30 April 2025, Preprint v1 · CC-BY 4.0

Research Article

An Empirical Study on Automatically Detecting AI-Generated Source Code: How Far Are We?

Hyunjae Suh[1], **Mahan Tafreshipour**[1], **Jiawei Li**[1], **Adithya Bhattiprolu**[1], **Iftekhar Ahmed**[1]

1. University of California, Irvine, United States

Artificial Intelligence (AI) techniques, especially Large Language Models (LLMs), have started gaining popularity among researchers and software developers for generating source code. However, LLMs have been shown to generate code with quality issues and also incurred copyright/licensing infringements. Therefore, detecting whether a piece of source code is written by humans or AI has become necessary. This study first presents an empirical analysis to investigate the effectiveness of the existing AI detection tools in detecting AI-generated code. The results show that they all perform poorly and lack sufficient generalizability to be practically deployed. Then, to improve the performance of AI-generated code detection, we propose a range of approaches, including fine-tuning the LLMs and machine learning-based classification with static code metrics or code embedding generated from Abstract Syntax Tree (AST). Our best model outperforms state-of-the-art AI-generated code detector (GPTSniffer) and achieves an F1 score of 82.55. We also conduct an ablation study on our best-performing model to investigate the impact of different source code features on its performance.

Correspondence: papers@team.qeios.com — Qeios will forward to the authors

I. Introduction

Artificial Intelligence (AI), such as machine learning techniques, has been widely used to tackle software development tasks, especially for the generation of source code [1][2][3][4]. More recently, Large Language Models (LLMs) that were pre-trained on large and diverse data corpora have shown state-of-the-art performance in code generation [5][6][7][8][9][10]. The generative LLMs such as ChatGPT [11], Gemini Pro [12] and **Starcoder2** [13] are able to generate code that is very similar to what a human developer would produce given the natural language specification. While a large amount of previous research works have explored a variety of fine-tuning/prompting techniques [14][15] to further boost model's performance in generating high-quality source code, numerous LLM-based tools (i.e., Github Copilot [16]) have been implemented to assist developers to design software architecture, generate production code/test cases, and refactor the existing code base. Therefore, leveraging LLMs for source code generation or assisting programming-related tasks is becoming popular among software practitioners.

However, the wide adoption of LLMs for code generation has raised a variety of concerns among researchers and software practitioners. Researchers have questioned the evaluation process of the source code quality generated by LLMs [17], and the correctnesss of the generated code could be easily impacted by the wording in the LLMs' prompts [18]. In addition, it's been proven that around 35% of Github Copilot-generated code snippets on Github have security issues of various types [19][20], indicating the security risks of the code generated by LLMs. Moreover, the violations of intellectual property rights have also been found in LLMs such as generation of licensed code [21].

Therefore, it has become necessary to determine whether a code snippet is written by humans or generated by the LLMs. While there exist a variety of automated tools (i.e., GPTZero [22], Sapling [23], and more) to detect Artificial Intelligence Generated Content (AIGC), such tools were built for detecting natural language texts and their performance in detecting AI-generated source code still remains far from being perfect [24][25]. To fill this gap, Nguyen et al. [24] proposed GPTSniffer by fine-tuning CodeBERT [26] to classify a code snippet as either human-written or LLM-generated. However, they only considered the code that is written in Java programming language and is generated only by ChatGPT. It has been proven that different LLMs are good at generating code for different sets of coding problems [10]; namely, they tend to perform differently given the same set of coding tasks. Thus, GPTSniffer's generalizability to code that's written in other programming languages or generated by LLMs other than ChatGPT is not investigated. A better detection approach for detecting AI-generated source code is still missing.

In this paper, we first conduct a comprehensive empirical study to evaluate the performance of existing AIGC detectors for detecting AI-generated source code. The goal was twofold: first, as a complement to prior studies (see, e.g., [24][25]), we investigate the widely-adopted AIGC detectors' ability to detect AI-generated source code that is written in various programming languages, generated by multiple popular generative LLMs, and from different domains (i.e., programming questions, open-source development tasks). Second, we analyzed the performance of the current state-of-the-art detector specifically for source code, GPTSniffer. Therefore, we asked the following research questions:

- **RQ1: How do existing AIGC detectors perform on detecting AI-generated source code?**
- **RQ2: How can we improve the performance of AI-generated source code detection?**
- **RQ3: How do the source code features captured by embeddings contribute to the overall effectiveness?**

The significance of our contributions are following:

- We show that existing AIGC detectors for text perform poorly in detecting AI-generated source code.
- We show that the current state-of-the-art technique for AI-generated code detection, GPTSniffer, fails to generalize effectively across different programming languages, programming tasks, and generative LLMs.
- We built a variety of machine learning and LLM-based classifiers to detect AI-generated code, which outperform the compared techniques and show decent performance across multiple programming languages, programming tasks, and generative LLMs.

The remainder of this paper is organized as follows: in Section 2, we provide related works of AIGC detection and background about LLM-based code generation and pre-trained source code embeddings. We outlined our data collection, model building, and performance analysis in Section 3. Next, we present the evaluation results and observations in Section 4. Then, we discuss the implication for our study in Section 5. Section 6 shows potential threats to the validity of our approaches and findings. Finally, we conclude with a summary of the findings in Section 7.

II. Related Work & Background

A. Large Language Models for Code Generation

Due to advancements in the field of Natural Language Processing (NLP), LLMs have seen substantial progress in their performance and widespread use [27]. More recently, source code has also been included to train the LLMs with the goal of helping with software development activities [26]. Since these models such as CodeBERT [26], CodeT5 [28], **Starcoder2** [13], and ChatGPT [11] have been trained on vast and diverse datasets of source code and natural language from various domains [29], they showed cutting-edge effectiveness when being fine-tuned [14] or prompted [15] to solve various downstream Software Engineering (SE) tasks [30][31][32].

LLMs are widely adopted to directly generate source code given the docstring/requirement in natural language (code generation) or complete the code for the developers based on the software context (code completion) [2]. To further ease the use of LLMs for code generation/code completion and improve the quality of the generated code, researchers have been investigating fine-tuning/prompting approaches to generate code of high quality that meets developers' intentions [8][9][10]. In addition, numerous software development tools for code generation/code completion, such as Github Copilot [16], have been released and widely used by developers. All these techniques significantly increase the chance that AI (i.e., LLMs) writes a piece of code instead of a human developer.

In this study, we specifically analyzed the code generated by Gemini Pro [12], ChatGPT, GPT-4 [33], **and Starcoder2-Instruct (15B)** [13] since they are among the state-of-the-art LLMs and have been the subjects of many prior SE studies in code generation [8][9][10][17][34][35][36][37]. **We believe that this selection of LLMs covers not only the detectability of code generated by general LLMs (i.e., ChatGPT, Gemini Pro, and GPT-4) but also that of code-specific LLMs (i.e., Starcoder2-Instruct).**

B. Automated Detection of Artificial Intelligence Generated Content (AIGC)

The emergence of generative LLMs such as ChatGPT has surged the demand for accurately detecting AIGC. A variety of AIGC detectors have been developed. For example, GPTZero [22] is a widely-used commercial AIGC detector, while Sapling [23] generates the probability of whether each token in the input is AIGC. It reached 97% accuracy in identifying AI-generated texts. In addition, researchers and open-source software practitioners have also been actively developing AIGC detectors such as GPT-2 Detector [38], DetectGPT [39], and Giant Language Model Test Room (GLTR) [40], which achieved decent performance in detecting AIGC.

However, the aforementioned AIGC detectors are only designed to detect AI-generated natural language texts. Since source code has unique linguistic syntax and writing styles that differ from natural language [25], these detectors may not perform well in determining whether a human or AI writes the source code snippet. Researchers have recently found that these text detectors showed limited effectiveness when detecting AI-generated code [24][25]. To fill this gap, Nguyen et al. [24] proposed GPTSniffer, where they fine-tuned CodeBERT to classify whether a code snippet is written by AI or human. However, it could not generalize well to the data that it was not trained on (Section 3.2). In addition, only ChatGPT was queried to generate the code, while Java was the only programming language considered. Since there is a gap in terms of comprehensive analysis across different languages and generative LLMs, in our study, we investigated our approaches and the existing AIGC detectors' performance on the code generated by four widely-used state-of-the-art LLMs, namely, Gemini

Pro, ChatGPT, GPT-4, **and Starcoder2-Instruct**. We also experimented with C++ and Python in addition to Java. In order to ensure the generalizability of our approaches across different datasets, we selected three widely-used code generation benchmarks, MBPP [41], HumanEval-X [42], and CodeSearchNet [43].

C. Pre-trained Source Code Embedding

Distributed numeric code representations, pre-trained code embeddings, have been proven to be effective in various SE tasks, such as automated program repair [44][45][46], vulnerability prediction [47][48], and code clone detection [49]. Various pre-trained embedding models have been proposed by researchers to better capture syntactical/semantic information of source code and improve downstream SE tasks [50][51].

More recently, researchers have started incorporating structural information (i.e., information from Abstract Syntax Tree (AST)) and the textual information of source code into code embeddings. For example, Zhang et al. [50] first split each large AST into smaller statement ASTs and encoded these ASTs to numeric vectors by capturing the lexical and syntactical knowledge of statements. They then used a bidirectional Recurrent Neural Network (RNN) to leverage the statements' naturalness and produce the embedding. Ding et al. [51] used a two-step unsupervised training strategy to integrate the textual and structural information from the code to make the embeddings more generalized to different SE tasks. They found that including structural information could help improve the code embeddings' quality.

Since code embeddings are usually obtained by training models with large source code datasets to acquire knowledge about the semantic and syntactic meaning, we posit machine learning models trained with such embeddings have the potential to perform better than models without such information when trying to differentiate human-written code from AI-generated code. In this study, we selected CodeT5+ 110M embedding model [52], which shows state-of-the-art performance on code understanding and generation tasks, to generate embeddings for source code and AST to incorporate both textual and structural information. Then, we trained our machine learning models with these embeddings to detect AI-generated code with the goal of achieving decent performance.

III. Methodology

Our goal was to investigate the effectiveness of the current AIGC detectors in detecting AI-generated source code (RQ1). Our other goal was to classify a code snippet as human-written or AI-generated (RQ2). Finally, we analyze the impact of various source code features on the performance of the best-performing approach (RQ3). In the following subsections, we detail the applied methodology. Figure 1 shows an overview of our methodology.

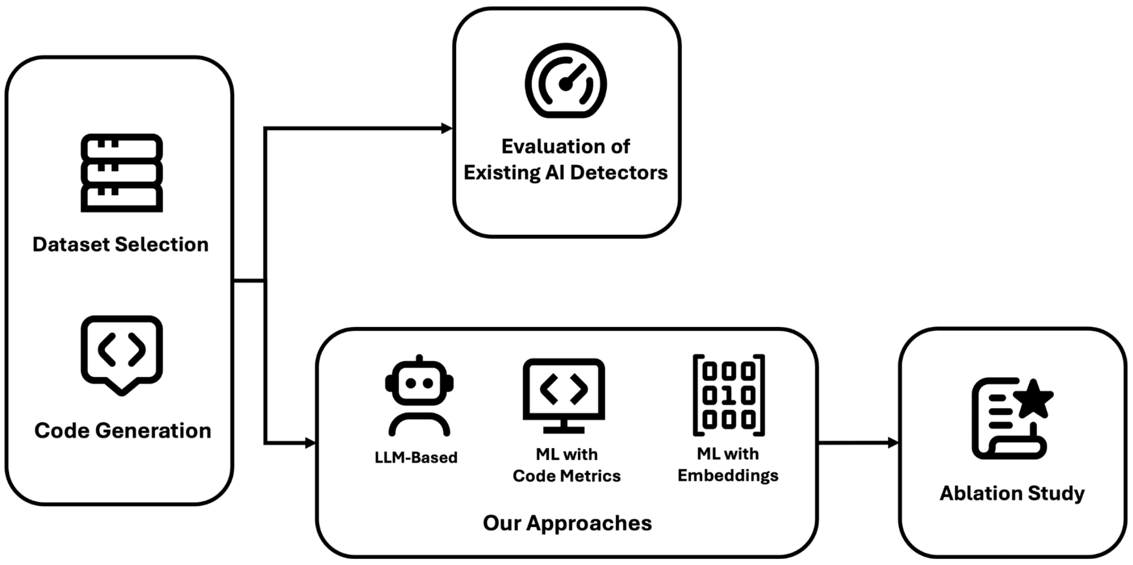


Figure 1. Overview of Research Method

A. Data Collection

Since we aimed to detect AI-generated source code and evaluate the effectiveness of the current AIGC detectors, we targeted the code generation benchmark datasets that have been studied by previous research related to code generation [8][9][10][17][34][35][36]. Specifically, we selected three datasets, namely, MBPP [41], HumanEval-X [42], and CodeSearchNet [43]. MBPP contains 974 crowd-sourced Python programming problems along with human-written Python functions solving the specified problems. The problems range from simple numeric manipulations to tasks that require basic usage of standard library functions. HumanEval-X consists of 820 data samples (function docstrings/specifications and the corresponding human-written code solutions) in Python, C++, Java, JavaScript, and Go. To make our evaluation more generalized and not restricted to human-crafted coding questions, we also included CodeSearchNet, a dataset of 2 million pairs of comment and human-written code collected from publicly available open-source non-fork GitHub repositories. In this study, we aimed to investigate the generalizability of the AIGC detectors and our approaches across multiple programming languages, so we selected Java, C++, and Python, three most widely-used programming languages by software practitioners [53][54]. We believe that selecting only three widely used languages instead of using all the languages available in the studied datasets would keep our experiments manageable. It's worth pointing out that the MBPP dataset only has code written in Python programming language, while CodeSearchNet has Java and Python but does not include C++. **To ensure there were no duplicates between different datasets, we manually reviewed all specifications and code snippets. Additionally, we used a clone detection tool, Nicad [55], to identify any potential code clones. Consequently, we found no duplicates or clones across different datasets.**

To collect the AI-generated counterparts for the human-written code in the datasets, we adopted four state-of-the-art generative LLMs that have been widely used in code generation literature, namely, ChatGPT[11], Gemini Pro[12], GPT-4[33], **and Starcoder2-Instruct**[13], to generate source code based on the natural language specifications or comments in the selected datasets. Due to a limited financial budget and prohibitively expensive OpenAI API[56], instead of generating code for the complete CodeSearchNet dataset, which consists of 457k Python and 497k Java codes, we randomly sampled 400 (confidence level 95%, margin of error 5%) data instances each for Python and Java.

In addition, the generation from the LLMs tends to be non-deterministic and creative when the temperature increases[30][57]. To make our evaluation more comprehensive, which covers code generations with greater linguistic variety, we generated multiple code snippets for each specification using the same LLM with different temperatures. To make our experiments controllable, we set the temperature as 0 and the default value provided by respective LLMs (For ChatGPT, GPT-4, **and Starcoder2-Instruct**, the default temperature is 1. For Gemini Pro, the default is 0.9) to generate code based on the specifications.

After code generation, **we obtained the code generated by each selected LLM. Combining the code generated by LLM with the human-written code which already existed in the original dataset, we obtained datasets with twice the size of original one. Then** we removed the data instances where the LLMs could not generate the source code given the specifications. We also removed code snippets that contain syntax errors, which would prevent us from extracting static code features later in Section 3.5. The statistics of the collected datasets are shown in Table 1 and Table 2.

	ChatGPT	Gemini Pro	GPT-4	Starcoder2-Instruct
MBPP (Python)	1,946	1,948	1,932	**1,948**
HumanEval-X (Python)	323	327	323	**328**
HumanEval-X (Java)	281	320	311	**328**
HumanEval-X (C++)	328	325	328	**328**
CodeSearchNet (Python)	400	400	400	**393**
CodeSearchNet (Java)	397	383	400	**390**

Table I. Collected Dataset with AI-generated Code (Temperature = 0)

	ChatGPT	Gemini Pro	GPT-4	Starcoder2-Instruct
MBPP (Python)	1,933	1,948	1,942	**1,948**
HumanEval-X (Python)	324	327	323	**328**
HumanEval-X (Java)	309	319	314	**323**
HumanEval-X (C++)	328	325	328	**317**
CodeSearchNet (Python)	400	400	400	**394**
CodeSearchNet (Java)	395	383	400	**390**

Table II. Collected Dataset with AI-generated Code (Default Temperature)

B. Compared AIGC Detectors

A number of AIGC detectors have been implemented to detect AI-generated natural language texts. Similar to Pan et al.[25][25], our goal was to investigate the effectiveness of these detectors in detecting AI-generated source code and compare them with our approaches. Following their work, we selected five AIGC detectors: GPTZero[22], GPT-2 Output Detector[38], DetectGPT[39], GLTR[40], and Sapling[23]. We ran these detectors on human-written code from our selected datasets and also on the AI-generated code by the three LLMs with different temperatures.

More recently, Nguyen et al.[24][24] built a classifier named GPTSniffer that specifically targeted the identification of AI-generated source code. They fine-tuned CodeBERT[26] with human-written code and ChatGPT-generated code to classify a code into human-written or AI-generated. In this study, we include GPTSniffer as a baseline to systematically evaluate its performance on AI-generated code from different LLMs, temperatures, and programming languages.

C. Evaluation Settings and Metrics

Similar to previous works[24], we split each of the selected dataset using the 80:10:10 ratio, allocating 80% for training, 10% for validation, and 10% for testing. **To prevent any data overlap between datasets that are from the same source but generated by different LLMs (i.e. HumanEval-C++-ChatGPT, HumanEval-C++-Gemini Pro, HumanEval-C++-GPT-4), we consistently splitted the datasets.** Each of the source code has a ground truth label of either *Human* or *AI*, representing that the code is either generated by humans or the LLMs. The metrics were calculated based on the comparison between the ground truth labels and the predicted labels. The specific metrics we used in this study include *Accuracy*, *True Positive Rate (TPR)*, *True Negative Rate (TNR)*, and *F1-score*. In this study, the positive label stands for *Human*, while the negative label represents *AI*. Detailed explanation of the metrics is as follows:

Accuracy: Accuracy is calculated as the ratio of correct predictions to the total number of predictions. It is calculated as $Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$. TP is the number of human-written code predicted correctly. TN denotes the count of correctly predicted samples of AI-generated code. We defined TN this way since its original definition corresponds to the count of correctly predicted negative class samples, and in our classification scheme, we set *AI* as the negative label. FP is the number of AI-generated code incorrectly predicted as human-written code, and FN is the number of human-written code incorrectly predicted as AI-generated code.

TPR: True Positive Rate (Recall) represents the ratio of actual positive cases that are correctly identified as positive by the classification model. It is calculated as $TPR = \frac{TP}{TP+FN}$.

TNR: True Negative Rate represents the ratio of actual negative cases that are correctly identified as negative by the classification model. It is calculated as $TNR = \frac{TN}{TN+FP}$.

F1-score: F1 score is the harmonic mean of precision and recall, where precision is calculated as $Precision = \frac{TP}{TP+FP}$. However, it can vary depending on which class (i.e., human-written or AI-generated) is designated as positive, potentially leading to a misrepresentation of the model's performance. For instance, consider a scenario where the *Human* class is designated as positive. If the model accurately predicts all instances of human-generated code but misclassifies all AI-generated code, its F1-score would be zero. Thus, we also calculated two variants of F1-scores by setting either *Human* or *AI* as the positive label. We represented the F1-score where *Human* is set as the positive label as *Human F1-score*, while the F1-score where *AI* is set as the positive label as *AI F1-score*. Finally, with the two variants, we calculated *Average F1-score*, which is the average of the two F1 score variants and is computed by taking the macro average of the *Human F1 score* and *AI F1 score*. It is calculated as $Average\ F1 = \frac{Human\ F1 + AI\ F1}{2}$. We believe this F1 score can better represent the overall effectiveness.

For each of our proposed techniques (i.e., models) in this study, we conducted the evaluation in two different settings. First, we trained the models on the training split of each of the datasets (Section 3.1), conducted hyper-parameter tuning to select the model with the best performance, and evaluated the model's performance on the testing split (*"Within" evaluation setting*). To further evaluate the generalizability of the models across different datasets with programs in different domains and written in different programming languages, we also tested the models on the testing split of another dataset (*"Across" evaluation setting*). For example, we trained a model on MBPP's training data split, and tested it on HumanEval-X's testing split. For the baseline AIGC detectors that we compared ours with, we ran these detectors on the testing splits of our datasets to evaluate their performance.

D. LLM-based Approaches

Since LLMs have shown state-of-the-art performance on code classification tasks such as defect detection and clone detection[58], we decided to harness the power of LLMs to detect AI-generated code. We used zero-shot learning, in-context learning with retrieved demonstrations, and fine-tuning. In this study, we selected ChatGPT (i.e., GPT-3.5-turbo) as the model to perform prompting/fine-tuning since it is one of the LLMs that showed state-of-the-art performance on a variety of SE tasks[30][31][32]. We did not take GPT-4 or Gemini Pro due to the unavailability of model fine-tuning.

In this study, we utilized three different representations of source code namely, the textual content of the source code, the AST representation by[59], and the concatenation of textual code and AST representation. Specifically, we followed the algorithm presented by Guo et al.[59] to generate AST representation using Tree-Sitter[60], which starts from the root node and recursively traverses the AST, appending node names with special suffixes (i.e., left, right) to the resulting sequence. The models trained with this AST representation showed state-of-the-art performance in various code understanding tasks[58]. In the following section, we will use *Code Only* for textual code representation, *AST Only* for the AST representation by[59], and *Combined* for the concatenation of *Code Only* and *AST Only*.

In the zero-shot learning setting, we prompted ChatGPT to determine whether the given source code snippet is generated by AI or human. We followed the established best practices[61][62] to design our prompt (The prompt is provided in the replication package[63]). In the in-context learning setting, we provided demonstration examples from the training datasets in the prompt in addition to the zero-shot setting. Following the best practices of setting in-context learning examples for SE tasks[64], we used BM-25[65] to retrieve four demonstration examples (two instances of human-written code and two instances of AI-generated code) from the training datasets that are the most similar to the code snippet to be predicted (i.e., test sample), and we ordered the examples based on their similarity to the code snippet in ascending order. In this study, we prompted the model with one of the three code representations to analyze the models' performance using different representations. In the fine-tuning setting, the training data consists of one of the representations (i.e., *Code Only*, *AST Only*, or *Combined*) and the ground truth labels. OpenAI API was called to fine-tune the model (i.e. ChatGPT). In the following sections, we will use *fine-tuned ChatGPT* to represent the ChatGPT model that is fine-tuned to detect AI-generated code with our datasets. We evaluated the models in *"Within" evaluation setting* only for the zero-shot learning setting as it does not require training data. For in-context learning and fine-tuning, we performed evaluation in both *"Within" evaluation setting* and *"Across" evaluation setting*.

E. Machine Learning Classifiers with Static Code Metrics

Despite the recent advancement in deep learning and LLMs, shallow machine learning algorithms still showed decent performance in some SE tasks given the appropriate features are provided[66][67]. In this study, we also investigated the feasibility and effectiveness of a machine learning-based classification approach for AI-generated source code detection.

To construct the features from source code for training our machine learning models, we used Scitools Understand[68] to extract static source code metrics as the features since these metrics have been used in a number of previous works[67][69][70][71][72]. In addition, we also collected features studied by Aljehane et al.[73], which includes Identifiers (method and variable names), Names and Operators in if, else, and while statements, Operators, Keywords, Arguments, and Method Signatures. We believe that these features also have the potential to be used to distinguish between human-written and AI-generated code since human developers focus on them when they review source code. We used Tree-Sitter[60] to collect the code features for the three programming languages in our study. Since our datasets consist of three different languages, we filtered the metrics to keep those that are commonly applicable to all three languages. For instance, Python does not have semicolon so we removed all features that are related to semicolon. In total, we retained 30 code features. Due to space constraints, we provided the whole set of these features in our replication package[63].

In order to avoid multicollinearity across features[74], we conducted Variance Inflation Factor (VIF) on all 30 features. VIF is a statistical measure used to assess multicollinearity in regression analysis. High VIF values indicate strong multicollinearity, which can lead to unstable and unreliable regression coefficients. Akinwande et al.[75] argued that a VIF value between 5 and 10 indicates a high correlation that may be problematic. Thus, we set our threshold value as 5 and make VIF values of our features below that value. As a result, we retained eight features. The finalized set of features and their corresponding descriptions are provided in Table [tab:features].

We used widely-used machine learning classifiers by researchers[67][71][76][77] such as Logistic Regression (LR)[78], K-Nearest Neighbor (KNN)[79], Multi Layer Perceptron (MLP)[80], Support Vector Machine (SVM)[80], Random Forests (RF)[81], Decision Tree (DT)[82], Gradient Boost (GB)[83], and Extreme Gradient Boost (XGB)[84]. We performed hyper-parameter tuning on each trained model to optimize the performance using random grid search[85].

F. Machine Learning Classifiers with Code Embedding

In this study, we also leveraged code embeddings to capture the information in source code with the goal of better distinguishing AI-generated and human-written code.

Features	**Definitions**
SumCyclomatic	Sum of cyclomatic complexity of all nested functions or methods.
AvgCountLineCode	Average number of lines contai- -ning source code for all nested functions or methods.
CountLineCodeDecl	Number of lines containing decl- -arative source code
CountDeclFunction	Number of functions
MaxNesting	Maximum nesting level of (if, w- -hile, for, switch etc.)
CountLineBlank	Number of blank lines
Keywords	The ratio between the number of language keyword tokens to the number of total tokens
Operators in if, else, and while statements	The ratio between the number of operators in if, else, and while sta- -tements to the number of total tokens

Table III. Collected Source Code Features

We experimented with a pre-trained source code embedding model. Specifically, CodeT5+ 110M embedding model[52] **was selected because it was the latest code embedding model at the time we conducted the experiments.** The generated embeddings were used as the features to train machine learning models. In order to generate the embeddings that capture different aspects of source code, we selected three code representations, namely, *Code Only*, *AST Only*[59], and *Combined* where we separated the concatenation of two representations with a special separator token following[59]. The goal was to explore the embeddings of various representations of source code (i.e., structural and textual information) to investigate the most distinguishing representations captured by embeddings. We used the same machine learning algorithms and steps used in Section 3.5, including random grid search. Figure 2 shows the overview of this approach.

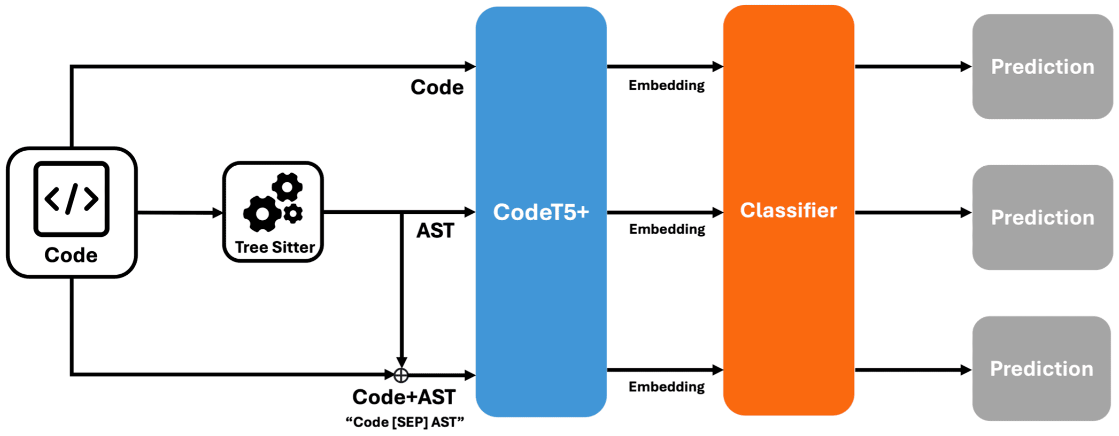


Figure 2. Overview of Machine Learning Classifiers with Embeddings

Furthermore, to explore performance inconsistencies across various approaches, we compared the similarity between AI-generated and human-written code to explain classification performance. We used semantic embeddings of code, which capture the underlying meaning of the code[28][86][87], providing a robust basis for comparison and understanding the differences between the two classes. We computed the cosine similarity between the code embeddings of AI-generated and human-written code from the same specification in our dataset and averaged these similarity values. The averaged cosine similarities between AI-generated and human-written code embeddings were then used to compare semantic similarity across the different LLMs. This process was carried out respectively for each of the four LLMs used in our study.

In addition, given that discrepancies between training and testing datasets can affect classification performance, we investigated these differences to understand why performance degrades in the *"Across"* evaluation setting compared to the *"Within"* evaluation setting. We focused on the cosine similarity of *AST Only* embeddings, since models trained with these embeddings performed best in the *"Within"* evaluation setting. In the *"Within"* setting, we averaged the *AST Only* embeddings separately for each dataset's training and testing splits. We then measured the cosine similarity between these averaged embeddings. We followed a similar process for the *"Across"* setting: averaging the *AST Only* embeddings for the training and testing splits but comparing the cosine similarity with splits from different datasets. We compared 30 training-testing split combinations for each LLM. Finally, we averaged the cosine similarity values for each evaluation setting and compared the two sets of values.

G. Ablation Study

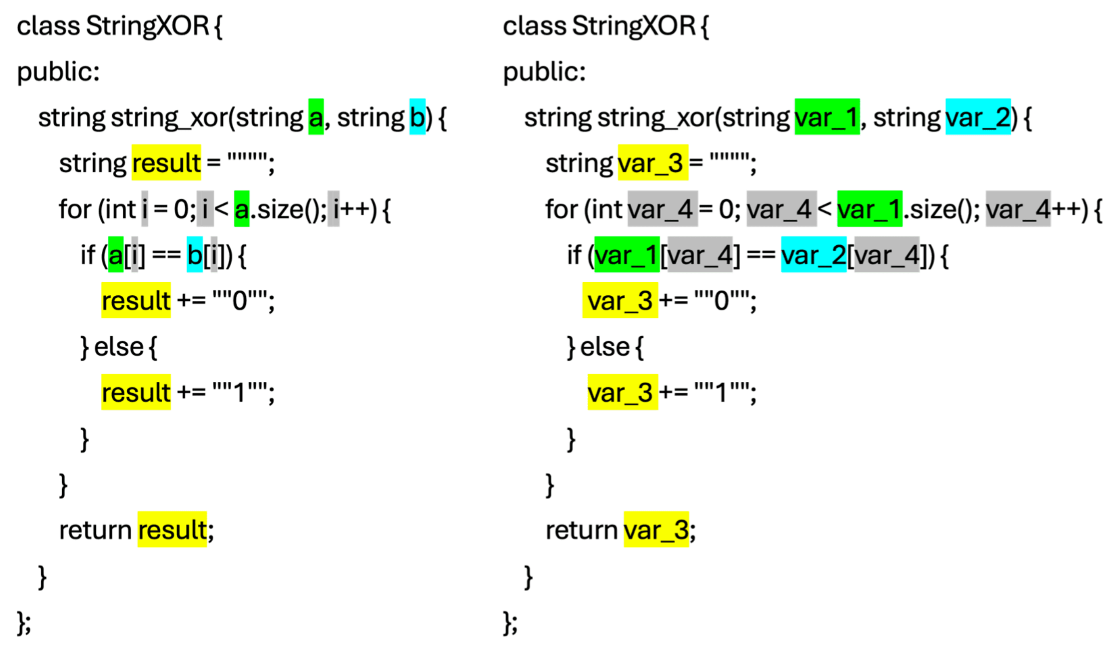
Since our embedding-based machine learning models with *AST Only* perform the best among all other approaches (Section 4.4), we conducted an ablation study to investigate the impact of different source code features on the models' performance. To do so, we first took all the 30 code features that are applicable to C++, Java and Python in Section 3.5. Among these features, we selected the ones that we can modify in the code without altering the code logic, namely *Comment Lines*, *Variable Names*, *Method Names*, and *Blank Lines*. We eliminated *Blank Lines* from our experiments since removing blank lines does not have any impact on the AST representation of the source code.

Next, we established the following code variant types that do not affect the code logic, based on each of the three features:

- Code with no comment line
- Code with uniform variable names (prefixed with 'var_' and numbered sequentially, starting from var_1)
- Code with uniform method names (prefixed with 'func_' and numbered sequentially, starting from func_1)

In order to create the mentioned variants, we used Tree-Sitter to parse the AST and change its relevant nodes. To create *Code with no comment line*, we removed the *comment* and *block_comment* nodes from the AST. To make the function names uniform (*Code with uniform method names*), we changed the *method_declaration* or *function_definition* nodes given that each programming language has specific AST node types for function declaration/definition. Some language-specific functions such as the "main" method in C++ and Java, or the constructor and deconstructor methods (i.e. "__init__" in Python) remained untouched. For renaming the variables (*Code with uniform variable names*), we performed a similar approach as we did for *Code with uniform method names*. Figure 4 shows an example of *Code with uniform variable names*.

Different AST nodes were changed based on the AST structure and node types of each programming language. For instance, we changed the nodes, including "identifier", "pattern_list", "assignment", "typed_parameter" for Python code, while we modified "local_variable_declaration", "formal_parameter" nodes for Java and "init_declarator" for C++. The complete implementation is available in our replication package[63]. After generating the code variants, we produced the embeddings for the AST of these code variants. Finally, we trained machine learning classifiers with these *AST Only* embeddings of the code variants for each variant type. Then we performed Welch's t-test[88] and calculated the effect size (Cohen's D[89]) using the *Average F1-scores* of each variant compared to those of the original code.



```
class StringXOR {
public:
  string string_xor(string a, string b) {
    string result = """";
    for (int i = 0; i < a.size(); i++) {
      if (a[i] == b[i]) {
        result += ""0"";
      } else {
        result += ""1"";
      }
    }
    return result;
  }
};
```

```
class StringXOR {
public:
  string string_xor(string var_1, string var_2) {
    string var_3 = """";
    for (int var_4 = 0; var_4 < var_1.size(); var_4++) {
      if (var_1[var_4] == var_2[var_4]) {
        var_3 += ""0"";
      } else {
        var_3 += ""1"";
      }
    }
    return var_3;
  }
};
```

Figure 3. Example of Code with uniform variable names Variant

IV. Results

In this section, we organized the results of this study based on our research questions in Section 1. **Due to space constraints, we present the results for the default temperature value. The result for 0 temperature is included on the companion website[63]. We also include both the results for *"Within"* and *"Across"* evaluation settings to provide a comprehensive view of the results.**

	ChatGPT				Gemini Pro				GPT-4				Starcoder2-Instruct				Mean of Every Datasets			
	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1
GPT2 Output Detector	45.61	0.00	100.00	31.88	36.66	0.00	100.00	34.54	46.65	49.07	53.07	32.25	**50.22**	**26.49**	**73.95**	**45.54**	**48.86**	**18.89**	**81.75**	**36.05**
DetectGPT	46.44	2.95	100.00	33.61	38.97	7.21	99.51	39.13	43.71	43.31	52.26	37.24	**52.64**	**7.26**	**98.01**	**39.70**	**48.99**	**15.18**	**87.45**	**37.42**
Sapling	55.17	94.46	11.60	44.10	33.79	92.73	7.83	37.25	45.90	43.62	35.09	35.55	**52.18**	**47.49**	**56.88**	**50.81**	**49.97**	**69.57**	**27.85**	**41.93**
GPTZero	53.13	85.00	0.83	35.22	45.34	100.00	0.00	31.16	38.82	39.65	29.85	34.98	**53.87**	**28.74**	**79.00**	**47.39**	**47.83**	**63.35**	**27.42**	**37.19**
GLTR	44.53	48.73	44.64	41.63	41.83	45.32	68.46	47.80	51.83	45.48	46.59	46.59	**47.21**	**47.08**	**47.34**	**44.22**	**50.83**	**46.65**	**51.76**	**45.06**

Table IV. Performance of ExistingAIGC Detectors (Default Temperature)

A. RQ1: How do existing AIGC detectors perform on detecting AI-generated source code?

To answer RQ1, we evaluate the performance of existing AIGC detectors by running them on the testing splits of our datasets (Section 3.3). As mentioned in Section 3, five AIGC detectors for AI-generated natural language text (GPTZero, GPT-2 Output Detector, DetectGPT, GLTR, and Sapling) and a state-of-the-art AI-generated source code detector, GPTSniffer, are included as baseline approaches that we compare ours with in this study.

Table [tab:aigc1] (AVG_F1 stands for *Average F1-score*) shows the performance of all five AIGC detectors for natural language text when the temperatures of the LLMs that generated the code are set as the default value (1 for ChatGPT, GPT-4, **and Starcoder2-Instruct**, 0.9 for Gemini Pro). We average the values of all the metrics across all datasets whose generation LLM is the same to obtain an overview of the performance. We find that the performance of detecting code when the generation LLMs' temperatures are set to 0 and the default value is similar, so we put the results when the temperatures of the LLMs are 0 to our replication package[63]due to space constraints. Similar to what researchers have found[24][25], their *Accuracy* is mostly less than 0.6, indicating their ineffectiveness in detecting AI-generated source code. Since source code has unique linguistic syntax and writing styles that are different from natural language, the reasons for such low performance in detecting AI-generated source code could include that these AIGC detectors were trained with only natural language texts.

Moreover, a certain AIGC detector can have a different performance in detecting code generated by different LLMs. For example, DetectGPT tends to classify human-written, ChatGPT-generated, and Gemini Pro-generated code as AI-generated based on the high *TNR* and low *TPR* values, but it does not have this issue when identifying GPT-4-generated code. We also find that some techniques tend to classify both human-written and AI-generated code as human-written code, such as GPTZero, on the datasets with code generated by Gemini Pro and ChatGPT. **Code generated by Starcoder2-Instruct showed a relatively higher average F1-score compared to code generated by other LLMs across all AIGC detectors.** Overall, all of them show limited effectiveness in detecting AI-generated code.

In addition, the *Average F1-score* across different generative LLMs does not show a significant difference, which indicates that AIGC detectors perform poorly regardless of the LLMs used to generate the AI-generated source code. We also find a similar trend in terms of the temperature settings of the generative LLMs[30][57](i.e,. LLMs used for code generation). Therefore, neither the generative LLMs nor their temperature settings affect the capability of existing AIGC detectors to detect AI-generated source code.

Observation 1: Existing natural language AIGC detectors perform poorly in classifying human-written and AI-generated source code.

As for GPTSniffer, a state-of-the-art AIGC detector fine-tuned with source code, we provide its performance results on all our datasets in Table 3. While GPTSniffer has been fine-tuned on ChatGPT-generated Java code, it still shows poor performance in detecting source code from different datasets containing code written in Java (i.e. HumanEval-Java and CodeSearchNet-Java) other than its training data, even for Java code generated by ChatGPT. For example, it only shows the *Accuracy* of 32.26 and *Average F1-score* of 24.39 on Java code generated by ChatGPT in HumanEval dataset. When it comes to other programming languages, it shows limited performance on detecting Python code (i.e. *Accuracy* of 44.85 and *Average F1-score* of 30.96 on MBPP dataset with code generated by ChatGPT). Surprisingly, it shows a superior performance in detecting AI-generated code in C++ where it achieves *Accuracy* of 100 and *Average F1-score* of 100 on HumanEval-C++ dataset with ChatGPT-generated code. The reason could be that the testing split of this dataset only contains 33 data instances. A relatively smaller number of instances may have led to overfitting.

For detecting code generated by LLMs other than ChatGPT, its *Accuracy* fluctuates around 50. Moreover, it tends to classify code as AI-generated based on the high *TNR* and low *TPR* values. In addition, we also trained and tested our best-performing machine learning models (Section 4.4) on GPTSniffer's data, and we achieved similar performance (*Average F1-score* of 1). Thus, we believe that GPTSniffer does not show a decent performance on datasets from a different domain (i.e., open-source projects in Github), code snippets written in programming languages other than Java, and generated by LLMs other than ChatGPT, which severely undermines its applicability to AI-generated code detection.

Observation 2: A state-of-the-art AIGC detector for source code, GPTSniffer, still performs poorly in classifying human-written and AI-generated source code.

Dataset-Generation LLM-Language	**ACC**	**TPR**	**TNR**	**AVG_F1**
MBPP-ChatGPT-Python	44.85	0.00	100.00	30.96
MBPP-Gemini Pro-Python	49.23	0.00	100.00	32.99
MBPP-GPT-4-Python	47.69	0.00	100.00	32.29
MBPP-Starcoder2-Instruct-Python	**72.95**	**52.04**	**93.87**	**71.72**
HumanEval-ChatGPT-Python	60.61	0.00	100.00	37.74
HumanEval-ChatGPT-Java	32.26	0.00	100.00	24.39
HumanEval-ChatGPT-C++	100.00	100.00	100.00	100.00
HumanEval-Gemini Pro-Python	54.55	0.00	100.00	35.30
HumanEval-Gemini Pro-Java	65.62	0.00	100.00	39.63
HumanEval-Gemini Pro-C++	60.61	85.71	42.11	60.02
HumanEval-GPT-4-Python	54.55	0.00	100.00	35.30
HumanEval-GPT-4-Java	57.58	0.00	100.00	36.54
HumanEval-GPT-4-C++	96.97	95.00	100.00	96.87
HumanEval-Starcoder2-Instruct-Python	**50.00**	**0.00**	**100.00**	**33.34**
HumanEval-Starcoder2-Instruct-Java	**50.00**	**0.00**	**100.00**	**33.34**
HumanEval-Starcoder2-Instruct-C++	**85.29**	**100.00**	**70.58**	**84.96**
CodeSearchNet-ChatGPT-Python	50.00	0.00	100.00	33.34
CodeSearchNet-ChatGPT-Java	47.50	0.00	100.00	32.21
CodeSearchNet-Gemini Pro-Python	50.00	0.00	100.00	33.34
CodeSearchNet-Gemini Pro-Java	47.50	0.00	95.00	32.21
CodeSearchNet-GPT-4-Python	50.00	0.00	100.00	33.34
CodeSearchNet-GPT-4-Java	35.90	0.00	93.33	26.42
CodeSearchNet-Starcoder2-Instruct-Python	**57.50**	**30.00**	**85.00**	**54.02**
CodeSearchNet-Starcoder2-Instruct-Java	**52.50**	**30.00**	**75.00**	**49.96**
Average	**57.24**	**20.53**	**93.95**	**45.01**

Table V. The Evaluation Of GPTSniffer On Our Datasets (Default Temperature)

B. RQ2: How can we improve the performance of AI-generated source code detection? (LLM-based approaches)

To improve the effectiveness of detecting AI-generated source code, we first experiment with a variety of LLM-based approaches. In this section, we evaluate the performance of ChatGPT after zero-shot learning, in-context learning and fine-tuning (*fine-tuned ChatGPT*) where we prompted/trained the model with three different representations of source code (Section 3.4), namely, *Code Only*, *AST Only*, and *Combined*.

We average the values of all the metrics across all datasets whose generation LLM is the same to obtain an overview of the performance as in **Table VI and VII for *"Within" evaluation setting* and *"Across" evaluation setting*, respectively. Our detailed results for both evaluation settings are provided in our replication package**[63]. For *fine-tuned ChatGPT*, their *Accuracy* and *Average F1-score* can reach more than 80 on some datasets such as datasets with ChatGPT and GPT-4-generated code **in *"Within" evaluation setting***, which suggests that *fine-tuned ChatGPT* with our datasets can detect AI-generated code significantly better than existing AIGC detectors. We also find that *fine-tuned ChatGPT* performs better in identifying AI-generated code produced by LLMs with a high temperature than that of a low temperature.

However, the fine-tuned models only show around 40 in terms of *Accuracy* and *Average F1-score* in *"Across" evaluation setting* where the models are evaluated on the testing splits of the different datasets (Section 3.3). For example, *fine-tuned ChatGPT* with MBPP dataset whose AI-generated Python code is by Gemini Pro shows only 50 in *Accuracy* when being tested on CodeSearchNet dataset whose AI-generated code is written in Java (default temperature for the generative LLM). This indicates that *fine-tuned ChatGPT* using *Code Only* as the input still lack generalizability across code snippets from different domains and different programming languages.

Similar to *fine-tuned ChatGPT* on *Code Only*, *fine-tuned ChatGPT* on *AST Only* also significantly outperforms its zero-shot and in-context learning counterparts. However, the *fine-tuned ChatGPT* performs significantly worse when using *AST Only* as input than that of *Code Only*. For example, the *Average F1-score* for the *fine-tuned ChatGPT* with human-written and ChatGPT-generated code using *AST Only* is **58.96** when the default temperature is used for code generation, while the score reaches **81.79** for *Code Only*. In addition, *fine-tuned ChatGPT* on *AST Only* shows poor performance in *"Across" evaluation setting* (mean *Average F1-score* of around 43 across all datasets when the generative LLMs' temperatures are 0, and 44 for default temperatures), suggesting limited generalizability.

When we concatenate *Code Only* and *AST Only* as input (i.e., *Combined*), we observe similar mean *Average F1-scores* to those when *Code Only* is used as input in zero-shot learning and in-context learning. For *fine-tuned ChatGPT*, there is a notable performance drop compared to when *Code Only* is used as input. For example, *fine-tuned ChatGPT* on *Code Only* outperform that of *Combined* by more than 10 in terms of mean *Average F1-scores* in the default temperature setting. Thus, we believe that AST representation of source code may not be suitable as the input for fine-tuning ChatGPT to detect AI-generated code.

Interestingly, identifying source code generated by Gemini Pro presents the lowest mean *Average F1-score*, suggesting it is particularly challenging to identify the code generated by this generative LLM using *fine-tuned ChatGPT*. For example, *fine-tuned ChatGPT* only has around 60 in *Average F1-scores* and *Accuracy* when detecting Gemini Pro-generated code, while it shows 70-80 when detecting ChatGPT, GPT-4, or **Starcoder2-Instruct** generated code when the temperature is set as 0.

Observation 3: *fine-tuned ChatGPT* significantly outperforms zero shot and in-context learning. In addition, AST representation is not suitable as the input for *fine-tuned ChatGPT* to detect AI-generated code.

C. RQ2: How can we improve the performance of AI-generated source code detection? (Machine Learning Classifiers with Static Code Metrics)

We evaluated the performance of the machine learning classifiers on the testing splits of the datasets. RF model shows the best performance when the temperature is set to 0, while GB shows the highest mean *Average F1-score* in the default temperature setting. Similar to LLM-based approaches, we average the values of all the metrics across all datasets whose generation LLM is the same to obtain an overview of the performance. The results obtained under ***"Within" evaluation setting*** **are presented in Table VI and the results under the *"Across" evaluation setting* are presented in Table VII.**

The difference in *Accuracy* and *Average F1-score* across different generative LLMs shows that machine learning-based detection techniques have varied effectiveness in detecting code generated by different LLMs. For example, when the generative LLMs' temperatures are set to 0, the RF classifier has more than 80 in *Average F1-score* and *Accuracy* for detecting ChatGPT-generated code, while it only has around 66 in detecting Gemini Pro-generated code. However, these classifiers still outperform existing AIGC detectors. We also find that our machine learning classifiers tend to perform better (i.e., higher overall mean *Average F1-score*) on detecting code generated with higher temperature (i.e., default temperature) than code generated with a temperature of 0.

In *"Across" evaluation setting*, the machine learning classifiers have a mean *Average F1-score* of around 50 for both temperature of 0 and default temperature, which indicates the limited performance of the trained machine learning models when they are tested on different programming languages and datasets from other sources.

Observation 4: The machine learning classifiers trained with static code features can identify AI-generated code. However, they show varied effectiveness in detecting code generated by different LLMs.

D. RQ2: How can we improve the performance of AI-generated source code detection? (Machine Learning Classifiers with Embeddings)

Instead of using static code features as in Section 4.3, we adopted the code embeddings generated from *Code Only*, *AST Only*, and *Combined* (Section 3.4) using CodeT5+ as the feature vectors to train our machine learning models. We take SVM for the embedding of *AST Only*, MLP for *Code Only*, and LR for *Combined* when the generative LLMs' temperatures are set to 0. In the default temperature setting, we select LR for all three representations.

We highlight the highest scores across all approaches in Table VI. Our findings are: machine learning models trained with the embeddings of *AST Only* demonstrate the best performance across all the other approaches that we experimented with in this study, indicating that machine learning models are able to capture the difference between AI-generated and human-written code. The mean *Average F1-score* is **81.44 when the temperature is set to 0, and 82.55** when the temperature is set to the default value. Thus, our machine learning models trained with embeddings significantly outperform AIGC detectors, including GPTSniffer.

However, these models still show inconsistent performance in identifying AI-generated code by different LLMs, similar to LLM-based and machine learning models trained with static code features. For example, their *Accuracy* reaches nearly 90 (i.e., 89.89) when being trained and evaluated on human-written and ChatGPT-generated code, while it's only 73.36 for Gemini Pro, given the temperature is set as default.

Our results on exploring performance inconsistencies across various approaches show that the cosine similarity between AI-generated and human-written code embeddings were as follows: 73.87 for ChatGPT, 77.58 for Gemini Pro, 76.56 for GPT-4, and 75.20 for Starcoder2-Instruct. These values indicate a high degree of semantic similarity between the code embeddings of AI-generated and human-written code. Consistent with our result that code generated by Gemini Pro was the hardest to detect, the cosine similarity between embeddings of AI-generated and human-written code was the highest for Gemini Pro. This indicates that Gemini Pro's code is semantically closest to human-written code, making it more difficult to distinguish. In contrast, we found that the cosine similarity for code generated by ChatGPT and Starcoder2-Instruct was the lowest, suggesting greater semantic differences. This aligns with our finding that code generated by ChatGPT and Starcoder2-Instruct was the easiest to detect. Therefore, the varying semantic differences between AI-generated and human-written code across different LLMs may account for the performance inconsistencies observed in code detection.

As for the model's generalizability, the results of the *"Across" evaluation setting* show that it still struggles to detect AI-generated code written in other programming languages and from other domains (around 42 in terms of mean *Average F1-score* across all datasets whose generative LLMs' temperatures are set to 0, and 45 for default temperature). **The results of analyzing the discrepancies of *AST Only* embeddings between training and testing splits are shown in Table [tab:within_vs_across_cossim]. On average, the cosine similarity in the *"Across"* evaluation setting was about 20% lower than in the *"Within"* evaluation setting (77.87 vs. 98.68). This finding suggests that dissimilarities between training and testing datasets contribute to performance degradation in the *"Across"* setting. However, further in-depth analysis is needed to fully understand all the factors affecting performance in the *"Across"* setting.**

Observation 5: The machine learning models trained with code embeddings of *AST Only* show the best performance among all the approaches. However, they show varied effectiveness in detecting code generated by different LLMs.

	ChatGPT				Gemini Pro				GPT-4				Starcoder2-Instruct				Overall Average			
	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1
GPT-3.5 (Zero shot)-Code	52.64	99.90	2.57	36.60	51.67	99.81	0.09	34.10	54.15	100.00	0.05	35.11	52.11	99.93	0.68	34.78	52.64	99.91	0.85	35.15
GPT-3.5 (In-context)-Code	51.71	81.31	18.50	44.51	41.19	66.64	18.01	37.20	55.58	83.80	25.76	49.80	49.53	80.21	18.12	42.94	49.50	77.99	20.10	43.61
GPT-3.5 (Fine-tuned)-Code	82.18	89.13	77.24	81.79	70.41	82.23	60.81	68.60	88.89	93.33	84.18	88.22	82.98	89.34	77.61	82.26	81.12	88.51	74.96	80.22
GPT-3.5 (Zero shot)-AST	48.16	69.77	19.30	40.92	48.56	74.73	25.81	44.19	43.92	64.61	21.51	39.85	48.19	70.53	25.90	42.73	47.21	69.91	23.13	41.92
GPT-3.5 (In-context)-AST	50.77	82.24	13.69	41.07	40.59	73.83	12.42	35.21	53.32	83.38	21.72	46.91	49.19	70.38	27.55	42.47	48.47	77.46	18.85	41.41
GPT-3.5 (Fine-tuned)-AST	64.18	46.71	81.82	58.96	55.86	56.76	59.55	53.07	65.64	83.22	48.96	62.61	60.10	53.50	68.20	54.53	61.44	60.05	64.63	57.29
GPT-3.5 (Zero shot)-Code+AST	55.50	97.78	4.29	38.78	44.65	97.54	0.69	31.30	52.05	98.19	1.77	35.51	52.56	97.99	2.28	35.89	51.19	97.87	2.26	35.37
GPT-3.5 (In-context)-Code+AST	46.27	50.71	40.43	42.90	37.90	52.01	22.01	32.70	52.57	61.81	43.50	50.25	49.10	59.15	38.30	45.66	46.46	55.92	36.06	42.88
GPT-3.5 (Fine-tuned)-Code+AST	75.25	91.68	57.12	72.57	63.06	85.23	44.26	59.37	75.37	90.42	61.87	71.72	67.09	73.66	61.43	61.79	70.19	85.25	56.17	66.36
Code Metrics (GB)	78.72	84.81	73.62	78.28	72.78	81.10	65.89	72.41	76.03	78.82	72.52	75.57	74.01	80.86	67.02	73.67	75.84	81.58	70.68	74.98
Code Embedding-Code	81.15	83.37	80.41	80.91	68.12	78.50	60.27	67.87	81.76	84.20	79.90	81.47	86.09	86.21	86.01	86.00	79.28	83.07	76.65	79.06
Code Embedding-AST	89.89	91.76	88.34	89.80	73.36	85.30	64.02	72.97	82.26	83.49	81.12	82.10	85.40	85.79	84.96	85.33	82.72	86.59	79.61	82.55
Code Embedding-Code+AST	84.85	84.67	85.45	84.46	70.05	79.90	62.28	69.81	82.16	81.88	82.25	82.00	87.12	90.28	83.93	87.10	81.05	84.18	78.48	80.84

Table VI. Overall Performance Comparison - Within (Default Temperature)

	ChatGPT				Gemini Pro				GPT-4				Starcoder2-Instruct				Overall Average			
	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1
GPT-3.5 (In-context)-Code	51.76	66.13	33.17	44.29	47.38	81.75	19.50	40.92	55.12	86.78	19.75	46.24	48.36	82.17	14.55	40.01	50.66	79.21	21.74	42.87
GPT-3.5 (Fine-tuned)-Code	52.11	56.50	45.63	44.80	50.64	72.27	34.30	43.44	53.26	49.10	54.55	46.01	51.26	12.98	89.53	37.61	51.82	47.71	56.00	42.96
GPT-3.5 (In-context)-AST	46.69	40.21	53.53	43.93	51.05	49.21	51.95	48.47	51.38	42.30	59.02	48.74	47.95	40.99	54.95	45.78	49.27	43.18	54.86	46.73
GPT-3.5 (Fine-tuned)-AST	52.77	39.29	64.67	44.33	51.17	47.87	53.49	46.27	51.47	60.54	40.53	44.67	52.93	30.49	75.51	39.75	52.09	44.55	58.55	43.76
GPT-3.5 (In-context)-Code+AST	47.50	54.70	41.11	44.16	48.08	60.96	35.71	45.02	54.08	60.78	44.40	50.46	49.95	59.66	40.10	47.82	49.90	59.02	40.33	46.86
GPT-3.5 (Fine-tuned)-Code+AST	51.72	51.55	49.57	46.32	52.64	71.98	34.70	45.42	54.48	56.08	51.90	48.72	51.54	29.68	73.50	45.47	52.59	52.32	52.42	46.48
Code Metrics (GB)	56.79	54.43	59.11	51.82	58.35	59.93	56.15	54.62	55.32	47.68	64.02	50.96	51.88	37.63	66.33	46.72	55.59	49.92	61.40	51.03
Code Embedding-Code	51.15	39.01	63.72	42.20	57.23	50.91	62.70	50.72	52.19	35.71	69.11	43.38	53.47	38.67	69.42	46.83	53.51	41.08	66.24	45.78
Code Embedding-AST	53.47	38.67	69.42	46.83	53.55	38.06	67.38	45.41	50.84	26.09	78.72	42.83	50.07	18.26	82.12	37.78	51.98	30.27	74.41	43.21
Code Embedding-Code+AST	54.85	42.37	67.89	46.64	55.27	45.83	64.11	48.42	55.94	37.82	74.49	47.58	49.09	17.31	81.06	38.36	53.79	35.83	71.89	45.25

Table VII. Overall Performance Comparison - Across (Default Temperature)

	ChatGPT		Gemini Pro		GPT-4		Starcoder2-Instruct		Overall Average	
	Within	Across	Within	Across	Within	Across	Within	Across	Within	Across
Cosine Similarity of AST Embeddings	98.61	78.76	98.65	75.18	98.63	78.09	98.85	79.46	98.68	77.87

Table VIII. Comparison on the Characteristics of Training and Testing dataset - Within vs. Across (Default Temperature)

E. RQ3: How do the source code features captured by embeddings contribute to the overall effectiveness? (Ablation Study)

In our ablation study, we generate the code variants as described in Section 3.7 and produce embeddings for these variants using CodeT5+. Specifically, we used the best-performing model across all the approaches we experimented with (the model with the highest mean *Average F1-score*) in this ablation study as the AI-generated code detection model, namely, the GB classifier trained with the embedding of *AST Only* with code generated by LLMs in default temperature setting. We use *Best model* to represent this model for simplicity. Then, we compare this model's performance with the performance of the GB classifiers trained on the embeddings of the code variants of the three variant types. The goal is to investigate whether the change of the code features captured by code embeddings can have an impact on the overall effectiveness of the *Best model*.

The results in Table [tab:ablation_result] show that the variants of *Code with no comment line* have the most impact on the performance of the *Best model* where it decreases the mean *Average F1-score* by **3.82**, while variants of *Code with uniform variable names* and *Code with uniform method names* have negligible effect. We conducted t-test and measured the effect size on the *Average F1-score* of each variants against the *Average F1-score* of the *Best model* to see if there were any statistically significant differences between the values. For variants of *Code with no comment line*, we observed p-value of **0.4543** and effect size of **0.2178** . The results show that the impact is statistically insignificant with a small effect size.

Observation 6: Removing code comments has an impact on the effectiveness of our best model. However, the impact is statistically insignificant with a small effect size.

	ChatGPT				Gemini Pro				GPT-4				Starcoder2-Instruct				Overall Average				Differenc	
	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR
Base AST Code Embedding	89.88	91.76	88.33	89.80	73.35	85.30	64.02	72.97	82.25	83.48	81.11	82.10	**85.40**	**85.79**	**84.96**	**85.33**	**82.72**	**86.59**	**79.61**	**82.55**	-	-
Uniform Method Names	88.36	92.89	85.07	88.33	75.30	88.93	64.78	74.97	80.90	82.56	79.59	80.83	**87.22**	**90.05**	**84.33**	**87.20**	**82.95**	**88.61**	**78.44**	**82.83**	**0.22**	**2.02**
Uniform Variable Names	85.49	87.80	83.49	85.40	75.07	85.35	67.06	74.56	83.52	86.66	80.51	83.39	**85.39**	**88.73**	**81.95**	**85.34**	**82.37**	**87.14**	**78.25**	**82.17**	**-0.36**	**0.55**
No Comment Line	83.98	86.21	81.58	83.43	73.29	83.79	64.56	72.68	78.03	79.99	75.38	77.37	**81.49**	**83.95**	**78.97**	**81.45**	**79.20**	**83.49**	**75.12**	**78.73**	**-3.53**	**-3.1**

Table IX. The Result of Ablation Study

V. Discussion

Despite the fact that the LLMs such as ChatGPT have shown state-of-the-art performance using zero-shot and in-context learning in a variety of SE tasks, including code summarization [30], code refinement [31], and Android bug replay [32], they both have relatively poor performance by showing around 40 in *Average F1-scores* and *Accuracy* across all datasets in detecting AI-generated code. Thus, we believe that even powerful LLMs such as ChatGPT lack the capability to effectively identify AI-generated source code if they are not fine-tuned, even when retrieved demonstration examples are provided in the prompt.

Based on our results in 4, we are able to reach over 80 in terms of *Accuracy* and *Average F1-scores*. Compared to existing AIGC detectors and GPTSniffer, our models have achieved significantly better performance. However, there is still room for improvement since all our models perform poorly in the *"Across" evaluation setting* where they fail to be effectively generalized to code written in programming languages other than the language of their training data and from different domains or sources. This is not surprising as it also happens to other SE tasks. For example, while there has been research improving the performance of defect prediction models for decades, the task can still be challenging when the models are evaluated on data from software projects other than the project that they are trained with [90]. Thus, further research from the community is required to devise generalizable detection techniques.

For using different representations of source code to train our models (*Code Only*, *AST Only*, and *Combined*), there is no representation that always outperforms another across all our approaches. *Code Only* shows the best performance in fine-tuning ChatGPT, while machine learning models trained with the embeddings of *AST Only* achieve the highest *Accuracy* and *Average F1-scores* among all other approaches. This may suggest the need for a multi-modal representation of code for better AI-generated code detection, and further research is needed to explain this observation conclusively.

Our models have different performances, and they, in general, perform better in detecting code generated by ChatGPT, GPT-4, **and Starcoder2-Instruct** than that of Gemini Pro. Some probable reasons behind this could be that Gemini Pro tends to generate code that resembles what humans would write, or our features and embeddings could not capture the difference between Gemini Pro-generated code and human-written code. As LLMs are being built and trained at an ever-increasing rate with better performance, more effective approaches with distinguishing features or embeddings should be implemented to detect AI-generated code that's getting ever closer to human-written one.

Finally, in our ablation study (Section 4.5), while removing comments from the source code, has the most impact on the *Best model*'s performance, the impact is statistically insignificant.

VI. Threats to Validity

We have taken all reasonable steps to mitigate potential threats that could hamper the validity of this study.

Construct validity

It is possible that the prompt we used to generate the source code with the LLMs might have impacted the generated code quality. To mitigate this threat, we followed the best practices of prompt design and the setting of in-context examples [61][62][64], **such as explicitly defining the persona of the LLMs, instructing the task definition, and providing necessary context information. However, we did not use any advanced prompting techniques, such as Chain-of-Thought, which could potentially improve the code quality. This omission may affect the validity of our results, as these techniques might yield better quality code and influence the detection outcomes.**

Another concern is the potential bias arising from the specifications and code in the datasets, which are collected from public platforms such as LeetCode and Github. ChatGPT, Gemini Pro, **or Starcoder2-Instruct** might have been trained on these datasets. **Since LLMs are probabilistic generative models that create code based on specifications rather than retrieving it from their training datasets, the risk of bias in AI-generated code is likely reduced. However, this issue is more pronounced with human-written code, which is part of the dataset and thus more accessible for LLMs during their learning. LLMs may therefore detect human-written code more effectively than AI-generated code because they may have encountered human-written code during their pre-training phase. This potential bias could influence our findings.**

Internal validity

The generation of the LLMs, such as ChatGPT or Gemini Pro, is non-deterministic, especially when the temperature is set to 1. This may negatively impact the reproducibility of our study.

There might have been possible mistakes in implementing the baseline AIGC detectors that we compared with. To alleviate this threat, we directly used the code implementations published by the authors [24][25]. For commercial tools (i.e., GPTZero and Sapling), we used their publicly available API.

External validity

The findings in this study may only be valid for the three datasets and the three LLMs we selected. However, the code snippets and specifications in the datasets come from a diverse range of sources. The three LLMs are among the most widely-used LLMs by software developers for code generation and they are built by different organizations (i.e. OpenAI VS Google). Moreover, multiple programming languages that are widely adopted by software practitioners are considered in this study. Thus, we believe the concern about the generalizability of our findings is mitigated.

VII. Conclusion

Our study identified that existing AIGC detectors perform poorly in detecting AI-generated source code. Our results indicated that current techniques for detecting AI-generated source code need to be improved, underscoring the need for further research in this area. From these findings, we suggested the following three different approaches for AI-generated source code detection: (1) LLM-based approach, (2) Machine Learning with Code Metrics, and (3) Machine Learning with Code Embeddings. We evaluated our approaches on multiple datasets from different sources, LLM, and languages. Among the approaches, Machine Learning with Code Embeddings yielded the highest mean *Average F1-score* of **82.55** in *"Within" evaluation setting*. We conducted an ablation study with our best-performing model to investigate the impact of different source code features on the model's performance. Our paper's findings unveil the current status of AI-generated source code detection and propose pathways for improvement.

References

1. ^*ShimS, PatilP, YadavR, ShindeA, DevaleV (2020). "DeeperCoder: Code generation using machine learning." In: 2020 10th annual computing and communication workshop and conference (CCWC). IEEE pp. 0194–0199.*
2. a, b*LuS, GuoD, RenS, HuangJ, SvyatkovskiyA, et al. (2021). "Codexglue: A machine learning benchmark dataset for code understanding and generation." arXiv preprint arXiv:210204664.*
3. ^*DehaerneE, DeyB, HalderS, DeGendtS, MeertW (2022). "Code generation using machine learning: A systematic review." Ieee Access. 10:82434–82455.*
4. ^*SiddiqML, SantosJC (2022). "SecurityEval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques." In: Proceedings of the 1st international workshop on mining software repositories applications for privacy and security. pp. 29–33.*

5. ^ChenM, TworekJ, JunH, YuanQ, Ponde de Oliveira PintoH, et al. (2021). "Evaluating large language models trained on code." arXiv preprint arXiv:210703374.
6. ^LiY, ChoiD, ChungJ, KushmanN, SchrittwieserJ, et al. (2022). "Competition-level code generation with alphacode." Science. 378(6624):1092–1097.
7. ^LuoZ, XuC, ZhaoP, SunQ, GengX, et al. (2023). "Wizardcoder: Empowering code large language models with evol-instruct." arXiv preprint arXiv:230608568.
8. a, b, c, dRenX, YeX, ZhaoD, XingZ, YangX (2023). "From misuse to mastery: Enhancing code generation with knowledge-driven AI chaining." In: 2023 38th IEEE/ACM international conference on automated software engineering (ASE). IEEE pp. 976–987.
9. a, b, c, dLiuM, YangT, LouY, DuX, WangY, et al. (2023). "CodeGen4Libs: A two-stage approach for library-oriented code generation." In: 2023 38th IEEE/ACM international conference on automated software engineering (ASE). IEEE pp. 434–445.
10. a, b, c, d, eYuH, ShenB, RanD, ZhangJ, ZhangQ, et al. (2024). "Codereval: A benchmark of pragmatic code generation with generative pre-trained models." In: Proceedings of the 46th IEEE/ACM international conference on software engineering. pp. 1–12.
11. a, b, cChatGPT: Optimizing language models for dialogue. https://openai.com/blog/chatgpt 2022.
12. a, b, cGoogle gemini. https://gemini.google.com/ 2024.
13. a, b, c, dLozhkovA, LiR, Ben AllalL, CassanoF, Lamy-PoirierJ, et al. StarCoder 2 and the stack v2: The next generation. 2024. Available from: https://arxiv.org/abs/2402.19173
14. a, bRadfordA, NarasimhanK, SalimansT, SutskeverI, et al. (2018). "Improving language understanding by generative pre-training."
15. a, bLiuP, YuanW, FuJ, JiangZ, HayashiH, et al. (2023). "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing." ACM Computing Surveys. 55(9):1–35.
16. a, bGithub copilot. https://github.com/features/copilot 2024.
17. a, b, cLiuJ, XiaCS, WangY, ZhangL (2024). "Is your code generated by chatgpt really correct? Rigorous evaluation of large language models for code generation." Advances in Neural Information Processing Systems. 36.
18. ^MastropaoloA, PascarellaL, GuglielmiE, CiniselliM, ScalabrinoS, et al. (2023). "On the robustness of code generation techniques: An empirical study on github copilot." In: 2023 IEEE/ACM 45th international conference on software engineering (ICSE). IEEE pp. 2149–2160.
19. ^FuY, LiangP, TahirA, LiZ, ShahinM, et al. (2023). "Security weaknesses of copilot generated code in GitHub." arXiv preprint arXiv:231002059.
20. ^AsareO, NagappanM, AsokanN (2023). "Is github's copilot as bad as humans at introducing vulnerabilities in code?" Empirical Software Engineering. 28(6):129.
21. ^YuZ, WuY, ZhangN, WangC, VorobeychikY, et al. (2023). "CODEIPPROMPT: Intellectual property infringement assessment of code language models." In: International conference on machine learning. PMLR pp. 40373–40389.
22. a, b, cGPTZero. https://gptzero.me/ 2023.
23. a, b, cSapling. https://sapling.ai/ai-content-detector 2023.
24. a, b, c, d, e, f, g, h, i, jNguyenPT, Di RoccoJ, Di SipioC, RubeiR, Di RuscioD, et al. (2023). "Is this snippet written by chatgpt? An empirical study with a codebert-based classifier." arXiv preprint arXiv:230709381.
25. a, b, c, d, e, f, g, hPanWH, ChokMJ, WongJLS, ShinYX, PoonYS, et al. (2024). "Assessing AI detectors in identifying AI-generated code: Implications for education." arXiv preprint arXiv:240103676.
26. a, b, c, dFengZ, GuoD, TangD, DuanN, FengX, et al. (2020). "Codebert: A pre-trained model for programming and natural languages." arXiv preprint arXiv:200208155.
27. ^BrownT, MannB, RyderN, SubbiahM, KaplanJD, et al. (2020). "Language models are few-shot learners." Advances in neural information processing systems. 33:1877–1901.
28. a, bWangY, WangW, JotyS, HoiSC (2021). "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation." arXiv preprint arXiv:210900859.
29. ^DaiAM, LeQV (2015). "Semi-supervised sequence learning." Advances in neural information processing systems. 28.
30. a, b, c, d, eGengM, WangS, DongD, WangH, LiG, et al. (2024). "Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning."
31. a, b, cQiG, CaoJ, XieX, LiuS, LiX, et al. (2024). "Exploring the potential of chatgpt in automated code refinement: An empirical study." In: Proceedings of the 46th IEEE/ACM international conference on software engineering. pp. 1–13.
32. a, b, cFengS, ChenC. (2024). "Prompting is all you need: Automated android bug replay with large language models." In: Proceedings of the 46th IEEE/ACM international conference on software engineering. pp. 1–13.
33. a, bGPT-4. (2023). https://openai.com/research/gpt-4.
34. a, bKhouryR, AvilaAR, BrunelleJ, CamaraBM. (2023). "How secure is code generated by ChatGPT?" arXiv preprint arXiv:230409655.

35. ^a, b^PoldrackRA, LuT, BegušG. (2023). "AI-assisted coding: Experiments with GPT-4." arXiv preprint arXiv:230413187.

36. ^a, b^RaneN, ChoudharyS, RaneJ. (2024). "Gemini or ChatGPT? Capability, performance, and selection of cutting-edge generative artificial intelligence (AI) in business management." Capability, Performance, and Selection of Cutting-Edge Generative Artificial Intelligence (AI) in Business Management (February 19, 2024).

37. ^JiangJ, WangF, ShenJ, KimS, KimS. (2024). "A survey on large language models for code generation." Available from: https://arxiv.org/abs/2406.00515.

38. ^a, b^GPT2Detector. (2023). https://github.com/MacroChip/gpt-2-output-dataset.

39. ^a, b^MitchellE, LeeY, KhazatskyA, ManningCD, FinnC. (2023). "Detectgpt: Zero-shot machine-generated text detection using probability curvature." arXiv preprint arXiv:230111305.

40. ^a, b^StrobeltH, GehrmannS. (2019). "Catching a unicorn with gltr: A tool to detect automatically generated text." Catching Unicorns with GLTR.

41. ^a, b^AustinJ, OdenaA, NyeM, BosmaM, MichalewskiH, et al. (2021). "Program synthesis with large language models." arXiv preprint arXiv:210807732.

42. ^a, b^ZhengQ, XiaX, ZouX, DongY, WangS, et al. (2023). "CodeGeeX: A pre-trained model for code generation with multilingual evaluations on HumanEval-x." Available from: https://arxiv.org/abs/2303.17568.

43. ^a, b^HusainH, WuH, GazitT, AllamanisM, BrockschmidtM. (2019). "CodeSearchNet challenge: Evaluating the state of semantic code search." arXiv preprint arXiv:190909436.

44. ^WangK, SinghR, SuZ. (2017). "Dynamic neural program embedding for program repair." arXiv preprint arXiv:171107163.

45. ^ChenZ, MonperrusM. (2018). "The remarkable role of similarity in redundancy-based program repair." arXiv preprint arXiv:181105703.

46. ^WhiteM, TufanoM, MartinezM, MonperrusM, PoshyvanykD. (2019). "Sorting and transforming program repair ingredients via deep learning code similarities." In: 2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER).: IEEE pp. 479–490.

47. ^HarerJA, KimLY, RussellRL, OzdemirO, KostaLR, et al. (2018). "Automated software vulnerability detection with machine learning." arXiv preprint arXiv:180304497.

48. ^PradelM, SenK. (2018). "Deepbugs: A learning approach to name-based bug detection." Proceedings of the ACM on Programming Languages. 2(OOPSLA):1–25.

49. ^BüchL, AndrzejakA. (2019). "Learning-based recursive aggregation of abstract syntax trees for code clone detection." In: 2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER).: IEEE pp. 95–104.

50. ^a, b^ZhangJ, WangX, ZhangH, SunH, WangK, et al. (2019). "A novel neural source code representation based on abstract syntax tree." In: 2019 IEEE/ACM 41st international conference on software engineering (ICSE). pp. 783–794. doi:10.1109/ICSE.2019.00086.

51. ^a, b^DingZ, LiH, ShangW, ChenTHP. (2022). "Can pre-trained code embeddings improve model performance? Revisiting the use of code embeddings in software engineering tasks." Empirical Softw Engg. 27(3). doi:10.1007/s10664-022-10118-5.

52. ^a, b^WangY, LeH, GotmareAD, BuiNDQ, LiJ, et al. (2023). "Codet5+: Open code large language models for code understanding and generation." arXiv preprint arXiv:230507922.

53. ^The top programming languages 2023. (2023). https://spectrum.ieee.org/the-top-programming-languages-2023.

54. ^TIOBE index for march 2024. (2024). https://www.tiobe.com/tiobe-index/.

55. ^CordyJR, RoyCK. (2011). "The NiCad clone detector." In: 2011 IEEE 19th international conference on program comprehension. pp. 219–220. doi:10.1109/ICPC.2011.26.

56. ^Pricing. (2024). https://openai.com/pricing.

57. ^a, b^PengK, DingL, ZhongQ, ShenL, LiuX, et al. (2023). "Towards making the most of chatgpt for machine translation." arXiv preprint arXiv:230313780.

58. ^a, b^NiuC, LiC, NgV, ChenD, GeJ, et al. (2023). "An empirical comparison of pre-trained models of source code." Available from: https://arxiv.org/abs/2302.04026.

59. ^a, b, c, d, e^GuoD, LuS, DuanN, WangY, ZhouM, et al. (2022). "Unixcoder: Unified cross-modal pre-training for code representation." arXiv preprint arXiv:220303850.

60. ^a, b^Tree-sitter. (2024). https://tree-sitter.github.io/tree-sitter/.

61. ^a, b^Awesome-chatgpt-prompts. https://github.com/f/awesome-chatgpt-prompts 2023.

62. ^a, b^IBM global AI adoption index 2022. https://www.ibm.com/watson/resources/ai-adoption 2022.

63. ^a, b, c, d, e, f^Replication package. 2024.

64. ^a, b^GaoS, WenX, GaoC, WangW, ZhangH, et al. (2023). "What makes good in-context demonstrations for code intelligence tasks with llms?" In: 2023 38th IEEE/ACM international conference on automated software engineering (ASE). IEEE pp. 761–773.

65. ^ZhangJ, WangX, ZhangH, SunH, LiuX (2020). "Retrieval-based neural source code summarization." In: Proceedings of the ACM/IEEE 42nd international conference on software engineering. pp. 1385–1397.

66. ^CanedoED, MendesBC (2020). "Software requirements classification using machine learning algorithms." Entropy. 22(9):1057. doi:10.3390/e22091057.

67. ^a, b, c^AlikhashashnehEA, RajeRR, HillJH (2018). "Using machine learning techniques to classify and predict static code analysis tool warnings." In: 2018 IEEE/ACS 15th international conference on computer systems and applications (AICCSA). pp. 1–8. doi:10.1109/AICCSA.2018.8612819.

68. ^Scitools understand. https://scitools.com/ 2024.
69. ^ClementeC, JaafarF, MalikY (2018). "Is predicting software security bugs using deep learning better than the traditional machine learning algorithms?" In pp. 95–102. doi:10.1109/QRS.2018.00023.
70. ^GuptaA, SuriB, KumarV, JainP (2020). "Extracting rules for vulnerabilities detection with static metrics using machine learning." International Journal of System Assurance Engineering and Management. :65–76. doi:10.1007/s13198-020-01036-0.
71. a, bMedeirosN, IvakiN, CostaP, VieiraM (2020). "Vulnerable code detection using software metrics and machine learning." IEEE Access. doi:10.1109/ACCESS.2020.3041181.
72. ^GesiJ, LiJ, AhmedI (2021). "An empirical examination of the impact of bias on just-in-time defect prediction." In: Proceedings of the 15th ACM/IEEE international symposium on empirical software engineering and measurement (ESEM). pp. 1–12.
73. ^AljehaneS, SharifB, MaleticJ (2021). "Determining differences in reading behavior between experts and novices by investigating eye movement on source code constructs during a bug fixing task." In: ACM symposium on eye tracking research and applications. pp. 1–6.
74. ^KatrutsaA, StrijovV (2017). "Comprehensive study of feature selection methods to solve multicollinearity problem according to evaluation criteria." Expert Systems with Applications. 76:1–11.
75. ^AkinwandeO, DikkoHG, AgboolaS (2015). "Variance inflation factor: As a condition for the inclusion of suppressor variable(s) in regression analysis." Open Journal of Statistics. 05:754–767. doi:10.4236/ojs.2015.57075.
76. ^EstevesG, FigueiredoE, VelosoA, ViggiatoM, ZivianiN (2020). "Understanding machine learning software defect predictions." Automated Software Engineering. 27(3-4):369–392. doi:10.1007/s10515-020-00277-4.
77. ^LiJ, AhmedI (2023). "Commit message matters: Investigating impact and evolution of commit message quality." In: 2023 IEEE/ACM 45th international conference on software engineering (ICSE). IEEE pp. 806–817.
78. ^LaValleyMP (2008). "Logistic regression." Circulation. 117(18):2395–2399.
79. ^PetersonLE (2009). "K-nearest neighbor." Scholarpedia. 4(2):1883.
80. a, bRiedmillerM, LernenA (2014). "Multi layer perceptron." Machine Learning Lab Special Lecture, University of Freiburg. 24.
81. ^BreimanL (2001). "Random forests." Machine learning. 45:5–32.
82. ^SuthaharanS, SuthaharanS (2016). "Decision tree learning." Machine Learning Models and Algorithms for Big Data Classification: Thinking with Examples for Effective Learning. :237–269.
83. ^FriedmanJH (2002). "Stochastic gradient boosting." Computational statistics & data analysis. 38(4):367–378.
84. ^ChenT, GuestrinC (2016). "Xgboost: A scalable tree boosting system." In: Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. pp. 785–794.
85. ^JiangN, LutellierT, TanL (2021). "CURE: Code-aware neural machine translation for automatic program repair." In: 2021 IEEE/ACM 43rd international conference on software engineering (ICSE). IEEE. doi:10.1109/icse43902.2021.00107.
86. ^AllamanisM, BrockschmidtM, KhademiM (2018). "Learning to represent programs with graphs." In: International conference on learning representations. Available from: https://openreview.net/forum?id=BJOFETxR-.
87. ^TufanoM, WatsonC, BavotaG, Di PentaM, WhiteM, et al. (2018). "Deep learning similarities from different representations of source code." In: Proceedings of the 15th international conference on mining software repositories. New York, NY, USA: Association for Computing Machinery pp. 542–553. (MSR '18). doi:10.1145/3196398.3196431.
88. ^RuxtonGD (2006). "The unequal variance t-test is an underused alternative to student's t-test and the mann–whitney u test." Behavioral Ecology. 17(4):688–690.
89. ^RosenthalR, CooperH, HedgesL, et al. (1994). "Parametric measures of effect size." The handbook of research synthesis. 621(2):231–244.
90. ^HosseiniS, TurhanB, GunarathnaD (2017). "A systematic literature review and meta-analysis on cross project defect prediction." IEEE Transactions on Software Engineering. 45(2):111–147.

Supplementary data: available at https://doi.org/10.32388/SBL97O

Declarations

Funding: University of California, Irvine

Potential competing interests: No potential competing interests to declare.