10.1145/3613904.3642773

Kotsiantis, S., Verykios, V., & Tzagarakis, M. (2024). AI-assisted programming tasks using code embeddings and transformers. *Electronics*, *13*(4), 1–25. https://doi.org/10.3390/electronics13040767

Krasniqi, R., & Do, H. (2023). Towards semantically enhanced detection of emerging quality-related concerns in source code. *Software Quality Journal*, *31*(3), 865–915. https://doi.org/10.1007/s11219-023-09614-8

Kuhail A. M., Mathew, S. S., Khalil, A., Berengueres, J., Jawad, S., & Shah, H. (2024). "Will I be replaced?" Assessing ChatGPT's effect on software development and programmer perceptions of AI tools. *Science of Computer Programming*, *235*, 103111. https://doi.org/10.1016/j.scico.2024.103111

Lalitha, L. V. K., Sree, V., Lekha, R. S., & Kumar, V. N. (2021). Plagiat: A code plagiarism detection tool*. EPRA International Journal of Research and Development (IJRD)*, *7838*, 97–101.

Li, Z., Jiang, Y., Zhang, X. J., & Xu, H. Y. (2020). The metric for automatic code generation. *Procedia Computer Science*, *166*, 279–286. https://doi.org/10.1016/j.procs.2020.02.099

Lundberg, S. M., & Lee, S.-I. (2017). A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems 30 (NIPS 2017)*. https://arxiv.org/abs/1705.07874

Makridakis, S. (2017). The forthcoming Artificial Intelligence (AI) revolution: Its impact on society and firms. *Futures*, *90*, 46–60. https://doi.org/10.1016/j.futures.2017.03.006

Maryono, D., Yuana, R. A., & Hatta, P. (2019). The analysis of source code plagiarism in basic programming course. *Journal of Physics: Conference Series*, *1193*(1). https://doi.org/10.1088/1742-6596/1193/1/012027

Nghiem, K., Nguyen, A. M., & Bui, N. D. Q. (2024). Envisioning the next-generation AI coding assistants: Insights and proposals. In *2024 First IDE Workshop (IDE '24)*. https://doi.org/10.1145/3643796.3648467

Odeh, A., Odeh, N., & Mohammed, A. S. (2024). A comparative review of AI techniques for automated code generation in software development: Advancements, challenges, and future directions. *TEM Journal*, *13*(1), 726–739. https://doi.org/10.18421/tem131-76

Pan, W. H., Chok, M. J., Wong, J. L. S., Shin, Y. X., Poon, Y. S., Yang, Z., Chong, C. Y., Lo, D., & Lim, M. K. (2024). Assessing AI detectors in identifying AI-generated code: Implications for education. https://arxiv.org/abs/2401.03676

Portillo-Dominguez, A. O, Ayala-Rivera, V., Murphy, E., & Murphy, J. (2017). A unified approach to automate the usage of plagiarism detection tools in programming courses. In *ICCSE 2017 – 12th International Conference on Computer Science and Education*, *ICCSE*, 18–23. https://doi.org/10.1109/ICCSE.2017.8085456

Raiaan, M. A. K., Mukta, M. S. H., Fatema, K., Fahad, N. M., Sakib, S., Mim, M. M. J., Ahmad, J., Ali, M. E., & Azam, S. (2024). A review on large language models: Architectures, applications, taxonomies, open issues and challenges. *IEEE Access*, *12*(February), 26839–26874. https://doi.org/10.1109/ACCESS.2024.3365742

ShaukatTamboli, M., & Prasad, R. (2013). Authorship analysis and identification techniques: A review. *International Journal of Computer Applications*, *77*(16), 11–15. https://doi.org/10.5120/13566-1375

Song, X., Sun, H., Wang, X., & Yan, J. (2019). A survey of automatic generation of source code comments: Algorithms and techniques. *IEEE Access*, *7*, 111411–111428. https://doi.org/10.1109/ACCESS.2019.2931579

Srivastava, S., Rai, A., & Varshney, M. (2021). A tool to detect plagiarism in java source code. *Lecture Notes in Networks and Systems*, *145*, 243–253. https://doi.org/10.1007/978-981-15-7345-3_20

Tereszkowski-Kaminski, M., Pastrana, S., Blasco, J., & Suarez-Tangil, G. (2022). Towards improving code stylometry analysis in underground forums. In *Proceedings on Privacy Enhancing Technologies*, *2022*(1), 126–147. https://doi.org/10.2478/popets-2022-0007

Varona, D., & Suárez, J. L. (2022). Discrimination, bias, fairness, and trustworthy AI. *Applied Sciences*, *12*(12). https://doi.org/10.3390/app12125826

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Illia, P. (2017). Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems* (pp. 5998–6008). https://doi.org/10.48550/arXiv.1706.03762

Wan, Y., He, Y., Bi, Z., Zhang, J., Zhang, H., Sui, Y., Xu, G., Jin, H., & Yu, P. S. (2023). Deep learning for code intelligence: Survey, benchmark and toolkit. *Arxiv.Org*, *1*(1), 771–783. https://arxiv.org/abs/2401.00288

White, J., Hays, S., Fu, Q., Spencer-Smith, J., & Schmidt, D. C. (2023). ChatGPT prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. https://doi.org/10.1007/978-3-031-55642-5_4

Zafar, S., Sarwar, M. U., Salem, S., & Malik, M. Z. (2020). Language and obfuscation oblivious source code authorship attribution. *IEEE Access*, *8*, 197581–197596. https://doi.org/10.1109/ACCESS.2020.3034932

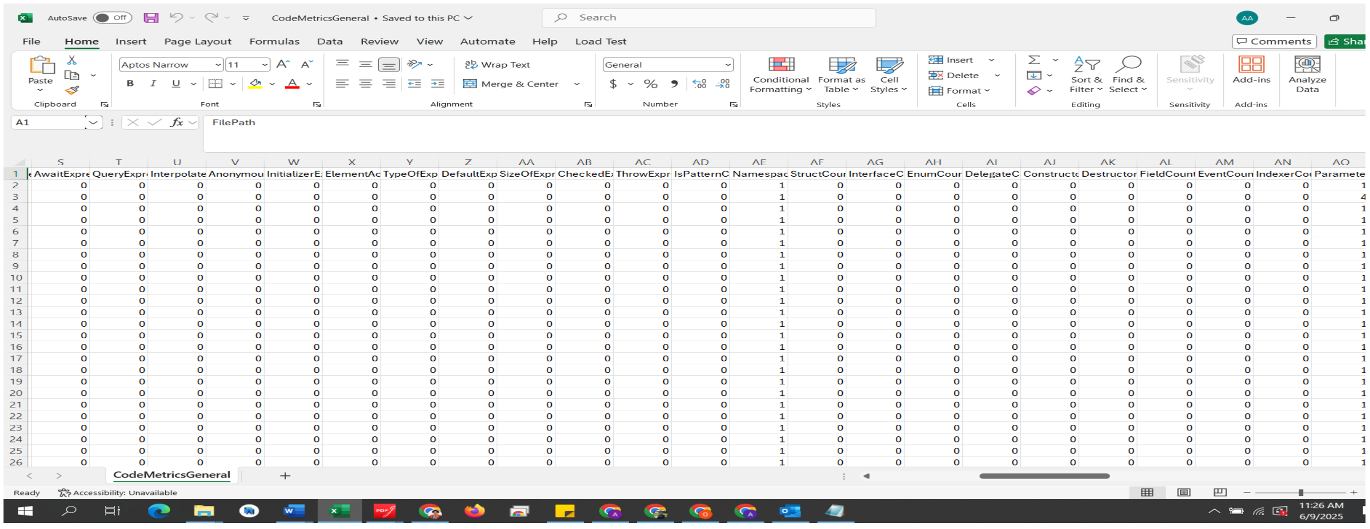
Zhang, H., Cruz, L., & van Deursen, A. (2022). Code smells for machine learning applications. In *Proceedings – 1st International Conference on AI Engineering – Software Engineering for AI (CAIN) 2022* (pp. 217–228). https://doi.org/10.1145/3522664.3528620

Zheng, M., Pan, X., & Lillis, D. (2018). CodEX: Source code plagiarism detection based on abstract syntax trees. *CEUR Workshop Proceedings*, *2259*, 362–373.

Appendix A: The 83 code stylometry features extracted using modified Roslyn

FilePath	UniqueIdentifiers	IfStatementCount	EnumCount
UsingDirectivesCount	StatementCount	AnonymousMethodCount	EventCount
FixedStatementCount	CommentCount	WhileLoopCount	LCOM
UsingStatementCount	MethodInvocationCount	QueryExpressionCount	FieldCount
SwitchStatementCount	ShortestIdentifierLength	AwaitExpressionCount	ClassCount
VariableDeclarationCount	AverageIdentifierLength	DefaultSwitchLabelCount	LineCount
InterpolatedStringCount	AverageMethodLength	LockStatementCount	UsesSpaces
	InitializerExpressionCount	ElementAccessCount	UsesTabs
TypeOfExpressionCount	DefaultExpressionCount	SizeOfExpressionCount	StructCount
CheckedExpressionCount	ThrowExpressionCount	IsPatternCount	TotalLines
NamespaceCount	InterfaceCount	ForEachLoopCount	EmptyLines
DelegateCount	ConstructorCount	YieldBreakCount	ClassCoupling
DestructorCount	ReturnStatementCount	YieldReturnCount	MethodCount
LongestIdentifierLength	ParameterCount	ElseClauseCount	PropertyCount
LocalVariableCount	CyclomaticComplexity	AfferentCoupling	AttributeCount
NestedBlockDepth	DepthOfInheritance	EfferentCoupling	LambdaCount
NonWhitespaceLines	MaxMethodBlockDepth	CaseSwitchLabelCount	TernaryCount
WhitespaceLines	MaxNestedBlockDepth	DoWhileLoopCount	IndexerCount
CommentLines	AverageLineLength	ExpressionStatementCount	ForLoopCount
MinLineLength	MaxLineLength	ObjectCreationCount	IdentifierCount
AssignmentCount	BinaryExpressionCount	LocalFunctionCount	

Appendix B: Snapshot of dataset extracted from modified Roslyn


AutoSave Off
CodeMetricsGeneral • Saved to this PC
Search
File Home Insert Page Layout Formulas Data Review View Automate Help Load Test
Comments
Paste Clipboard
Aptos Narrow 11 Font
Wrap Text Merge & Center Alignment
General Number
Conditional Formatting Format as Table Cell Styles Styles
Insert Delete Format Cells
Sort & Filter Find & Select Editing
Sensitivity
Add-ins
Analyze Data
CodeMetricsGeneral
Ready
Accessibility: Unavailable
11:26 AM
6/9/2025

A1 FilePath

	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE	AF	AG	AH	AI	AJ	AK	AL	AM	AN	AO
1	AwaitExpre	QueryExpr	Interpolate	Anonymou	InitializerE	ElementAc	TypeOfExp	DefaultExp	SizeOfExpr	CheckedE	ThrowExpr	IsPatternC	Namespac	StructCour	InterfaceC	EnumCour	DelegateC	Constructo	Destructor	FieldCount	EventCoun	IndexerCo	Paramete
2	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
3	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	4
4	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
5	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
11	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
13	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
14	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
15	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
16	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
17	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
18	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
19	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
20	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
21	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
22	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
23	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
24	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
25	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
26	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1

Appendix C: The nine problems used

1.	Develop a C# console application to generate an invoice for the CSI Hoodies company. Collect customer name and full address (street, city, province, postal code). Accept the number of hoodies ordered (whole number). Calculate the total due, including a hardcoded 15% VAT, with a hoodie price of R230. Display a formatted invoice using string.Format() for the address and properly formatted currency values. Clear the console before showing the invoice.
2.	Create a C# program to decide if a car should be sold based on its age and mileage. Input the car's model year and odometer reading (in kilometers) as integers. Sell the car if: Odometer exceeds 100,000 km (regardless of age). Model year is before 2014 (older than 10 years) and after 1950 (not antique). Do not sell if the car is an antique (1950 or earlier) or less than 10 years old (2014 or later). Use one Console.WriteLine() per outcome with newline and tab escape characters, avoiding compound conditions or logical operators.
3.	Build a C# console application named StudentGrades to compute a student's average mark and grade level. Display a title and prompt for three test marks. Calculate the average in one statement, handling integer division. Assign a grade based on the average: A: 80-100 B: 70-79 C: 60-69 D: 50-59 E: Below 50 Use a single Console.WriteLine() to show the result, building a general string and appending the grade dynamically. Add comments to separate code sections.
4.	Write a C# program to find the highest common factor (HCF) of two integers. Accept two positive integers as input. Use a while loop to calculate the HCF by dividing the larger number by the smaller one, updating values with the remainder until it reaches 0; the last non-zero remainder is the HCF. No error checking is required.
5.	Develop a C# console application for a café ordering system. Display a menu of meal items, each with an associated number. Allow the user to select a meal by entering its number and specify the quantity. Display order details: number of meals, price per meal, total price (formatted as currency), and a thank you message. Use a do-while loop to handle multiple orders; exit the program when the user enters -1. Implement three custom static methods: GetInt: Takes a string prompt, displays it, reads user input, and returns it as an integer. TotalPrice: Takes quantity and unit price as parameters, returns the total cost as a decimal. GenerateOrder: Takes a meal price, prompts for quantity using GetInt, calculates the total using TotalPrice, and displays the order details. Use a try-catch block to handle invalid inputs, showing an error message (using an Exception property) and a prompt to retry. Use a switch-case structure to set the price based on the selected meal number and call GenerateOrder.

6.	Create a C# console application named MultiplicationTable to generate multiplication tables. Prompt the user to enter a whole number to specify the multiplication table. Generate and display the table up to the 12th place (e.g., 1 × n to 12 × n) using a while loop. Use string.Format() for aligned output, starting from 1 (not 0). After each table, ask if the user wants to generate another (Y/N), using a do-while loop to repeat the process. Use a char variable for Y/N input and handle both uppercase and lowercase (e.g., with ToUpper() or ToLower()). Exit the program when the user enters 'N.' Use a try-catch block to handle invalid inputs, displaying a custom error message.
7.	Develop a C# console application for a café ordering system. Display a menu of meal items with numbers. Allow the user to select a meal by number and specify the quantity. Display order details: number of meals, price per meal, total price (formatted as currency), and a thank you message. Use a do-while loop to handle multiple orders; exit on -1. Implement custom methods: GetInt, TotalPrice, and GenerateOrder (same as UFSCSI 051). Handle invalid inputs with a try-catch block, showing an error message and retry prompt. Use a switch-case to assign prices and call GenerateOrder.
8.	Develop a C# console application named CompositionOfMoney to break down a monetary amount into the smallest number of coins/notes. Accept and validate a decimal amount using the GetDecimal method (returns a bool and the amount). Convert the amount to cents and use the DisplayUnits method to display the breakdown into units: 1c, 5c, 50c, R1, R10, R100. Loop for multiple conversions using the isAnotherOne method to control repetition. Handle invalid inputs without try-catch.
9.	Create a C# console application named Revision for basic mathematical operations. Show a menu with options: Addition (+), Subtraction (-), Multiplication (*), Division (/). Use a do-while loop to ensure valid operation selection (no if-else for selection). Implement methods for each operation: Addition(): Sum multiple numbers with a while loop. Subtraction(): Subtract two numbers using compound assignment. Multiplication(): Multiply two numbers. Division(): Divide two numbers, handling division by zero with a red error message. Validate numerical inputs with a custom method.