StackPlagger: A System for Identifying AI-Code Plagiarism on Stack Overflow

Aman Swaraj, Harsh Goyal, Sumit Chadgal, Sandeep Kumar
Department of Computer Science and Engineering, Indian Institute of Technology Roorkee, India
{aman_s, harsh_g1, sumit_c, sandeep.garg}@cs.iitr.ac.in

***Abstract*—Identifying AI code plagiarism on technical forums like Stack Overflow (SO) is critical, as it can directly impact the platform's trust and credibility. While previous studies have explored AI-generated code detection, they have focused on long, standalone samples from repositories and competitions. In contrast, SO snippets are often short, fragmented, and context-specific, which can make detection more challenging. Furthermore, existing methods have also not adequately addressed the concern of obfuscated or adversarially prompted code that are crafted to mimic human style and evade detection. To address these gaps, we first introduce a curated dataset of 8000 SO-ChatGPT snippet pairs generated using multiple adversarial prompts. While earlier methods solely relied on pre-trained models, we propose an ensemble approach combining stylometric features of code along with the pre-trained embeddings to improve detection performance. Finally, we deploy our fine-tuned model as a Google Chrome extension called 'StackPlagger', which can flag AI-generated code in SO answers and display AI confidence scores. Video demonstration and the associated artifacts of our tool can be found at https://youtu.be/6O9Urp2mvbI and https://github.com/harsh-g1/StackPlagger, respectively.**

***Index Terms*—AI-Code Detection, AI plagiarism, Chrome Extension, Code Stylometry, Code LLMs, Prompt Engineering**

I. Introduction

A. Background

Stack Overflow (SO) has long been the go-to platform for developers seeking technical assistance. Despite the emergence of AI tools like ChatGPT, developers still actively rely on SO, largely due to its peer-reviewed answers, and community-driven reputation system[1].

However, this transparency has recently come under threat owing to the increasing presence of AI-generated answers on the site. Besides authorship attribution, the issue also concerns the reliability of the answers as earlier research has shown that AI responses can be flawed or misleading [1].

Given these challenges, it has become essential to come up with relevant measures for detecting AI-generated code on such technical platforms.

The current state-of-the-art works in AI code detection have mostly involved pre-trained models or zero-shot methods. For instance, Nguyen et al. [2] used CodeBERT, Xu et al. [3] applied a contrastive learning approach with UnixCoder, and Gurioli et al. [4] employed a CodeT5+ model for detection. Similarly, with regards to zero-shot methods, two notable works comprise Zhenyu et al. [5] presenting a framework based on targeted perturbation and perplexity scoring and Yang et al. [6] proposing a method utilizing a surrogate model for approximating token probabilities.

[1]https://survey.stackoverflow.co/2024/developer-profile#2-online-resources-to-learn-how-to-code

B. How We Overcome Limitations in Prior Works

While these studies have provided useful insights, they suffered from certain limitations. First and foremost, these studies have mostly focused on long and complete code samples taken from programming tasks and GitHub repositories. In contrast, code snippets found on SO posts are often short in length and contextual to the specific questions, which can make the distinction more difficult.

These characteristics make the detection task notably more challenging compared to earlier settings, and at the same time, more representative of real-world scenarios. In this work, we have focused on SO snippets and thereby addressed this limitation.

Secondly, with regards to methodology, we observed that existing methods have primarily relied on pre-trained models; but have overlooked the contribution of traditional stylometric features of code such as whitespace usage or Abstract Syntax Tree (AST) node structures. Although code language models can themselves encode certain stylistic characteristics, their predictions primarily depend on the latent tokens and sequence-level representations rather than the stylometric cues. However, these features, which are often referred to as an 'author's fingerprint', have long played a central role in tasks related to code authorship attribution [7], [8] and have also been attributed to being more robust to code obfuscation attempts [7].

We have addressed this limitation by incorporating stylometric features, such as lexical, structural, and layout-level patterns of code. Given the conceptual similarity between code authorship attribution and our task of distinguishing human-AI code, we tap these stylistic cues to complement deep code embeddings and improve classification.

Next limitation concerns the aspect of adversially prompted AI code, where prompts are carefully crafted to imitate human coding style and skip detection. Several studies have shown that with such sort of targeted prompting and minor edits, distinguishing AI code from human becomes more difficult [2], [5], [9]. This concern again becomes more significant for developer forums like SO, where accepted answers can be used to earn reputation points. Yet, despite such concerns, current

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00386

research has not sufficiently addressed the issue of adversarial AI code plagiarism.

We again address this challenge by first generating a benchmark dataset of obfuscated prompts and then evaluating several leading pre-trained models over these newly generated sets of adversarially prompted AI code. While some prior studies have explored adversarial prompting, our study specifically investigates how these attacks influence detection in community-driven platforms like SO. Moreover, as stylometric features have been attributed to offering greater robustness against adversarial attacks [7], we evaluate their effectiveness in this context and open a new research direction towards exploring such ensemble approaches for further code analysis tasks.

Another limitation found in previous studies was that they utilized older versions of ChatGPT, such as 3.5. We overcome this limitation by using the latest version of ChatGPT, 4O-mini, for generating the dataset and thus ensuring practical relevance.

Finally, we release our best performing model in the form of a Google Chrome extension, 'StackPlagger', that can detect AI-generated code on live SO pages. To the best of our knowledge, no prior work has presented such a real-time tool for SO. This extension not only makes our research directly usable by developers but also creates opportunities for integration on other platforms facing similar challenges.

Given the rapid pace of advancement in generative AI and the specific gaps mentioned earlier, we believe our system StackPlagger can provide a step forward towards the challenges of AI code detection. The main contributions of our work can be listed as follows:

1) An Ensemble strategy integrating stylometric features with pre-trained embeddings to enhance the identification of AI-generated code, especially in obfuscated settings.
2) A newly curated dataset consisting of 3,500 human-authored SO code snippets paired with AI-generated counterparts, along with 4,500 additional adversarially crafted AI responses to support real-time plagiarism analysis.
3) A practical browser extension, StackPlagger, that can flag AI-generated code directly on live Stack Overflow pages, helping SO users and moderators alike.

II. Design and Development of StackPlagger

In this section, we outline the key steps of our proposed approach as highlighted in Figure 1.

A. Data Preparation

Following prior work, we focused on two of the most widely used languages, Python and Java, and selected questions with high upvotes and accepted answers to ensure relevance and reliability. We extracted 1,750 questions for each language, resulting in a total of 3,500 question–answer pairs. For the corresponding AI responses, we used ChatGPT, as it is the most widely used AI tool by developers [2], and has been considered in earlier works concerning AI code detection as well. Further, we specifically employed the 4O-mini version owing to its free-of-cost availability, where many developers face the challenge of buying paid AI tools [3].

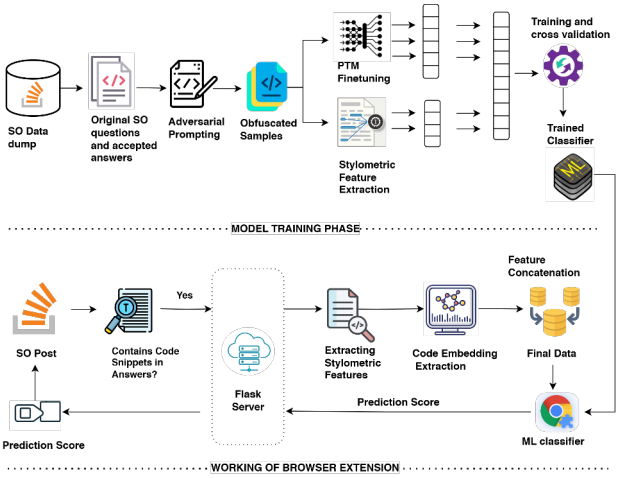

Fig. 1. Workflow of our StackPlagger Tool

For our adversarial sets, we designed nine obfuscation strategies, ranging from renaming variables and modifying formatting to mimicking beginner-like or overly confident styles. These prompts were carefully crafted to generate human-like behavior, such as verbosity, informal tone, minor errors, formatting inconsistency, and other coding practices observed on Question-Answer platforms. Next, we selected a subset of 500 questions from our original dataset and applied these nine prompts to each, resulting in 4,500 adversarial samples. This gave us a final corpus comprising 3,500 human-written and 8,000 AI-generated code snippets. All the data, along with the prompt details and SO SQL query, can be found in our online repository[4].

B. Feature Extraction from Pre-trained Code Models

We selected three leading code models for our task, namely, CodeBERT [10], CodeT5+ [11], and UnixCoder [12], owing to their strong performance in code-related tasks and their prior application in AI code detection studies [2]–[4].

To have a comparative analysis of our ensemble approach with the base models, we fine-tuned them in two different settings. First, these models were trained to act as direct classifiers. In contrast, for our StackPlagger approach, we extracted intermediate embeddings from the layer before the classification head, and subsequently combined them with the stylometric features before final classification.

[2]https://survey.stackoverflow.co/2024/technology/#1-ai-search-and-developer-tools
[3]https://survey.stackoverflow.co/2024/ai/#efficacy-and-ethics
[4]https://github.com/harsh-g1/StackPlagger

C. Leveraging Stylometric Features

Guided by prior research, we extracted three broad categories of stylometric features: lexical, syntactic, and layout-based. While lexical features included statistics such as average line length, token count, and use of comments, syntactic features were derived from abstract syntax trees (ASTs), and finally, the layout-based features comprised whitespace usage, indentation patterns, and blank lines.

While we referred to the work by Caliskan et al. [7] to guide our feature selection, we excluded certain language-specific features that did not generalize well across both Java and Python, ensuring better relevance to our dataset. The complete set of stylometric features is available in our public repository.

D. Feature Concatenation and Model Fitting

After extracting relevant stylistic features, we concatenated them with embeddings tuned from the PTMs to form a unified feature set. This combined representation was then sent for training various ML classifiers.

We experimented with a diverse set of models, which included Random Forest, XGBoost, SVM, Naive Bayes, and a feed-forward Artificial Neural Network (ANN). We selected these classifiers based on their strength and prior use in existing AI code detection tasks [13]. We utilized grid search with cross-validation for hyperparameter tuning, the details of which can be found in our repository. Following performance comparisons, we selected the best-performing model and saved it as a pickle file, which was then integrated at the backend of our Chrome extension.

E. Working of the Browser Extension

The working of the StackPlagger extension is depicted in Figure 1. Since our work is limited to Java and Python languages, the extension first scans the programming language tag of the post and proceeds accordingly. Next, it verifies the presence of code snippets within all the posted answers and then extracts them individually and sends them to the backend via a Flask server API.

At the backend, stylistic features and code embeddings are extracted and combined in the same manner that was carried out during the training pipeline. The final feature set is then passed to the trained classifier, which is present there in the form of a pickle file. Instead of generating a binary label, StackPlagger generates a confidence score to better visualize the nuances of the classification. This score is then sent back through the Flask API and displayed on the front-end as a badge next to each code snippet.

III. Tool Evaluation and Comparative Analysis

To evaluate performance in a reliable manner, we constructed a dedicated test set using 500 SO questions for which we had earlier generated the adversarial code variants. These subsets were isolated from the training process to prevent data leakage. We created a total of ten test sets where the human response to each of these 500 questions was paired with the ten sets of AI responses, i.e., the original ChatGPT output and nine

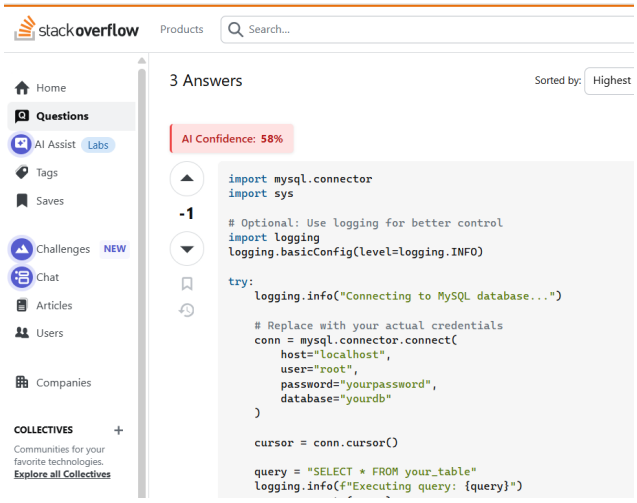

Fig. 2. Snapshot of the working tool, StackPlagger on a sample SO post answer code snippet.

adversarially prompted variants. This resulted in ten separate test folds, each aligned with a specific type of adversarial prompt and containing a consistent set of human–AI code pairs. This design enabled us to systematically compare how different adversarial strategies affected detection performance.

All ten test sets were evaluated over the PTMs and classifiers specified in the earlier section. For each configuration, five independent seeds were run, and the average accuracy scores were recorded to ensure stability. A comparative analysis of our work based on these accuracy measures alongside a screenshot of the working tool in Figure is presented in Table I and 2 respectively.

A. Performance of PTMs over Adversarial Snippets

The first aspect of our analysis was aimed at assessing the capabilities of existing code language models in differentiating AI and human code in the context of SO under normal and adversarial settings. As shown in Table I, all three PTMs showed good results on the first test set, which included human-written responses and the original ChatGPT outputs. However, their performance declined across the adversarial test sets (from 2 to 10), indicating that adversarial prompting does reduce detection chances.

We can also observe that certain prompts caused a more significant drop in detection performance than others across all the models. In particular, prompts such as Prompt 7 (Conversational & Opinionated Answer), Prompt 8 (Slightly Imperfect, Human-Like Code), caused the highest drop. One possible reason could be that they typically introduced more human-like characteristics, while simple code-level changes, such as removing indentation or changing variable names caused relatively moderate deductions. Interestingly, Prompt 5 (Code Blending) showed only a minor performance difference. A possible reason could be inferred that even when AI-generated code is partially mixed with human-written snippets,

the presence of little AI content can still provide enough information for PTMs to distinguish them from purely human-authored code.

Overall, the variation in performance across different prompts goes to show that some prompting strategies more closely mimic human-written code, while others still leave detectable traces. More importantly, these findings highlight the need for prompt-aware testing when creating AI code detectors and open promising directions for future research.

B. StackPlagger Vs Existing Approaches

The second phase of our analysis was to evaluate the effect of combining stylometric features along with the code embeddings. As shown in Table I, we report the accuracy scores for each code language model, both with and without stylometric integration. While the table contains results of the Random Forrest classifier (for the case of stylometric features), the extended results of the remaining classifiers and metrics can be found at our online repository.

Besides, Table I not only compares variants of our approach but also benchmarks it against prior work, which primarily relied on pretrained models without stylometric features. By aligning our baselines with these existing methods, we highlight the added value of our enhanced approach as well. We can see from the table that across all ten test sets, including the adversarial ones, integrating stylometric features consistently improves performance over using model embeddings alone. These results support our hypothesis that stylometric cues, such as lexical attributes, layout patterns, and structural constructs, contribute to the resilience against adversarial changes. This is also in line with earlier studies where stylometric features were found to have more stability against obfuscation attempts [7].

TABLE I
ACCURACY ACROSS PROMPT SETS (S1–S9) FOR BASELINE MODELS VS STACKPLAGGER-ENHANCED VERSIONS (ACCURACY)

PType	**CodeBERT**		**UnixCoder**		**CodeT5+**	
	[2]	Ours	[3]	Ours	[4]	Ours
Set 1	0.80	**0.84**	0.87	**0.90**	0.82	**0.86**
Set 2	0.69	**0.73**	0.81	**0.83**	0.77	**0.80**
Set 3	0.67	**0.70**	0.81	**0.83**	0.73	**0.76**
Set 4	0.68	**0.71**	0.79	**0.82**	0.72	**0.75**
Set 5	0.66	**0.70**	0.77	**0.80**	0.76	**0.79**
Set 6	0.79	**0.82**	0.86	**0.89**	0.81	**0.85**
Set 7	0.67	**0.70**	0.77	**0.79**	0.70	**0.73**
Set 8	0.58	**0.62**	0.71	**0.74**	0.68	**0.72**
Set 9	0.68	**0.71**	0.76	**0.79**	0.70	**0.73**

IV. IMPLICATIONS AND THREATS

Our work has direct implications for SO users and moderators, as it provides them a basis for making more informed decisions about code credibility and authorship. Regarding threats, one major concern is the subjectivity involved in crafting adversarial prompts. To mitigate this risk, we followed guidelines from prior work and designed prompts to resemble common human coding behavior. Another limitation is our focus on only two programming languages, i.e, Python and Java. While this may restrict the generalizability of our findings, these languages are not only widely used but also represent diverse syntactic structures and have been used in previous works.

V. CONCLUSION

In this work, we have presented StackPlagger, a real-time tool for identifying AI-code plagiarism on SO pages. Our work introduces a curated benchmark dataset encompassing both regular and obfuscated code snippets. Through extensive experiments, we show that existing pre-trained models perform well on regular samples but struggle in adversarial settings. To address this, we propose an ensemble approach that integrates stylometric features with code PTM embeddings, leading to improved detection accuracy. Finally, we deploy our best-performing model as a lightweight Chrome extension that highlights the AI confidence score of code snippets directly on SO pages, helping users and community moderators make informed decisions. Overall, our work highlights the need for prompt-aware evaluation and contributes toward maintaining authenticity and trust in community-driven programming forums.

ACKNOWLEDGMENT

We thank Harshit Kumar, Kandarp Singh, Lakshya Kataria, and Merugu Puneet for assistance with data curation.

REFERENCES

[1] Borji, Ali. "A categorical archive of chatgpt failures." arXiv preprint arXiv:2302.03494 (2023).

[2] Nguyen, Phuong T., et al. "GPTSniffer: A CodeBERT-based classifier to detect source code written by ChatGPT." Journal of Systems and Software 214 (2024): 112059.

[3] Xu, Xiaodan, et al. "Distinguishing LLM-generated from Human-written Code by Contrastive Learning." ACM Transactions on Software Engineering and Methodology 34.4 (2025): 1-31.

[4] Gurioli, Andrea, Maurizio Gabbrielli, and Stefano Zacchiroli. "Is this you, llm? recognizing ai-written programs with multilingual code stylometry." 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2025.

[5] Xu, Zhenyu, and Victor S. Sheng. "Detecting AI-generated code assignments using perplexity of large language models." Proceedings of the aaai conference on artificial intelligence. Vol. 38. No. 21. 2024.

[6] Yang, Xianjun, et al. "Zero-shot detection of machine-generated codes." arXiv preprint arXiv:2310.05103 (2023).

[7] Caliskan-Islam, Aylin, et al. "De-anonymizing programmers via code stylometry." 24th USENIX security symposium (USENIX Security 15). 2015.

[8] Dauber, Edwin, et al. "Git blame who? stylistic authorship attribution of small, incomplete source code fragments." Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. 2018.

[9] Pan, Wei Hung, et al. "Assessing ai detectors in identifying ai-generated code: Implications for education." Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training. 2024.

[10] Feng, Zhangyin, et al. "Codebert: A pre-trained model for programming and natural languages." arXiv preprint arXiv:2002.08155 (2020).

[11] Wang, Yue, et al. "Codet5+: Open code large language models for code understanding and generation." arXiv preprint arXiv:2305.07922 (2023).

[12] Guo, Daya, et al. "Unixcoder: Unified cross-modal pre-training for code representation." arXiv preprint arXiv:2203.03850 (2022).

[13] Bukhari, Sufiyan, Benjamin Tan, and Lorenzo De Carli. "Distinguishing AI-and human-generated code: A case study." Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses. 2023.