Distinguishing AI-Generated Code Through Fine-Tuned Language Models

Niels Van den Broeck
University of Antwerp
Antwerp, Belgium
niels.vandenbroeck@student.uantwerpen.be

Abstract

The rise of AI-generated source code presents new challenges in distinguishing it from human-written code. In this work, we address these challenges by fine-tuning a RoBERTa-based classifier to detect AI-generated Python code. We systematically analyze which code features, such as comments, identifier names, and formatting, contribute the most to model predictions through dataset variations and by performing a stepwise ablation study. In addition, we employ counterfactual testing to interpret model behavior and identify minimal input changes that alter classification outcomes. Our findings show that surface-level features, particularly comments, heavily influence detectability. The final model demonstrates strong generalization across a wide range of variants, providing a solid foundation for robust AI code detection.

1. Introduction

As large language models (LLMs) such as ChatGPT [1] are getting increasingly capable of generating high-quality content, distinguishing between human-written and AI-generated content, and in particular, programming source code, has become a significant challenge. While recent advances have led to the development of classifiers that can detect AI-generated content in natural language with promising accuracy [2–5], there is little to no research focused specifically on the detection of AI-generated source code.

In this work, we address the gap in AI-generated code detection by fine-tuning RoBERTa [6], a publicly available language model, to classify Python code fragments as either human-written or AI-generated. This study is divided into two main parts. In the first part, we focus on improving the classification performance and analyzing which characteristics of AI-generated code most influence the model's decisions. To do this, we systematically modify code samples by removing comments, replacing identifiers, reformatting structure, and altering generation prompts (e.g., instructing the model to "mimic a real person."), and then evaluate how these changes affect the model's predictions. In the second part, we adopt a systematic counterfactual testing approach to further investigate the model's behavior by constructing minimal changes to inputs that could flip the prediction outcome, thereby providing insight into what features the model is most sensitive to.

2. Related work

Current AI-generated content (AIGC) detectors perform well at differentiating between human written and AI generated natural language [2–5]. Although, as shown in [7], their effectiveness in detecting AI-generated source code remains significantly limited. Most existing AIGC detectors are optimized for natural language patterns and fail to generalize to the structural and syntactic characteristics of programming languages. Consequently, these models often produce high false negative rates [1] when applied to generated code, especially when the code is syntactically correct or semantically plausible.

Another paper [8] demonstrates that modifications to code, such as altering prompts, removing comments, or inserting dead code, have limited impact on the effectiveness of current AIGC detectors. Their extensive evaluation of five widely used detectors (GPTZero, Sapling, GPT-2 Detector, DetectGPT, and GLTR) across 13 prompt and code variants reveals that most detectors perform only slightly better than random guessing, with accuracy often hovering around 0.5.

3. Model and Fine-Tuning Setup

To be able to test the possibilities and limits of detecting AI-generated source code, we fine-tuned the publicly available `roberta-base` [9] model for binary classification. RoBERTa was selected for its strong performance in se-

[1] We define AI-generated code as the positive class and human-written code as the negative class. Therefore, a false negative refers to AI-generated code incorrectly classified as human-written. This differs from the labeling used in [7], where human-written code is treated as the positive class. A detailed explanation of our labeling and metrics can be found in Section 4..

quence classification tasks [10] and its architectural advantages over BERT, such as dynamic masking and more robust pretraining. Dynamic masking means that during training, the tokens that are masked change each time an input is processed, allowing the model to see more varied examples and learn better contextual representations. More robust pretraining includes training on larger datasets for longer periods, using optimized hyperparameters, and removing certain training objectives like next sentence prediction, which altogether result in improved language understanding and generalization.

Instead of training a model from scratch, which would require a large amount of domain-specific data to capture the patterns of that data, RoBERTa provides a powerful starting point. The pretraining enables it to recognize a wide range of linguistic features, and fine-tuning on AI-generated and human-written code data refines its ability to identify subtle distinctions between human and AI writing.

By fine-tuning this model on a dataset containing thousands of samples of source code, where each snippet is labeled as either human- or AI-generated, we can adapt it to this new domain and see how well it performs. This approach also fits well within the scope of our research. Since the model is easy to fine-tune, it allows us to focus more on the interesting part: understanding what makes code look AI-generated. Section 7. explains how fine-tuning is conducted in this study through a stepwise ablation approach to identify the most influential code features.

Before fine-tuning, we split the dataset into training and validation sets, using an 80% / 20% ratio, respectively. This is done to ensure that training remains efficient and controlled, while still evaluating the model on a large and diverse set of unseen code snippets. To prepare the data before passing it to the model, we use the AutoTokenizer from Hugging Face. The tokenizer converts code snippets into token sequences, using padding and truncation to ensure consistent input lengths. Tokenization is applied in a batched fashion across all subsets of the data. The model itself is loaded using Hugging Face's AutoModelForSequenceClassification class, with the number of output labels set to 2 (representing human versus AI). For training, we use the Trainer API from Hugging Face's Transformers library. Training is performed over 5 epochs with a batch size of 8 on both training and validation. The reason for only training for 5 epochs [2] is that the model learns extremely fast in most cases, as well as the resource limitation encountered during the research. Validation is performed at the end of each epoch.

4. Evaluation Metrics

In our experiments, we label AI-generated code as 1 and human-written code as 0 (See Figure 1), reflecting the positive detection of AI content. To effectively evaluate the fine-tuned models, we used the following metrics:

TPR: True Positive Rate or **Recall**, calculated as $TPR = \frac{TP}{TP+FN}$ is the proportion of AI-generated code that is correctly classified as AI-generated by the model. TP represents the number of AI-generated code fragments correctly labeled as AI-generated, and TP+FN is the total number of AI-generated code fragments.

Precision: calculated as $TPR = \frac{TP}{TP+FP}$, the amount of correctly classified AI-generated samples, divided by all samples classified by the model as AI-generated.

FPR: False Positive Rate represents the ratio of human-written code fragments that are incorrectly classified by the model, calculated by $FPR = \frac{FP}{TN+FP}$. FP represents the number of human-written codes incorrectly labeled as AI-generated, and TN+FP is the total number of human-written code fragments.

TNR: True Negative Rate, calculated as $TNR = \frac{TN}{TN+FP}$ is the proportion of human-written code that is correctly classified as human-written by the model. TN represents the number of human-written code fragments correctly labeled as human-written, and TN+FP is the total number of human-written code fragments.

FNR: False Negative Rate represents the ratio of AI-generated code fragments that are incorrectly classified by the model, calculated by $FNR = \frac{FN}{TP+FN}$. FN represents the number of AI-generated codes incorrectly labeled as human-written, and TP+FN is the total number of AI-generated code fragments.

Accuracy (ACC): Accuracy is the ratio of correctly classified code fragments to the total number of classified code fragments. This is measured by $ACC = \frac{TP+TN}{TP+FP+TN+FN}$ where TP is the number of human-written codes correctly labeled as human-written, TN is the number of AI-generated codes correctly labeled as AI-generated and TP+TN+FP+FN is the total number of codes.

F1-Score The F1 score is the harmonic mean of precision and recall, providing a single metric that balances both concerns. It is calculated as: $F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$ Unlike accuracy, the F1 score is especially useful in cases of class imbalance, as it accounts for both false positives and false negatives. An F1 score of 1 indicates perfect precision and recall, while an F1 score of 0 suggests that either precision or recall (or both) are 0.

		Predicted	
		AI	Human
Actual	AI	True Positives (TP)	False Negatives (FN)
	Human	False Positives (FP)	True Negatives (TN)

Figure 1: Confusion Matrix used in this research

[2] For the final optimized model, more epochs are trained in section 9.

5. Data Collection

To train and evaluate our AI-generated code detector, we constructed a labeled dataset containing both human-written and AI-generated samples for specific coding tasks.

Python was chosen as the programming language of the training data due to its wide use in both industrial and academic settings [11]. As one of the most popular programming languages, it provides a large pool of real-world code examples. In addition, Python's simple and consistent syntax makes it suitable for machine learning tasks such as code classification. Its structure allows models to more easily capture stylistic and structural patterns, which is crucial to distinguish human-written and AI-generated code.

It is essential for this research that the collected data is of high quality as the model entirely depends on the training data. Without a robust dataset, the results would have little to no practical value. Human-written code samples that are not easily replicated by LLMs can result in inherently imbalanced datasets leading to easier classification. For example, scraping large, real-world, complex code blocks from GitHub repositories as human-written samples is infeasible to prompt an LLM to generate identical outputs without extensive prompting or context. On the other hand, using simple functions for both human-written and AI-generated samples as training data would result in a model that fails to generalize to real-world usage.

Therefore, we decided to select a dataset that contains human-written samples, as well as a description or task that in some way explains how to get to this code. A coding competition suits best for this case, as it includes a brief explanation about the coding problem, which we will be able to use to retrieve the AI-generated samples, and it has tasks of different difficulties and thus various sample lengths. We utilized a publicly available dataset hosted on Hugging Face. This dataset consists of programming problems and their corresponding solutions submitted by participants from various Codeforces contests [12]. Codeforces is a well-known competitive programming site that regularly hosts contests and maintains a public archive of past problems and solutions submitted by users [13].

It is of high importance that our human-written samples do not contain AI-generated code, because this would lead our classifier to train on incorrect samples and thus result in poor performance. Since it is difficult to determine whether a given codeforces submission was assisted by generative AI, a filtering heuristic was applied. Only submissions to contests that took place before 2021 were included. This choice is based on the fact that practical AI code assistants were not publicly available or widely used before that time. However, Codeforces allows users to submit solutions to older contests in "virtual" or practice mode at any time, making it possible that some submissions were still made after 2021 and could have been AI-assisted. Unfortunately, the dataset does not provide the actual submission timestamps, which complicates the usage of a stricter filter.

Furthermore, it is worth noting that the coding style of programmers may have evolved over time due to the influence of AI tools, tutorials, and best practices spread since their release. Therefore, older code used in training may differ in style from modern human-written code, potentially leading to misclassifications when applied to modern code. This highlights an interesting avenue for further research: investigating how evolving programming styles over time affect the reliability and robustness of AI-generated code detectors.

The columns used in this study include:

- **problem-description**: description of the programming problem.
- **input-specification**: A description of the expected input format.
- **output-specification**: A description of the required output format.
- **demo-input**: A sample input illustrating how the problem should be tested.
- **demo-output**: The expected output corresponding to the demo input.
- **note**: Additional clarifications or constraints that apply to the problem.
- **verdict**: The result of executing the source code (e.g. `OK`, `COMPILATION_ERROR`, `WRONG_ANSWER`).
- **code**: The implementation submitted as the solution to the problem.

ChatGPT [1] was chosen as the source of the AI-generated portion in the dataset since it is one of the most used and powerful large language models, making it highly relevant for real-world detection scenarios. ChatGPT is known for its consistent coding style, which provides a strong signal for a classifier to learn from.

We extended the dataset by adding a column representing the AI-generated code samples. To efficiently generate these samples, we utilized OpenAI's batch processing functionality, which allows us to submit multiple prompts at once. A custom script was developed to automate the batching process, manage prompt formatting, and handle the submission and retrieval of results. For each sample in the dataset, we used GPT-4o-mini (via ChatGPT) to generate a Python implementation for the corresponding problem. The prompt used the format given in Figure 2.

```
prompt = (
    "!Only return the Python code in a Markdown block. "
    "Keep the usual amount of comments, but do not add extra explanations
        ↪ outside the code block. "
    "Do not remove comments that would normally be included.! "
    "Write a python script for the following problem: " + str(df.loc[row, '
        ↪ problem-description']) + "\n"
    "Input specification: " + str(df.loc[row, 'input-specification']) + ". "
    "Output specification: " + str(df.loc[row, 'output-specification']) + "\n
        ↪ "
    "Demo input: " + str(df.loc[row, 'demo-input']) + ", "
    "Demo output: " + str(df.loc[row, 'demo-output']) + "\n"
    "Note: " + str(df.loc[row, 'note'])
)
```

Figure 2: Prompt template for AI-generated code.

6. Data Variations

To investigate which characteristics of source code influence model performance, we fine-tuned and evaluated the model on several variations of the dataset. Each variation isolates a different aspect of the code, either by modifying the human-written and AI-generated samples directly or by changing the way the AI-generated samples were generated. This allows us to analyze how specific features, such as formatting, naming conventions, or language model behavior, affect the model's ability to distinguish between human-written and AI-generated code. Some datasets were reduced in size due to modifications made during the data preparation process. For instance, in the correct only dataset, the code samples with syntax errors are discarded, resulting in a smaller dataset. Furthermore, many data sets were limited to 1,000 samples, as they are intended primarily for evaluation purposes instead of training the model. We chose the size of 1,000 samples due to our credit limit of the OpenAI API.

A. *Baseline*

This serves as the reference point, where the human-written and AI-generated code is kept as-is, without any additional modifications. Default prompting (see Listing 2) is performed using OpenAI's GPT-4o-mini model. This choice is motivated by the fact that GPT-4o-mini was the default model available at the time of conducting this research and it was also compatible with our budget. Performance on this dataset reflects the model's raw ability to identify typical LLM-generated output.

Surface-level variants

B. *Removing Comments*

Comments are often written in a generic or overly descriptive style by language models. By removing them, we evaluate how much the model relies on comment structure for classification. We used regular expressions to remove both full-line, as inline comments in the code fragments from the original dataset.

C. *Replacing Identifiers*

Variable and function names can sometimes reveal a lot about the author. Like in text generation, language models tend to use and reuse a specific vocabulary when naming variables and functions. These naming patterns can become easy giveaways that the code was generated rather than written by a human. In this variation, all identifiers are systematically mapped and replaced with single lowercase letters, alphabetically ordered, to test whether the model relies on these naming habits.

D. *Formatting*

Formatting styles, such as spacing, indentation, and line breaks, may introduce obvious patterns when consistently used by LLMs. This variation helps assess how much formatting influences the model's predictions. We applied formatting to all code samples using the `black` code formatter [14]. `black` is a widely used Python code formatter that enforces a standardized style to source code. By using black, we can evaluate whether the classifier depends on formatting artifacts and how its performance changes when those are neutralized.

E. *Passed Tests / Correct Code*

In this setting, we include only code samples that successfully pass their respective test cases. This allows us to see whether functional correctness affects detectability. Due to time constraints, verifying the correctness of all samples in the dataset was infeasible. Luckily, the codeforces dataset contains a verdict column, indicating whether the participants' code passed the test cases on the contest itself. This means that we can gather the correct samples from the human-written dataset, while leaving the AI-generated dataset as-is. Although this seems like an unfair and biased experiment—since we do not check whether the AI-generated samples are correct or contain errors—the main goal is to filter out code submissions that were not serious attempts at solving the contest problem such as joke submissions, code written in other programming languages, or entries that do not contribute to the task. While this may appear to create an imbalance, it does not affect our experiment or goals, which focus on distinguishing AI-generated code from human-written code, not on evaluating functional correctness. Moreover, for these types of competitive programming tasks, modern language models like GPT-4o-mini are quite capable of generating syntactically and semantically plausible solutions [15], making it unnecessary to filter out potentially incorrect AI-generated samples.

Prompt-level variants:

F. *Primed completion*

One common trait of AI-generated code is that models often follow a fixed pattern or "template" when solving problems. For instance, AI-generated code usually starts with the same variable initialization or control structure. To reduce this behavior and generate more diverse, natural-looking code, we use a *priming* strategy. Specifically, before retrieving AI-generated code for a given problem, we prepend the first x lines of the human-written counterpart to the prompt. The number of lines added is calculated as $x = min(max(floor(num\_lines/4), 1), 8)$, which ensures that a meaningful, yet concise context is given.

This priming technique prompts the model to complete the partially written code rather than generate it from scratch. As a result, the output tends to be more variable in structure and potentially harder to detect. By evaluating the classifier on these samples, we can assess whether this strategy makes AI-generated code less distinguishable from human code. An example of a normal prompt and its primed variant is given in Figure 3.

```
"!Only return the Python code in a Markdown block. Keep the usual amount of
      ↪ comments, but do not add extra explanations outside the code block.
      ↪  Do not remove comments that would normally be included!
Write a python script for the following problem: A number is called almost
      ↪ prime if it has exactly two distinct prime divisors. For example,
      ↪ numbers 6, 18, 24 are almost prime, while 4, 8, 9, 42 are not. Find
      ↪  the amount of almost prime numbers which are between 1 and *n*,
      ↪ inclusive.
Input specification: Input contains one integer number *n* (1<=*n*<=3000).
Output specification: Output the amount of almost prime numbers between 1 and
      ↪  *n*, inclusive.
Demo input: ['10\n', '21\n'],
Demo output: ['2\n', '8\n']
Note: none"
```

```
"!Only return the Python code in a Markdown block. Keep the usual amount of
      ↪ comments, but do not add extra explanations outside the code block.
      ↪  Do not remove comments that would normally be included.!
Complete the given code for the following problem: A number is called almost
      ↪ prime if it has exactly two distinct prime divisors. For example,
      ↪ numbers 6, 18, 24 are almost prime, while 4, 8, 9, 42 are not. Find
      ↪  the amount of almost prime numbers which are between 1 and *n*,
      ↪ inclusive.
Input specification: Input contains one integer number *n* (1<=*n*<=3000).
Output specification: Output the amount of almost prime numbers between 1 and
      ↪  *n*, inclusive.
Demo input: ['10\n', '21\n'],
Demo output: ['2\n', '8\n']
Note: none

Complete the following code:
```python

def count_primes(num):
    #does not consider num as a prime number
    count = 0

    for divisor in range(2, int(num**0.5)+1):
        if num % divisor == 0: count += 1
````"
```

Figure 3: Comparison of standard and primed example prompts for the "almost prime number" problem task.

G. *Mimic Person*

In this variant, the prompts were modified to explicitly instruct the language model to mimic human coding behavior. To be precise, "Try to mimic a real person." is added to the prompt. This tests how easily our classification model can be misled when the LLM attempts to imitate human-like unpredictability and variation in coding style.

H. *Prompt model to not include comments*

Programmers tend to prompt LLMs to generate code without comments, since they are usually a giveaway to the human eye that it is generated by AI. With this modification, we investigate the accuracy compared to the post-processing comment removal method (variant *B.*).

I. *Temperature*

The temperature parameter controls the level of randomness in LLMs outputs by scaling the logits before sampling. Lower temperatures (e.g., 0.2) make the model more deterministic, often producing consistent and safe results. Higher temperatures (e.g., 1.3 or 1.7) introduce more variation and unpredictability, leading to more diverse and creative outputs [16]. A study [17] shows that for generated programming code, each 1.0% increase in temperature corresponded to a 0.57% drop in the correctness of the response between temperatures of 0.5 and 1.5.

This variation explores how temperature affects the detectability of AI-generated code. At low temperatures, models reuse patterns, such as repetitive structures, generic variable names, or overly concise logic. These are all traits that a classifier can pick up on more easily. In contrast, high-temperature outputs may deviate from these patterns and better mimic human-like variation.

We prompted the LLM with the following temperatures: 0, 0.3, 0.7, 1(default), 1.3, 1.7 and 2.

J. *Model Variants*

Another dimension of variation is the language model used to generate code samples. We explore how different models influence the detectability of AI-generated code.

OpenAI's **GPT-o3-mini** is a model specifically designed and optimized for reasoning-focused tasks. It tends to focus on logical coherence and structured thinking, which can influence how it writes code, especially for algorithmic problems.

OpenAI's **GPT-4o** represents a more powerful and context-aware model. As the larger sibling of our baseline model `GPT-4o-mini`, it produces code with better structure, naming, and stylistic variation. If the code from `GPT-4o` is harder to detect, it might suggest that larger and more capable models produce more human-like outputs.

By comparing these models, we investigate whether the model architecture and capability impact the detectability of generated code.

K. *Variant Combinations*

Each variant described can be applied individually, but can also be combined to create more complex and realistic variants. In the next section, we describe how these combinations are used in a stepwise manner to discover the impact of each modification.

7. Stepwise Ablation Study

Isolating one aspect at a time could lead to misleading conclusions. Unchanged features may still have strong giveaways when testing the effect of another feature, which could result in a high accuracy either way. This can create the false impression that the tested feature has little to no impact, while in reality, it still has a high contribution to the detection of AI-generated code.

To address this, we perform a stepwise ablation approach. We start with training a model on samples from the baseline variant. In each step, we evaluate the model

on all variants and select the most influential one (i.e., the one that causes the largest drop in accuracy when applied). This variant gets included to the training data along with the previous selected variants. We then retrain the model from start on this expanded set and re-evaluate performance on all variants. This approach can be extended further by training not only on samples of the individual variants (e.g., samples from A and samples B), but also on their combinations (e.g., samples from A+B).

The goal of this method is to discover the impact and influence of code features on the decision-making process of the model, as well as to improve the generalization of the model, making it more resilient and effective in all variants tested.

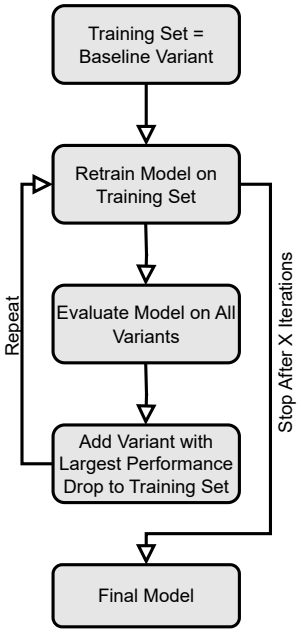

Figure 4: Overview of the stepwise ablation process.

8. Results

8.1. Baseline Performance

Table 1 shows the evaluation results of the classifier trained solely on unmodified baseline samples. Despite being fine-tuned for only 5 epochs on just 1,000 samples, the model achieves near-perfect performance across most variants, which is somewhat surprising given the limited training. When evaluated on the same baseline data on which it was trained, the classifier achieves an impressive 99.3% accuracy with a false negative rate of 0.000 and a false positive rate of 0.014. This high level of accuracy remains the same across nearly all variants.

8.1.1. Impact of Comment Removal

The variants where comments are removed (*removed comments* and *prompt no comments*) produced the worst results. With the surface-level comment removal variant, the model misclassified almost every sample as human-written, resulting in an extremely poor true positive rate and a correspondingly high false negative rate. These results are further illustrated in the confusion matrix in Figure 5, which makes it clear that this model struggles to recognize AI-generated code when comments are absent.

After examining the construction of comments in both human-written and AI-generated code, we found that there is a clear separation between the two. Large language models tend to overuse comments (average of 6.28 comments per sample), which are stylistically verbose, overly explanatory, and redundant. In contrast, the human-written code samples contain far fewer comments (average of 1.53 comments per sample), and those that do appear are often sparse, consisting of commented-out code lines or incomplete sentences. Furthermore, unlike other parts or source code, there are no strict syntactic or stylistic rules for comments themselves. As a result, human comments tend to be highly diverse over different samples, while AI-generated comments follow the same patterns.

This finding indicates the importance of comment structure as a learned feature in distinguishing between human-written and AI-generated code.

		Predicted	
		AI-generated	Human-written
Actual	AI-generated	18	482
	Human-written	3	497

Figure 5: Confusion Matrix for the removed comments variant

8.1.2. Temperature

The temperature used during code generation also has a notable impact on classification performance. As shown in Table 1, the classifier maintains high performance for temperature values ranging from 0.0 to 1.3. In this range, accuracy remains above 98.9%. However, a clear performance drop is observed at higher temperatures. At temperature 1.7, accuracy becomes 96.3%. The decline becomes even more severe at temperature 2.0, where the model achieves only 80.0% accuracy. In Figure 6, the F1 scores are plotted for each temperature setting using the base model trained for 1 epoch. While it holds a great performance for lower temperatures, it drastically drops when the temperature gets close to 2.0.

This dramatic change is the consequence of the behavior of language models at high temperature settings, which increases the randomness of token selection. At extreme temperatures, the model often produces incoherent or nonsensical output that differs significantly from valid source code. In most cases, it generates random character sequences or incorrect syntax that no longer represents or functions as programming code. Because of this, we will not include the evaluation of these variants in the following

Dataset	ACC	F1	Precision	Recall	TNR	FPR	FNR	Loss
baseline	0.993	0.993	0.987	1.000	0.986	0.014	0.000	0.050
passed code	0.993	0.993	0.988	0.998	0.988	0.012	0.002	0.048
removed comments	0.502	0.071	0.864	0.037	0.994	0.006	0.963	2.971
replaced identifiers	0.973	0.973	0.951	0.996	0.951	0.049	0.004	0.193
formatted code	0.969	0.971	0.945	0.998	0.938	0.062	0.002	0.229
complete code	0.984	0.984	0.996	0.972	0.996	0.004	0.028	0.109
prompt no comments	0.496	0.019	0.625	0.010	0.994	0.006	0.990	2.933
mimic person	0.994	0.994	0.988	1.000	0.988	0.012	0.000	0.048
o3mini	0.997	0.997	0.994	1.000	0.994	0.006	0.000	0.019
4o	0.991	0.991	0.986	0.996	0.986	0.014	0.004	0.060
temperature 0.0	0.992	0.992	0.986	0.998	0.986	0.014	0.002	0.053
temperature 0.3	0.990	0.990	0.986	0.994	0.986	0.014	0.006	0.064
temperature 0.7	0.991	0.991	0.986	0.996	0.986	0.014	0.004	0.062
temperature 1.0	0.991	0.991	0.986	0.996	0.986	0.014	0.004	0.059
temperature 1.3	0.989	0.989	0.986	0.992	0.986	0.014	0.008	0.072
temperature 1.7	0.963	0.963	0.986	0.941	0.986	0.014	0.059	0.233
temperature 2.0	0.800	0.758	0.978	0.619	0.986	0.014	0.381	1.257

Table 1: Evaluation results across code variants. The model was fine-tuned on 1,000 samples of unmodified code (RoBERTa, 5 epochs), and evaluated on 1,000 samples per variant.

Note: The temperature 1.0 variant is conceptually identical to the baseline (default temperature), but may produce slightly different results due to independent prompting.

steps of the ablation study.

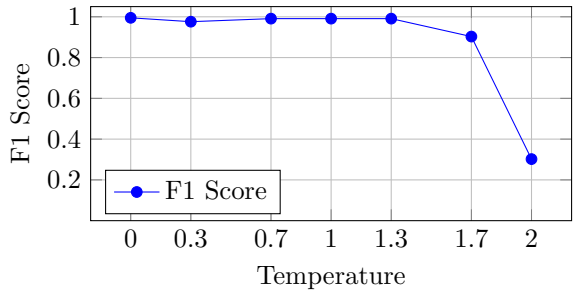

Figure 6: F1 scores of the baseline model (1 epoch) across samples generated using different temperature settings.

8.2. Removing comments

As discussed in previous sections, we cannot simply conclude that comments are the only indicator for distinguishing between human-written and AI-generated code, nor that all other features have no or little influence on the model's prediction. In this subsection, we will train a model using a dataset composed of 50% unmodified baseline samples and 50% samples with all comments removed. This is again fine-tuned for 5 epochs on top of the RoBERTa base model. By doing this, the model will hopefully be more generalized to both code fragments where comments are present and where comments are removed.

During evaluation, we will not only test performance on the original individual variants, but also the combinations of comment removal with other variants. We do this by individually applying all variants to the samples where comments were removed. These generated variants will be named as '`RC + VariantName`' where '`RC`' stands for removed comments, and '`VariantName`' the name or short description of the applied variant. The results are shown in Table 2.

8.2.1. Single variants

Evaluating the model on the original single variants, we observe a slight decrease in accuracy. This is expected, as the model now learned to classify two types of code samples instead of one, increasing the task complexity without any change in training time or computational resources. Originally, the model performed poorly when tested on samples with its comments removed, defaulting to predicting everything as human-written and achieving 50.2% accuracy. Now, the accuracy improves significantly to 93.7%, while the majority of wrongly predicted samples shift toward the false positive rate. This is a great improvement and justifies the small sacrifice in performance on the other variants.

8.2.2. Combined variants

With a model that now generalizing better to code without comments, we can start to analyze the next most impactful features. When looking at the combined variants, the model now struggles when identifiers are replaced with single letters, often misclassifying samples as human-written. This is a solid indication that identifiers play a crucial role in the model's decision-making and are probably the second most informative feature to distinguish AI-generated code from human-written code.

Dataset	ACC	F1	Precision	Recall	TNR	FPR	FNR	Loss
baseline	0.987	0.988	0.975	1.000	0.973	0.027	0.000	0.082
passed code	0.990	0.990	0.980	1.000	0.980	0.020	0.000	0.063
removed comments (RC)	0.937	0.941	0.902	0.984	0.887	0.113	0.016	0.331
replaced identifiers	0.978	0.978	0.960	0.996	0.961	0.039	0.004	0.137
formatted code	0.937	0.942	0.891	1.000	0.870	0.130	0.000	0.366
complete code	0.968	0.969	0.958	0.980	0.955	0.045	0.020	0.210
prompt no comments	0.908	0.909	0.909	0.909	0.907	0.093	0.091	0.545
mimic person	0.990	0.990	0.981	1.000	0.980	0.020	0.000	0.068
o3mini	0.991	0.991	0.983	1.000	0.982	0.018	0.000	0.051
4o	0.987	0.987	0.975	1.000	0.974	0.026	0.000	0.085
RC + passed code	0.943	0.944	0.903	0.990	0.898	0.102	0.010	0.317
RC + replaced identifiers	0.618	0.370	0.949	0.230	0.988	0.012	0.770	2.486
RC + formatted code	0.933	0.938	0.904	0.975	0.887	0.113	0.025	0.384
RC + complete code	0.739	0.706	0.820	0.621	0.860	0.140	0.379	1.527
RC + mimic person	0.949	0.951	0.919	0.986	0.911	0.089	0.014	0.297
RC + o3mini	0.755	0.717	0.864	0.613	0.901	0.099	0.387	1.518
RC + 4o	0.952	0.955	0.916	0.996	0.907	0.093	0.004	0.284

Table 2: Evaluation results for the model from Step 2 of the ablation study. This model was trained on both baseline samples, as samples where comments are removed.

Furthermore, it is also noticeable that the accuracy of the '`RC + complete code`' variant drops significantly. One possible explanation is that the partially human-written code snippet may confuse the classifier, it still relies on the style and structure of that part when making its prediction. Additionally, the AI may not be able to fully impose its typical stylistic patters since it is constrained to completing an existing fragment. As a result, the AI-generated code may seem more human-like.

The usage of a different model, such as o3-mini, seems to have a large impact as soon as comments are removed. This is likely due to the fact that these language models have different architectures and are likely trained on different data. These differences result in distinct generation styles, including differences in formatting, identifier naming, code structure, and even common patterns in logic or syntax. While the classifier is still able to distinguish samples when comments are present, the removal of comments causes AI-generated samples to diverge significantly from samples generated by the baseline model 4o-mini. As a result, the number of false negatives increases. In this experiment, we included the models GPT-o3-mini and GPT-4o to explore whether the choice of generation model would significantly affect classification outcomes. Now that this has been confirmed, we will not include these variants in the remaining steps of the ablation study.

8.3. Replacing Identifiers

Once again, we train a model on a mix of code samples to improve robustness. The dataset is evenly divided: 25% baseline samples, 25% with comments removed (`RC`), 25% with replaced identifiers (`RI`), and 25% where both comments are removed and identifiers are replaced (`RC + RI`). This setup aims to teach the model to recognize AI-generated code under varying transformations. The results are shown in Table 3.

From the results, we observe that the overall accuracy across variants slightly decreases again. However, this trade-off results in a significant improvement on the combined '`RC + RI`' variant, where the model now achieves an accuracy of 92.6%. This indicates that the model has effectively learned to handle samples with both removed comments and replaced identifiers at the cost of a slight drop in performance on simpler variants.

In the previous step, the combination '`RC + complete code`' variant dropped in accuracy. Now, with identifiers replaced as well ('`RC + RI + complete code`' variant), the accuracy decreases once again, this time to 67.8%.

Another notable drop occurs with formatted code, where accuracy on the '`RC + RI + formatted code`' variant falls to 85.5%. This may suggest that consistent formatting patterns are another signal the model uses to differentiate human and AI-generated code.

8.3.1. Limits of Code Modification

As shown throughout this ablation study, modifications such as removing comments and replacing identifiers can significantly impact the classifier's performance. While these experiments are valuable in understanding which features the model relies on, there is an important boundary to acknowledge. With each step, we are modifying the original AI-generated code further, often with the goal of making it harder to detect. However, this raises the question: at what point does modified AI-generated code cease to be meaningfully "AI-generated"?

The more modifications we apply on the AI-generated code, the more they change from what the large language

Dataset	ACC	F1	Precision	Recall	TNR	FPR	FNR	Loss
baseline	0.980	0.981	0.963	1.000	0.959	0.041	0.000	0.104
passed code	0.979	0.979	0.959	1.000	0.959	0.041	0.000	0.100
removed comments (RC)	0.933	0.939	0.887	0.996	0.866	0.134	0.004	0.288
replaced identifiers (RI)	0.958	0.959	0.921	1.000	0.918	0.082	0.000	0.179
formatted code	0.895	0.907	0.831	1.000	0.784	0.216	0.000	0.442
complete code	0.945	0.948	0.911	0.988	0.901	0.099	0.012	0.233
prompt no comments	0.919	0.924	0.882	0.970	0.866	0.134	0.030	0.378
mimic person	0.975	0.976	0.953	1.000	0.949	0.051	0.000	0.098
RC + RI	0.926	0.927	0.894	0.963	0.891	0.109	0.037	0.304
RC + RI + passed code	0.910	0.913	0.864	0.969	0.853	0.147	0.031	0.393
RC + RI + formatted code	0.855	0.875	0.805	0.958	0.739	0.261	0.042	0.623
RC + RI + complete code	0.678	0.638	0.721	0.573	0.782	0.218	0.427	1.632
RC + RI + mimic person	0.905	0.910	0.865	0.960	0.850	0.150	0.040	0.411

Table 3: Evaluation results for the model from Step 3 of the ablation study, trained on baseline samples as well as variants with comments removed, identifiers removed, and both removed.

model would produce in practice. This creates a gray area where the label "AI-generated" still technically applies, but the content no longer resembles actual model outputs. In such cases, we risk training classifiers on unrealistic edge cases that may not be useful to real-world detection tasks.

9. Optimized model

To build a more robust and generalized classifier, we trained a final model using an extended training configuration. The model was trained for 25 epochs (instead of 5), and on a larger dataset of 2000 samples (instead of 1000). The dataset was composed as follows:

- Baseline: 600 samples (30%)
- RC: 500 samples (25%)
- RI: 300 samples (15%)
- RC + RI: 200 samples (10%)
- Complete code: 100 samples (5%)
- RC + RI + Complete code: 100 samples (5%)
- Formatted code: 100 samples (5%)
- RC + RI + Formatted code: 100 samples (5%)

This distribution places the largest weight on unmodified or mildly changed samples (e.g., baseline, removed comments, replaced identifiers). The more complex combinations are intentionally kept limited to both prevent overfitting on edge cases and maintain alignment with realistic AI-generated outputs.

As shown in Table 4, our final model performs well across a wide range of variants. When excluding extreme out-of-distribution variants such as those generated with high temperatures or generated with other models like GPT-o3-mini, the classifier maintains an accuracy well above 85%. These results confirm that the classifier generalizes well to a broad set of modifications, achieving its goal of reliably detecting AI-generated code while accounting for common variations.

10. Counterfactual Analysis

It is crucial to understand what features a model relies on for improving its interpretability. Up to this point, we have focused primarily on identifying "giveaway" patterns and improving our classifier accordingly. In this section, we systematically modify a sample by inserting or removing lines to observe how the model's predictions respond. We begin by experimenting with the effect of AI-generated comments on human-written code, since they have emerged as one of the most significant indicators for detecting AI-generated code. Next, we conduct a powerset-based perturbation analysis on samples to explore how different combinations of lines contribute to the model's decision.

10.1. Effect of Comments on Human-written Code

As established earlier, removing comments from either human-written or AI-generated samples no longer leads to a prediction flip, with accuracy remaining stable at 96.2%. However, we can still check what the impact is when adding AI-generated comments to human-written samples.

The experiment goes as follows: we randomly sample an entry in our programming code dataset, which includes both the human-written and the AI-generated implementation of a specific programming task. Each comment in the AI-generated version is extracted and iteratively inserted at every possible line position within the human-written version. After each insertion, we observe the predicted label of the model and its confidence.

The results of inserting four representative comments into a random sample are shown in Table 5. It is clearly visible that the model is significantly more sensitive to longer, more structured comments than to

Dataset	ACC	F1	Precision	Recall	TNR	FPR	FNR	Loss
baseline	0.999	0.999	0.998	1.000	0.998	0.002	0.000	0.009
passed code	0.996	0.996	0.992	1.000	0.992	0.008	0.000	0.042
removed comments (RC)	0.984	0.985	0.973	0.996	0.971	0.029	0.004	0.140
replaced identifiers (RI)	0.968	0.968	0.940	0.998	0.939	0.061	0.002	0.296
formatted code	0.965	0.967	0.938	0.998	0.930	0.070	0.002	0.298
complete code	0.980	0.980	0.992	0.968	0.992	0.008	0.032	0.187
prompt no comments	0.920	0.919	0.938	0.901	0.939	0.061	0.099	0.736
mimic person	0.994	0.994	0.988	1.000	0.988	0.012	0.000	0.058
o3mini	0.984	0.984	0.986	0.982	0.986	0.014	0.018	0.152
4o	0.991	0.991	0.984	0.998	0.984	0.016	0.002	0.085
temperature 0.0	0.992	0.992	0.984	1.000	0.984	0.016	0.000	0.075
temperature 0.3	0.992	0.992	0.984	1.000	0.984	0.016	0.000	0.075
temperature 0.7	0.992	0.992	0.984	1.000	0.984	0.016	0.000	0.075
temperature 1.0	0.990	0.990	0.984	0.996	0.984	0.016	0.004	0.094
temperature 1.3	0.991	0.991	0.984	0.998	0.984	0.016	0.002	0.085
temperature 1.7	0.804	0.764	0.975	0.628	0.984	0.016	0.372	2.037
temperature 2.0	0.515	0.107	0.784	0.057	0.984	0.016	0.943	5.127
RC + RI	0.929	0.931	0.888	0.977	0.883	0.117	0.023	0.634
RC + RI + passed code	0.911	0.915	0.864	0.971	0.853	0.147	0.029	0.760
RC + RI + formatted code	0.868	0.887	0.810	0.981	0.741	0.259	0.019	1.187
RC + RI + complete code	0.858	0.850	0.892	0.812	0.903	0.097	0.188	1.263
RC + RI + mimic person	0.897	0.903	0.850	0.964	0.830	0.170	0.036	0.962
RC + RI + o3mini	0.831	0.832	0.826	0.837	0.825	0.175	0.163	1.417
RC + RI + 4o	0.904	0.908	0.867	0.954	0.855	0.145	0.046	0.884

Table 4: Evaluation results for the final optimized model.

short or generic ones. For example, inserting the comment "`# Calculate the factorial of A and B using math.factorial`" led to a prediction flip in 7 out of 9 positions and caused an average confidence shift of 0.778 toward the AI label. In contrast, short and generic comments like "`# Read input`" had virtually no effect on the model's prediction or confidence. This suggests that the classifier has learned to associate more descriptive or verbose comment styles with AI authorship. However, relying on stylistic features introduces a vulnerability: the model can be misled by artificially inserting AI-generated comments into human-written code, causing it to wrongly classify the code as AI-generated. One way to address this issue is to remove all comments before evaluation. Alternatively, a new model could be trained on a new variant in which human-written code is intentionally augmented with AI-generated comments. However, this once again raises a similar question: at what point is modified human-written code not human-written anymore?

We also observed that the position of the inserted comment influences the model's behavior. As shown in Figure 7, inserting a comment near the end of the code tends to have more impact on the model's confidence. When inserting the comment `# Calculate the CGD of A! and B!` within the first four lines, the model predicts the sample as human-written with a confidence of approximately 0.6 (i.e., 0.4 for the AI class). When inserting the comment closer to the end, the confidence increases sharply, reaching values between 0.9 and 1.0. This suggests that the model may assign greater importance to comments that appear later in the input.

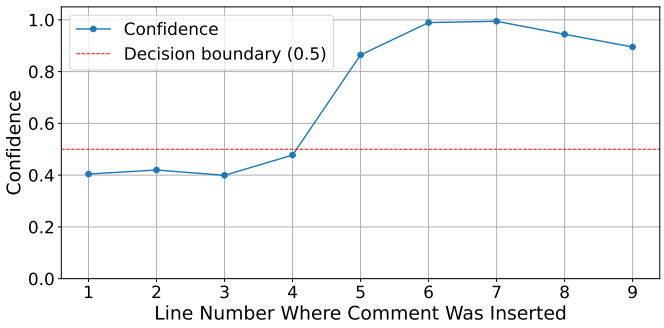

Figure 7: Effect of comment insertion position on model confidence. Confidence values for human-written predictions (lines 1–4) are flipped to illustrate the transition in model behavior. The test was performed by inserting the comment `"# Calculate the GCD of A! and B!"` at different lines on a sample (index 875) from the baseline dataset.

10.2. Line Removal

To better understand which parts or specific lines of an AI-generated code sample contribute most to a model's classification, we applied a powerset-style line removal strategy. In this approach, we systematically generate all

Comment	Flip Count	Avg ΔConf	Final Label
`# Read input`	0/9	0.003	Human
`# Calculate the factorial of A and B using math.factorial`	7/9	0.778	Mostly AI
`# Calculate the GCD of A! and B!`	5/9	0.712	Mixed
`# Print the result`	1/9	0.321	Mostly Human

Table 5: Effect of Comment Insertion on Human-Written Code Sample (baseline dataset at index 875).

Lines remaining	Sample count	Avg Conf	Flipped predictions
10	1	0.995	0
9	10	0.995	0
8	45	0.997	0
7	120	0.995	0
6	210	0.993	0
5	252	0.994	0
4	210	0.991	3
3	120	0.961	22
2	45	0.852	4
1	10	0.849	2

Table 6: Effect of Line Removal on Model Predictions using a sample from the baseline dataset (index 18066).

possible subsets of the code by gradually removing lines in various combinations. This allowed us to observe how the confidence and predictions of the model were affected as more lines were omitted.

Table 6 shows the results of one representative experiment. Although only a single experiment is shown here, multiple runs with different AI-generated samples yielded the same results. Unexpectedly, there were no small subsets of lines whose removal led to a significant drop in classification confidence or a flip in prediction. Instead, the model remained highly confident until approximately 70–100% of the lines had been removed. It is only at this point that prediction flips began to occur with notable frequency.

These findings suggest that the model does not rely heavily on a few isolated features or lines, but rather forms its judgment based on a more holistic understanding of the entire code fragment.

11. Future work

While this research is focused on the viability and robustness of detecting AI-generated code, there are several opportunities for further exploration.

First, the current study focuses on RoBERTa-based binary classification. Exploring alternative model architectures or techniques could offer deeper insight and better performance. For example, the Hugging Face's AutoTokenizer—which is primarily designed for natural language—is used in this study, but is primarily designed for natural language and may not be optimal for code-based tasks. Techniques that incorporate structural representations, such as Abstract Syntax Trees, could provide a more suitable and semantically aware approach for detecting AI-generated code. Furthermore, selecting data from a wider range of sources could make this study more useful for real-world use. The current work relies on code samples from a competitive programming platform, which often results in a specific coding style. Adding data from different domains would expose the model to a wider variety of programming practices and stylistic patterns.

Second, although this study included a brief exploration of different generation models (e.g., GPT-4o, GPT-o3-mini) and decoding strategies (e.g., temperature sampling), deeper evaluation across model domains and settings could help define clearer decision boundaries and test the classifier under broader generalization conditions.

Furthermore, the counterfactual experiments can be expanded with more systematic approaches such as integration with explainability frameworks like SHAP, LIME, or Integrated Gradients. Additionally, the current removal-based method could be extended to edit-based counterfactuals that propose minimal changes to flip predictions while keeping the code functionally correct.

Finally, an important open question remains: when does a modified AI-generated code fragment stop being AI-generated? As the classifier learns from increasingly perturbed samples, future work could investigate the boundary between AI-generated and human-written code.

12. Conclusion

This study explored the effectiveness and possibilities of detecting AI-generated code, focusing on how both surface-level features (e.g., comments, identifier names, formatting) and prompt-level strategies (e.g., mimicking human style or completing partial code) influence classification performance. Through a stepwise ablation study, we identified comments as the most influential indicator of AI-generated code, followed by identifier naming, prompt-

ing techniques like code completion and the underlying language model used. Based on these findings, we developed a classifier that performed well across diverse variants, achieving an average accuracy of 93%. To further interpret the model's decision-making process, we performed a counterfactual analysis, examining how small perturbations, such as adding AI-generated comments or removing specific lines of code, can change predictions and reveal key discriminative signals.

The complete source code is available at: `https://github.com/NielsVandenBroeck/AI_generated_code_detector`

References

[1] OpenAI, "Gpt-4 technical report," 2023. Accessed: 2025-05-10.

[2] L. Mindner, T. Schlippe, and K. Schaaff, *Classification of Human- and AI-Generated Texts: Investigating Features for ChatGPT*, p. 152–170. Springer Nature Singapore, 2023.

[3] Y. Li, Q. Li, L. Cui, W. Bi, Z. Wang, L. Wang, L. Yang, S. Shi, and Y. Zhang, "MAGE: Machine-generated text detection in the wild," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (L.-W. Ku, A. Martins, and V. Srikumar, eds.), (Bangkok, Thailand), pp. 36–53, Association for Computational Linguistics, Aug. 2024.

[4] Y. Zhang, Q. Leng, M. Zhu, R. Ding, Y. Wu, J. Song, and Y. Gong, "Enhancing text authenticity: A novel hybrid approach for ai-generated text detection," in *2024 IEEE 4th International Conference on Electronic Technology, Communication and Information (ICETCI)*, 2024.

[5] A. M. Salem, N. Al-Madi, L. A. Daouk, and A. A. Al-Wabil, "Evaluating the efficacy of ai content detection tools in differentiating between human and ai-generated text," *International Journal for Educational Integrity*, 2023.

[6] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[7] H. Suh, M. Tafreshipour, J. Li, A. Bhattiprolu, and I. Ahmed, "An empirical study on automatically detecting ai-generated source code: How far are we?," 2024.

[8] W. H. Pan, M. J. Chok, J. L. S. Wong, Y. X. Shin, Y. S. Poon, Z. Yang, C. Y. Chong, D. Lo, and M. K. Lim, "Assessing ai detectors in identifying ai-generated code: Implications for education," 2024.

[9] Liu et al., "roberta-base: A pytorch implementation of roberta pretrained model." `https://huggingface.co/FacebookAI/roberta-base`, 2019. Accessed: 2025-04-02.

[10] "Comparative analysis of state-of-the-art qa models: Bert, roberta, distilbert, and albert on squad v2 dataset,"

[11] K. Srinath, "Python–the fastest growing programming language," *International Research Journal of Engineering and Technology*, vol. 4, no. 12, pp. 354–357, 2017.

[12] M. Studio, "Codeforces python submissions." `https://huggingface.co/datasets/MatrixStudio/Codeforces-Python-Submissions`, 2024. Accessed: 2025-02-24.

[13] Codeforces, "About codeforces." `https://codeforces.com/help#q1`, 2025. Accessed: 2025-05-13.

[14] Łukasz Langa *et al.*, "Black: The uncompromising python code formatter." `https://github.com/psf/black`, 2018. Accessed: 2025-01-26.

[15] C. E. A. Coello, M. N. Alimam, and R. Kouatly, "Effectiveness of chatgpt in coding: A comparative analysis of popular large language models," *Digital*, vol. 4, no. 1, pp. 114–125, 2024.

[16] M. Renze, "The effect of sampling temperature on problem solving in large language models," in *Findings of the Association for Computational Linguistics: EMNLP 2024* (Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, eds.), (Miami, Florida, USA), pp. 7346–7356, Association for Computational Linguistics, Nov. 2024.

[17] A. Tophel, L. Chen, U. Hettiyadura, and J. Kodikara, "Towards an ai tutor for undergraduate geotechnical engineering: a comparative study of evaluating the efficiency of large language model application programming interfaces," *Discover Computing*, vol. 28, p. 76, May 2025.