Is This You, LLM? Recognizing AI-written Programs with Multilingual Code Stylometry

Andrea Gurioli
DISI
University of Bologna
Bologna, Italy
andrea.gurioli5@unibo.it

Maurizio Gabbrielli
DISI
University of Bologna
Bologna, Italy
maurizio.gabbrielli@unibo.it

Stefano Zacchiroli
LTCI, Télécom Paris
Institut Polytechnique de Paris
Palaiseau, France
stefano.zacchiroli@telecom-paris.fr

***Abstract*—With the increasing popularity of LLM-based code completers, like GitHub Copilot, the interest in automatically detecting AI-generated code is also increasing—in particular in contexts where the use of LLMs to program is forbidden by policy due to security, intellectual property, or ethical concerns.**

We introduce a novel technique for *AI code stylometry*, i.e., the ability to distinguish code generated by LLMs from code written by humans, based on a transformer-based encoder classifier. Differently from previous work, our classifier is capable of detecting AI-written code across 10 different programming languages with a single machine learning model, maintaining high average accuracy across all languages (84.1% $\pm 3.8\%$).

Together with the classifier we also release H-AIRosettaMP, a novel open dataset for AI code stylometry tasks, consisting of 121 247 code snippets in 10 popular programming languages, labeled as either human-written or AI-generated. The experimental pipeline (dataset, training code, resulting models) is the first fully reproducible one for the AI code stylometry task. Most notably our experiments rely only on open LLMs, rather than on proprietary/closed ones like ChatGPT.

***Index Terms*—code stylometry, large language models, AI detection, code generation, data provenance, deep learning**

I. Introduction

LLM-based code completers [8], [24], [39] (or *code LLMs* for short in this paper), as exemplified by GitHub Copilot[1], are becoming popular automatic programming tools among software developers. Preliminary evaluations of code LLM results show that they can produce either correct or buggy code [11], [40], depending on how they are used. Specifically, code LLMs can be useful assets for expert programmers who quickly learn to use them well or a liability for novice developers who lack the experience to skip misleading answers quickly. The real impact of code LLMs on developer productivity also remains unclear, with growing interest in defining proper metrics to evaluate it [1].

Similarly, policy-wise, the use of code LLMs can be frowned upon or outright forbidden, depending on the context. Security- and privacy-sensitive environments might forbid the use of code LLMs hosted by 3rd parties—like Copilot, hosted by GitHub, or ChatGPT[2] by OpenAI—to avoid leaking internal code in prompts. (Self-hosted open-weight LLMs, like Code Llama [39] and StarCoder [24] mitigate this issue.) In teaching contexts, such as schools and universities, the use of code LLMs can be considered cheating (depending on the assignment goals), with severe consequences for the students who use them [18].

Legal and licensing risks are also ongoing concerns when using code LLMs [38]. Even leaving aside the hot legal topic of whether training LLMs on third-party unlicensed material is allowed (or ethical), code LLMs can output verbatim parts of their training datasets, a phenomenon known as *recitation* [44], which might expose their *users* to legal liabilities [7] if generated code is integrated into a product put on the market.

A. Problem statement

These practical needs have spawned an interest in *automatically recognizing code generated by LLMs*, distinguishing it from code written by humans. This is an instance of the more general task of automatically detecting *who* wrote a given piece of code, known in the literature as *code stylometry* (or *code authorship attribution*, or *code author recognition*) [6], [20], [34].

Previous work [7], [18], [19], [23], [33] has already applied code stylometry techniques to the recognition of "AI authors" (i.e., code LLMs), with three recurrent characteristics: (1) detection is possible on a single programming language *at a time*; (2) the tested code LLM is a proprietary, non-open tool or model (e.g., Copilot, or ChatGPT), which hinders scientific reproducibility and replicability; (3) detection is based on traditional machine learning techniques (e.g., random forest classifications, LSTM, or code2vec embeddings).

The goal of this paper is to improve the state-of-the-art of the detection of AI-written programs, by addressing the following research question:

> ***RQ1****: Is it possible to detect source code generated by code LLMs, achieving high accuracy across several different programming languages?*

Answering this question affirmatively would improve over limitation (1) above, which can be particularly annoying in projects where multiple programming languages are in use, as it is often the case. Methodologically, we aim to answer RQ1 following a fully reproducible experimental approach (addressing limitation 2 above) and using more recent machine learning techniques (point 3 above) that, as we will see, are

[1] https://github.com/features/copilot/, accessed 2024-09-24
[2] https://openai.com/chatgpt/, accessed 2024-09-24

needed to achieve good results in the desired multilingual setting.

B. Contributions

With this paper we make the following novel contributions:

1) We release a novel, balanced, open **dataset for the AI code stylometry task**, which contains 121 247 code snippets in total, written in 10 different popular programming languages: C++, C, C#, Go, Java, JavaScript, Kotlin, Python, Ruby, Rust. Each snippet is labeled as either having been authored by a human (to solve a specific task in the context of the Rosetta Code project [9]) or as generated by StarCoder2 [27] (a state-of-the-art open code LLM), via code translation from a (human-authored) snippet written in a different programming language to solve the same task.
2) We train a **transformer-based encoder classifier**—a novel architecture for the AI code stylometry task—on the above dataset. Using it we answer the stated research question affirmatively, showing that it is possible to recognize multilingual AI-generated code, across 10 popular programming languages, with an average accuracy of 84.1% ($\pm$ 3.8%).
3) We release an **open source tool** based on the trained classifier, which can be used to detect whether code snippets of interest have been AI-generated or not. The tools is available both as a hosted version on Hugging Face[3] and as a command line (CLI) tool distributed with this paper replication package.
4) All our experiments are **fully reproducible**: the initial dataset is openly available and can be regenerated using Rosetta Code data and StarCoder2; the training and evaluation pipeline is available as part of the replication package of this paper (see the Data availability statement at the end).

II. Methodology

The experimental methodology followed for this work is depicted in Figure 1. It consists of two parts: (1) dataset construction, described next, leading to the creation of the H-AIRosettaMP dataset; (2) model training, described in Section II-B, leading to multiple classifier models, whose performances are analyzed in Section IV.

A. Dataset construction

Our goal is to train a machine-learning classifier that can distinguish human-written from AI-generated code, across several programming languages (RQ1). To that end we need first and foremost a training dataset, covering multiple languages, and containing snippets labeled as either human-written or AI-generated. To the best of our knowledge such a dataset did not exist before, so we set to create one. While doing so, we pursued the methodological goals of making its construction fully reproducible and releasing it open data.

[3]https://huggingface.co/spaces/isThisYouLLM/Human-Ai

1) Programming language selection: As a starting point for human-written code snippets, we used Rosetta Code [9], a programming chrestomathy project that collects and publishes solutions to the same programming tasks in as many different languages as possible, to showcase similarities and differences across languages. We retrieved a version of the Rosetta Code timestamped as July 1st, 2022. The retrieved dataset contained 79 013 code snippets, each representing a solution to one among 1203 programming tasks in total, written in one among 883 programming languages.

In order to both respond to real-world use cases and maximize data availability for the later training phase, we selected our target programming languages for AI code stylometry based on their popularity. To rank languages by popularity, we retrieved the TIOBE index [35] ranking, as of May 2024. From the TIOBE ranking we removed all languages not present in the training dataset of the open code LLM used in our experiments, namely StarCoder2-15B [27] (see later in this section for a discussion of our choice of LLM). To conclude this step (*Popular languages filtering* in Figure 1), we selected the top-10 remaining languages by ranking order—10 languages being a very significant step forward w.r.t. the state of the art of AI code stylometry performed on at most 2 languages at a time.

We hence obtained a set of 10 popular programming languages, together with hand-written snippets in those languages from Rosetta code, that are also well-known to the code LLM used later to generated AI-authored snippets: C++, C, C#, Go, Java, JavaScript, Kotlin, Python, Ruby, Rust.

Before dwelling into details, here are the two key intuitions behind the next steps:

1) *Human-written snippets* in the target dataset are unmodified snippets coming from Rosetta Code: they were all contributed as task solutions by humans participating in the initiative.
2) *AI-written snippets* in the target dataset are generated by a code LLM (specifically: StarCoder2 [27]) using *cross-language translation* form a source programming language *src* (called the *provenance language* in the following) to a destination language *dst*, as previously done by Li et al. [23]. Input to the translation is a human-written snippet coming from Rosetta Code (as per (1) above); output of the translation is an AI-written snippet (by StarCoder2) that will be integrated into the target dataset. Details on the translation step are provided later in this section.

2) Task balancing: To avoid skewing the AI-written part of the dataset by translating from a single source programming language (which might be more affine to one target language than another), all *human-written* snippets in the dataset for a given programming language *src* have been translated to *all other 9 languages* among the 10 selected languages. This constitute a total of 90 ($= 10 \times 9$) sub-datasets, each formed by a $\{src, dst\}$ unordered language pair, where $src \neq dst$.

The initial 90 (sub-)datasets (before the *Translation* step

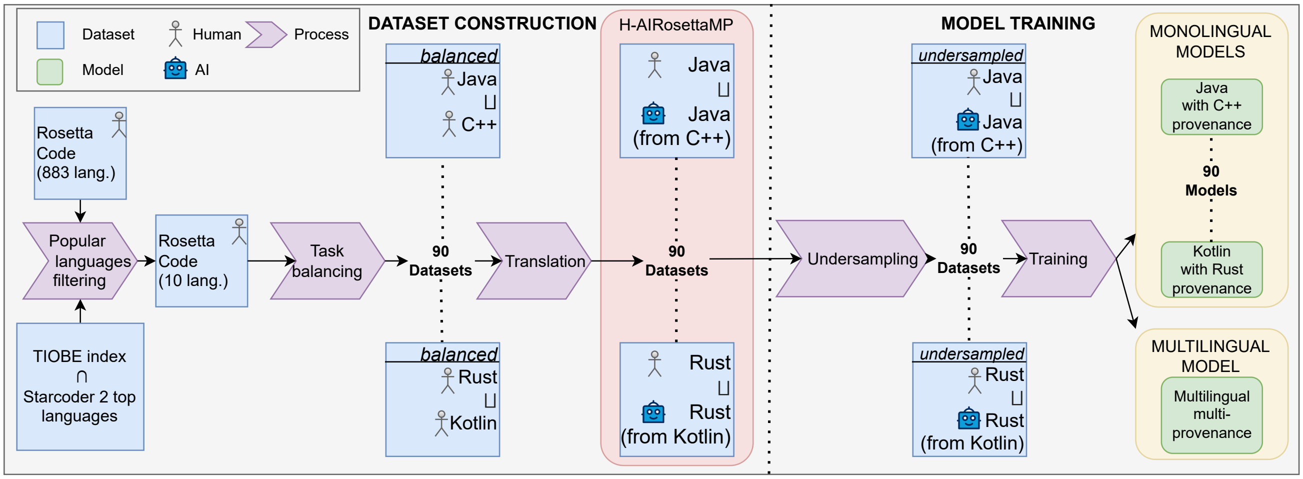

Fig. 1. Experimental methodology. The process is divided into two main steps: (1) The *Dataset construction*, which starts from the filtered Rosetta Code dataset and terminates in the *H-AIRosettaMP*, obtained via code translation, comprising 90 (sub-)datasets. Each dataset is labeled by the author (Human or AI) and is represented by dst (the language of the dataset) and src language (the provenance of the AI-generated part of the dataset); (2) The *model training*, that shows the process leading to 90 monolingual models (one per dataset) and 1 multilingual model.

in Figure 1) have been obtained by selecting snippets from Rosetta Code in a way that created balanced datasets. Specifically, in each $\{src, dst\}$ dataset, we only kept Rosetta Code snippets pertaining to the same task. That is, each solution written in programming language *src* is kept in the dataset $\{src, dst\}$ if and only if a solution *for the same task* exists also for programming language *dst*, and vice-versa.

After this step, we obtained the balanced 90 (sub-)datasets shown in Figure 1 just before the *Translation* step that we describe next.

3) Translation: Li et al. [23] pioneered using code translation for building the AI-generated part of datasets for AI code stylometry. They discussed three alternative methodologies to do so:

1) *Code translation* involves providing the generative model with a code snippet in one programming language, asking the model to translate it into a different language;
2) *Functional translation* involves providing a natural language description of the desired task, asking to generate a solution;
3) *Functional customization* involves providing an existing snippet of code, asking to provide an explanation of what it does first, and then asking to generate a solution based on the description.

We considered all three options for our needs and concluded that (2) and (3) are not suitable option, because in our evaluation (using various code LLMs) they often end up producing "skeleton code", with holes that remain to be filled by the user. Keeping those incomplete snippets in the target dataset would give an unfair advantage to the AI detector, because they will be fairly easy to distinguish from complete code snippets from Rosetta Code (written by humans). We then settled for code translation (1) and applied it to the 90 balanced datasets obtained from the previous step.

Specifically, for each (sub-)dataset $\{src, dst\}$, we took all the snippets in it written in the *dst* programming language, and used the StarCoder2 [27] generative model to translate the snippet into the *src* programming language (*Translation* step in Figure 1, further detailed in Figure 2).

The choice of StarCoder2 as code completion model is due to its being an open model, both in its weights (available for download and reuse under the terms of the Open RAIL-M v1 license) and in its training dataset (obtained from the Software Heritage archive [10]). Openness is a strong requirement to achieve our goal of full reproducibility of the experimental pipeline, which would not be achievable using closed models such as Copilot or ChatGPT (and indeed has not been achieved in previous work in the literature). StarCoder2 achieves 46.3% accuracy in the HumanEval benchmark [8], a widely used benchmark for assessing the coding abilities of generative models, making it outperform coding models like CodeLlama and DeepSeekCoder [16], [39]. In summary: StarCoder2 is the best performing code LLM among those that are open enough to satisfy the reproducibility requirement.

To translate a snippet from language *src* to *dst*, we give to StarCoder2 the prompt whose synopsis is shown in Figure 3. When reading the generated output we select each next token by taking the one with the highest likelihood (greedy search), as it is commonplace in translation tasks.

Due to memory and computational limitations, we excluded snippet pairs $\{src, dst\}$ that would result in prompts longer than 1024 tokens, and we set 2048 as the maximum length in the generative phase. We have also excluded snippet pairs for which StarCoder2 returned malformed outputs (e.g., empty or lacking the closing `'''` delimiter).

After translation, each of the 90 $\{src, dst\}$ sub-datasets is now composed of snippets in a *single* programming language

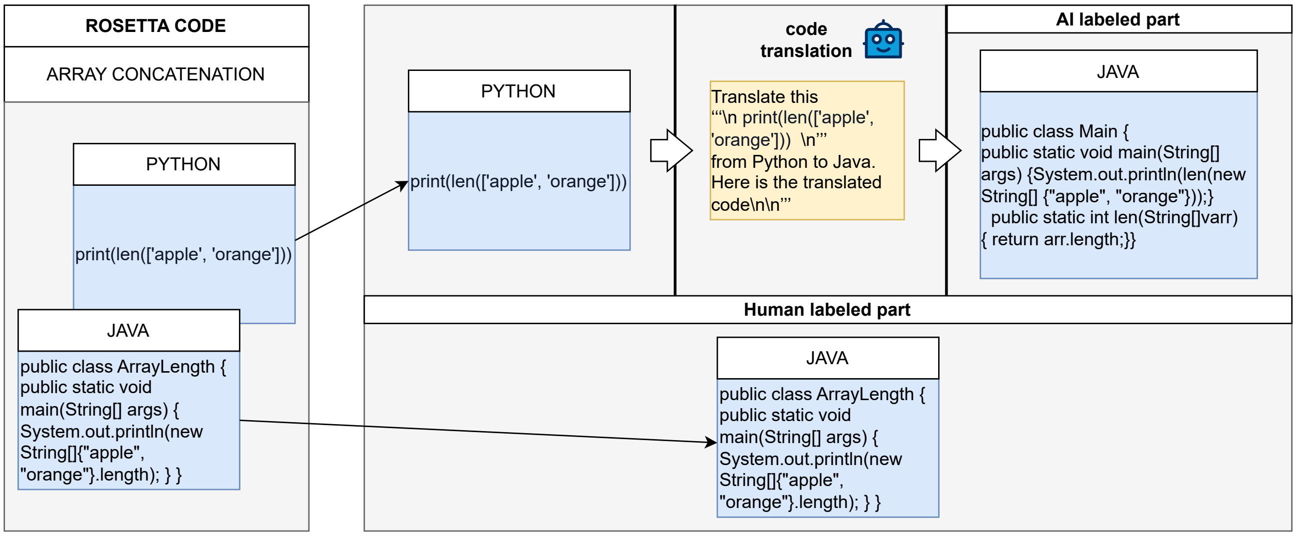

Fig. 2. Code translation step. The human-labeled part of the dataset (the `ArrayLength` Java class here) is a solution to a task from Rosetta Code (Array concatenation). The AI-labeled part is obtained via code translation (from Python to Java) using StarCoder 2. Input to the translation is a human-written solution for the same task, in a different programming language.

```
Translate this '''\n CODE_SNIPPET \n'''
from SOURCE_LANGUAGE to TARGET_LANGUAGE.
Here is the translated code\n\n'''
```

Fig. 3. Synopsis of the prompt given to StarCoder2 for translating a given code snippet (`CODE_SNIPPET` in the text) from a source programming language (`SOURCE_LANGUAGE`) to a target one (`TARGET_LANGUAGE`). A prefix of the desired answer (`Here is the...`) is provided because StarCoder2 has not been fine-tuned for chat-based interaction and is strictly a completion model.

(*dst*), labeled as either human-written (by a Rosetta Code contributor) or AI-written (by StarCoder2 via code translation).

All together, the 90 sub-datasets form a single reproducible dataset, called *H-AIRosettaMP*, which we release publicly as open data for others to experiment with. *H-AIRosettaMP* comprises 121 247 snippets, with 1127 unique tasks in 10 popular programming languages.

Note how the H-AIRosettaMP dataset satisfies multiple important requirements for the AI code stylometry task. As discussed by Caliskan-Islam [20] as an important feature: it contains multiple snippets, authored by multiple authors (grouped in two, in our case: humans vs AI), implementing different tasks. Specifically, it is not the case that the two authors are partitioned by task: for each task we have both a human-authored solution and an AI-authored one. This avoids the risk that the classifier will learn to distinguish *tasks*, rather than authors. Additionally, the dataset satisfies all the requirements associated with RQ1, namely: it is multilingual (with 10 languages), it is openly available and fully reproducible.

B. Model training

1) Classifier architecture: Until now, only classical machine learning techniques (discussed in detail in Section VII) have been applied to the AI code stylometry task. On the other hand, Niu et al. [32] showed how transformer-based architectures are state-of-the-art for several *code understanding* tasks. In the context of natural language (as opposed to code), recent works obtained successful results on AI recognition [25], [30] using analogous architectures.

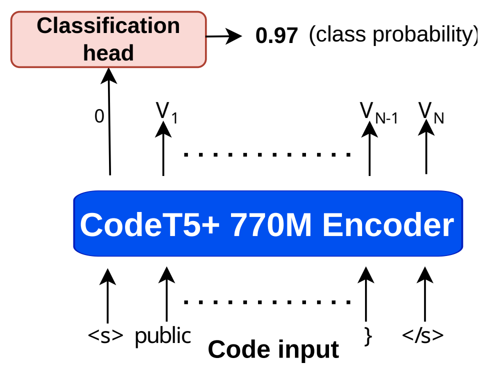

Fig. 4. Transformer-based architecture of the Human/AI stylometry classifier. Input source code is tokenized and provided as input to the CodeT5plus encoder, which produces as outputs multiple vectorial representations. The first token (`<s>` in the figure) is used as input for the classification head, which produces the final class probability.

In this work, we apply, to the best of our knowledge for the first time, a transformer-based machine learning architecture to the task of AI code stylometry. Specifically, we use CodeT5plus-770M [42], a pre-trained code transformer architecture, in an encoder setup, as shown in Figure 4. We provide the model with a tokenized text input and obtain several vectorial representations as the number of tokens. To distinguish human- from AI-written code, we add, following Niu et al. [32], a classification head to the first output representation of the model, corresponding to an always present classification token (see Figure 4). The classification head consists of a linear layer with a ReLU activation function, 20% dropout, followed by a final linear layer for binary classification.

2) Undersampling: As we describe below, we have trained 91 classifier models in total: one *monolingual model* for each of the 90 $\{src, dst\}$ sub-datasets + one *multilingual model* on a multilingual dataset sampled from the entire H-AIRosettaMP.

Before training the monolingual models, we undersampled each sub-dataset to the threshold of 470 code snippets for each Human/AI class, corresponding to the minimum amount of snippets across all sub-datasets.

Before training the multilingual model, we want to make sure that: (1) AI-written snippets, generated via code translation, come from a uniform distribution of source languages before the translation; (2) for both AI-written and human-written snippets, only a single solution for a given task is present (to avoid learning about the task, rather than learning about the author style). To ensure these properties, we processed the 10 languages one by one. For each language *dst*, we collect $470+470 = 940$ snippets (half AI-written, half human-written). When collecting AI-written snippets, we sample across the other 9 provenance languages *src*, with a uniform distribution. When collecting both AI-written and human-written snippets, we never select more than one solution for the same task; at most, the solution to the same task can hence appear twice in a given programming language, once as human-written and once as AI-written.

As an additional data cleaning step, we also removed all leading and trailing spaces from all code snippets (both human-written and AI-written) because AI-translated snippets exhibit recognizable heading/trailing spacing patterns, and we wanted to avoid unfairly advantaging classifiers that might learn from them (in real-world use cases those spaces would most likely not be preserved as is).

3) Training: We adjusted the model hyperparameters, starting from the setup proposed by Wang et al. [42], picking a subset of the dataset (all languages with Python provenance except for the Python language, translated from C++) and validated the model, obtaining a hyper-parameter setup for the rest of the experiments. We used AdamW [26] as an optimizer with a weight decay of 0.01. We trained each model for 15 epochs, multiplying the learning rate after 10 epochs by a 0.1 factor, with an initial learning rate of 2e-05.

After training the 90 monolingual models, we observed different results for the same *dst* language, coming from datasets with different provenance language *src* (see Section IV for details). We inspected this phenomenon by testing the best model for a language *dst* on datasets with different *src* provenance.

To obtain a model capable of handling several different languages as input, we trained the multilingual model using the entire H-AIRosettaMP dataset (after undersampling).

All models were trained with an 80%/20% training/test split.

We then compared our results with the best-performing results in the literature [23], [33]. Our classifiers were trained on the novel H-AIRosettaMP dataset, which is different from datasets used in previous works. Thus, for a fair comparison, we re-trained Li et al. and Oedingen et al. [23], [33] classifiers over our dataset.

We trained at first four baseline models following Li et al. [23] two methodologies (Random Forest and J48) over our Java and C++ sub-datasets with Kotlin provenance (because it obtained the best performances across all *src* provenances). The models were trained and evaluated using the same methodology of Li et al., with 10-fold cross-validation.

As our last comparison, we trained a baseline model following the best-performing methodology of Oedingen et al [33] over our Python dataset (also with Kotlin provenance).

4) Evaluation: We evaluated each of the trained monolingual classifiers on the respective dataset (in-distribution test), noting down the resulting accuracy. We evaluated in the same way the trained multilingual classifier on its own dataset (in-distribution test), which contains snippets in all languages and translated from all *src* languages (for the AI-labeled part).

We evaluated the 5 baseline models on different datasets (see Table IV for reference): RF Java and J48 Java baselines on the Java sub-dataset with translation from Kotlin (best average accuracy among all *src* provenance languages for Java); RF C++ and J48 C++ on the C++ sub-dataset (provenance: Kotlin); XGB-TF-IDF Python on the Python sub-dataset (provenance: Kotlin). Finally, we evaluated the monolingual models needed for comparison with baselines on out-distribution languages with Kotlin provenance, except for Kotlin itself, where Go provenance was used.

III. Dataset

The H-AIRosettaMP dataset is released publicly as part of our replication package (on Zenodo, see Data availability statement at the end of the paper) and also mirrored on Hugging Face.[4] The dataset comes in tabular form, with one row per snippet, for a total of 121 247 rows. Each column in the table provides some information about the snippet:

- `task_name`, `task_url`, `task_description`: information about the Rosetta Code [9] task that the snippet implements, respectively: the task name (e.g., Array concatenation), URL on the Rosetta Code website (e.g., http://rosettacode.org/wiki/Array_concatenation), and natural language description of the task.
- `language_name`: the programming language in which the code snippet is written, one of: C++, C, C#, Go, Java, JavaScript, Kotlin, Python, Ruby, Rust.
- `code`: the actual, full source code of the snippet as a string.
- `target`: a binary label denoting whether the snippet is human- or AI-written.
- `set`: the name of the specific sub-dataset, e.g., `"Java_from_C++"` for the Java snippet dataset whose

[4]https://huggingface.co/datasets/isThisYouLLM/H-AIRosettaMP

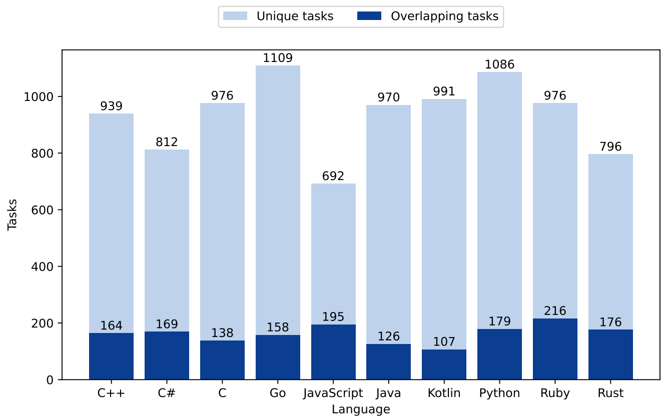

Fig. 5. Distribution of unique tasks for which solutions are present in the dataset per language. For each task, both a human-written and an AI-written snippet is always provided. Overlapping tasks denote the number of tasks for which multiple AI-written solutions are present, with a guarantee that they have been translated from all *other* programming languages in the dataset.

AI-written parts were obtained via translation from C++ (the human-written snippets, on the other hand, were natively written in Java).

As a simple descriptive statistics, Table I shows the average length of code snippets in the entire dataset by programming language, measured as the number of characters. We conducted a t-test for each language between the Human and AI-labeled groups of snippets with $\alpha = 0.05$ after having tested data normality and variance. When looking at the breakdown between AI- and human-written snippets, we see that six languages out of ten significantly differ in number of characters ($p < 0.05$ Table I). The noticeable differences in snippet lengths between AI- and human-written code suggest that length could be a predictive feature in code detection models.

We recall from Section II-A that the generation of AI-labeled snippets in the dataset is followed by task balancing, ensuring that each language-specific sub-dataset contains *pairs* of human-written/AI-written snippets solving the same task. Figure 5 shows the number of unique tasks (or, equivalently, the number of snippet pairs) for which solutions are present in each sub-dataset, aggregated by programming language. It also shows, for each language, the number of tasks for which there are solutions coming from all other provenance languages ("Overlapping tasks" in the figure).

Due to the uneven distribution of task solutions across languages in Rosetta Code, different languages in the dataset show a different number of tasks present. On the other hand, dataset users interested in avoiding the effect of translation provenance on AI-generated snippets can safely work in the subset of overlapping tasks.

In comparison to previous work in the literature [18], [23], [33], [43], this dataset provides the ability to develop and test AI stylometry classifiers along multiple snippet dimensions—(1) programming language of the snippet, (2) provenance language (for AI-written snippets, obtained via translation), (3) snippet length, (4) task implemented by the snippet—allowing to isolate how each of them influences the performances of AI code stylometry.

IV. Results

We show the results of our experiments in two separate tables. We first report, in Table II, accuracy results for all our models, both monolingual and multilingual. Then we compare, in Table IV, our best models to the best baselines from the literature.

We present all results in terms of overall accuracy, which is defined as the ratio of correctly classified snippets to the total number of snippets to be classified. In order to establish the statistical significance of our results, we also conducted t-tests (see Table III) and ANOVA tests with $\alpha = 0.05$ after testing data normality and variance.

In Table II, we depict the results of the experiments across all languages and provenance. Each monolingual classifier is tested only on snippets of the same language (in-distribution results), making the provenance language (i.e., the language AI-written snippets in the dataset were translated from) vary across all other languages. Therefore, we show the provenance language (*src*) of the snippets in rows, while in the columns, we display the target language *dst*, which is the language the models have been trained on. The table also shows the average accuracy by provenance language (*Prov. accuracy* column) and the average accuracy by tested language (*Language accuracy* row). The *Multilingual model* row provides results for the multilingual classifier, which has been trained on the multilingual dataset sampled from H-AIRosettaMP and tested separately on each language. Finally, the bottom-right cell of Table II contains the average accuracy of the multilingual model across the 10 programming languages considered.

The results in Table II show significant differences both in rows (same provenance language and different destinations) and columns (different provenances and same destination). Table III confirms that these differences among the models are significant, since we observed both for average provenance accuracies ($\mathrm{F-statistic} = 2.00$ with $p = 0.04$ in the table) and average language accuracy ($\mathrm{F-statistic} = 3.56$ with $p < 0.001$) relevant values.

In Table IV we compare our models with the baseline ones, namely J48 and Random forest algorithms from Li et al. [23] and XGB from Oedingen et al. [33]. All the baselines and our monolingual models (Java, C++, Python) are trained with our best dataset, namely the one that provides the best value for column *Prov. accuracy* in Table II, that is Kotlin. The test datasets are also all with Kotlin provenance, except for Kotlin itself which has Go provenance.

For monolingual models—when analyzing in-distribution tests—we highlight a substantial positive gap (+7.2% for the Java language, +8.9% for the C++ language, and +1.5% for the Python model) compared to the baselines. We notice the positive gap between the C++ model tested on the Java (out-distribution test) dataset (+2.4%), showing how, even when trained on a different language, this methodology performs better than the one adopted by Li et al. [23]. Out-distribution

TABLE I
AVERAGE LENGTH OF CODE SNIPPETS IN THE H-AIROSETTAMP DATASET, BY PROGRAMMING LANGUAGE, MEASURED IN CHARACTERS. T-TEST ($\alpha = 0.05$) IS COMPUTED BETWEEN THE AI-WRITTEN AND HUMAN-WRITTEN GROUPS

Language	**Snippet length** (character avg. ± std)			**t-test**		
	All	**AI-written**	**Human-written**	t-statistic	95% CI	p-value
C++	1183 ± 67	1061 ± 73	1306 ± 82	6.72	168 to 323	< 0.01
C	1094 ± 57	1077 ± 91	1112 ± 44	1.04	-37 to 107	0.31
C#	1248 ± 69	1240 ± 103	1257 ± 76	0.39	-74 to 107	0.69
Go	977 ± 63	883 ± 79	1072 ± 72	5.30	114 to 265	< 0.01
Java	1262 ± 88	1245 ± 102	1278 ± 101	-0.67	-134 to 69	0.51
JavaScript	933 ± 60	787 ± 82	1079 ± 87	7.32	207 to 377	< 0.01
Kotlin	920 ± 56	872 ± 91	969 ± 67	2.60	18 to 178	0.02
Python	744 ± 57	729 ± 86	761 ± 51	0.96	-38 to 102	0.35
Ruby	584 ± 42	650 ± 62	518 ± 36	-5.52	-183 to -82	< 0.01
Rust	992 ± 54	909 ± 81	1076 ± 71	4.63	90 to 243	< 0.01

TABLE II
ACCURACY (%) OF ALL TRAINED CLASSIFIERS WHEN SOLVING THE AI CODE STYLOMETRY TASK.

	Language tested										
Prov. language	**C++**	**C**	**C#**	**Go**	**Java**	**Javascript**	**Kotlin**	**Python**	**Ruby**	**Rust**	**Prov. accuracy** (avg. ± std)
Monolingual models											
C++	-	88.3	87.8	91.0	81.4	86.2	91.0	87.2	86.2	84.0	87.0 ± 2.9
C	89.9	-	90.9	91.5	92.0	90.9	94.1	89.9	88.8	88.8	90.8 ± 2.0
C#	75.5	87.8	-	93.6	88.3	91.5	92.5	87.8	88.3	88.8	88.2 ± 4.9
Go	94.7	94.7	87.2	-	90.9	95.2	94.7	84.0	90.4	85.6	90.8 ± 4.1
Java	91.5	92.0	86.7	92.0	-	89.4	92.5	87.8	88.8	83.0	89.3 ± 2.9
Javascript	90.9	95.2	88.3	91.5	87.7	-	93.1	85.6	90.4	84.0	89.6 ± 3.4
Kotlin	98.4	92.0	94.7	92.6	94.1	96.8	-	94.1	94.7	89.9	94.1 ± 2.4
Python	92.5	95.2	81.9	90.9	90.9	90.9	96.3	-	77.6	85.1	89.0 ± 5.9
Ruby	90.4	96.3	86.7	93.1	87.7	89.4	94.1	78.2	-	86.2	89.1 ± 5.0
Rust	88.3	97.8	90.9	87.3	94.7	91.5	87.8	88.8	89.4	-	90.7 ± 3.3
Language accuracy (avg. ± std)	90.2 ± 5.9	93.2 ± 3.3	88.3 ± 3.4	91.5 ± 1.7	89.7 ± 3.8	91.3 ± 3.0	92.9 ± 2.3	87.0 ± 4.1	88.3 ± 4.3	86.1 ± 2.3	-
Multilingual model											
Multilingual model accuracy	88.8	88.3	84.0	89.4	82.4	79.3	87.8	79.3	80.8	81.4	$\mathbf{84.1 \pm 3.8}$
Multilingual model F1	88.8	88.8	84.0	89.3	82.4	78.9	87.8	78.8	80.5	81.1	$\mathbf{84.0 \pm 4.0}$
Multilingual model AUC	94.1	94.9	91.8	93.9	89.9	90.4	94.6	90.8	88.6	93.9	$\mathbf{92.3 \pm 2.1}$

TABLE III
HYPOTHESIS TESTS FOR TABLE II: ANOVA TEST RESULTS ($\alpha = 0.05$) FOR LANGUAGE ACCURACY; ANOVA TEST FOR PROVENANCE ACCURACY; T-TEST ($\alpha = 0.05$) FOR MULTILINGUAL MODEL AND (MONOLINGUAL) LANGUAGE ACCURACY.

Values	**t/F-statistic**	**95% CI**	**p-value**
Lang. accuracy (ANOVA)	3.56	-	< 0.001
Prov. accuracy (ANOVA)	2.00	-	0.04
Multilingual comp. (t-test)	4.01	2.7 to 8.7	< 0.001

tests for the baselines are not shown as these architectures employ predefined features—lexical, syntactical (extracted from the abstract syntax tree of the source code), or deriving from source code layout—strictly linked to the language used during training, making the model specific to the designed language (a net advantage for the approach proposed in this paper).

We notice how the multilingual classifier performs worse than monolingual classifiers in both Table II and Table IV. In particular, Table III shows that the multilingual classifier has an accuracy that is worse than that one of the monolingual model in the average language case (-5.17% avg. $t\text{−}statistic = 4.01$ with $p < 0.001$). The multilingual classifier Table IV, however, does not present outliers in terms of accuracies, obtaining a model with consistent results, effective in handling multiple languages and the provenance phenomenon. In addition to the practical benefits of having a single model, this is another reason, as we will discuss in Section V, why the multilingual model is preferable for detecting AI-generated code in practice.

We also observe that our reimplementations of the baselines perform worse than the results reported in the original papers by both Li et al. [23] and Oedingen et al. [33]. Specifically, we obtained negative gaps of -18.5% with Java [23], -3.5% with the C++ baseline [23] and -5.2% for the Python baseline [33]. Since we reimplemented the same methodologies and applied them on our dataset (to perform a fair comparison on an identical dataset), we attribute these differences to the dataset itself, suggesting that H-AIRosettaMP is a harder benchmark than the ones used in previous work—which also makes intuitive sense, given the fast-paced advances in LLM-based code generators.

TABLE IV
COMPARISON BETWEEN THE ACCURACY OF BASELINES CLASSIFIERS [23], [33] AND CLASSIFIERS INTRODUCED IN THIS WORK. BEST RESULTS FOR EACH COLUMN ARE SHOWN IN **BOLD**.

	Tested language									
Model	**Java**	**C++**	**Python**	**Javascript**	**C#**	**C**	**Go**	**Ruby**	**Rust**	**Kotlin**
Baseline models										
RF Java [23]	79.3	-	-	-	-	-	-	-	-	-
J48 Java [23]	86.9	-	-	-	-	-	-	-	-	-
RF C++ [23]	-	85.5	-	-	-	-	-	-	-	-
J48 C++ [23]	-	89.5	-	-	-	-	-	-	-	-
XGB-TF-IDF Python [33]	-	-	92.6	-	-	-	-	-	-	-
Proposed models										
Java (monolingual)	**94.1**	83.9	87.7	90.6	**88.9**	80.4	86.7	86.6	85.0	31.3
C++ (monolingual)	89.3	**98.4**	88.0	**90.7**	86.7	**89.5**	86.1	87.0	**86.2**	24.1
Python (monolingual)	63.1	82.4	**94.1**	90.2	58.5	83.6	58.7	**90.7**	84.0	44.1
Multilingual	82.4	88.8	79.3	79.3	84.0	88.3	**89.4**	80.8	81.4	**87.8**

V. DISCUSSION

a) Findings: Based on the results presented in Section IV we can answer the stated research question affirmatively: it is possible to recognize AI-written programs with high average accuracy (84.1%), across 10 different programming language (multilingual code stylometry), with a single trained classifier based on a transformer-based architecture, novel for this task.

To achieve this, we devised and implemented a fully open and reproducible methodology and also replicated previous experiments in the literature [23], [33]. We observe significant performance differences not only across different datasets (which is to be expected), but also between accuracies previously reported in the literature and our replications of the same experiments with the same architectures. The following factors might be the cause of these discrepancies:

(1) The language in which the snippets to be recognized are written in plays an important role. For example, we see a pattern in our results comparing C snippets (accuracy $93.2\% \pm 3.3$) with Rust snippets ($86.1\% \pm 2.3$), for a difference of 7 points. It is entirely possible that AI-written code in some programming languages is intrinsically easier to recognize as such than code written in other languages. Establishing this conclusively is an interesting direction for future work.

(2) We also observe significant accuracy differences between datasets with different provenance languages (i.e., the languages AI-written code snippets where translated *from*). In order to test the impact of this factor, we analyzed models in the same language but with different provenances for the testing datasets (Figure 6). More precisely, since for the same language we have different models (depending on the provenance language), we first selected the models with the best provenance accuracy (last column of Table II), namely the models with Kotlin provenance. We depicted in Figure 6, with the dashed orange line, the accuracy of these models on the various languages (that is, we reported in the Kotlin row of Table II). Then, we compared these accuracies to those obtained for each language $L \neq \text{Kotlin}$ by averaging the accuracies of the L model on all the training datasets generated in L from the other provenances. This is reported in Figure 6 with the blue line.

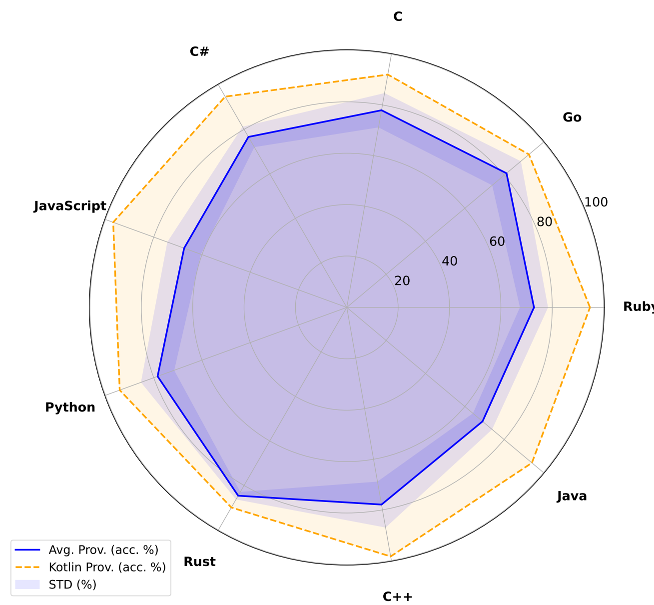

Fig. 6. Accuracy of our best-performing monolingual models. The dashed orange line corresponds to the Kotlin row in Table II (in-distribution). The solid blue line shows the accuracy of the same models on test sets with a different language provenance (potentially out-distribution, as is general translating solutions for the same task from different languages to the same language will result in different snippets). Model performances degrade significantly in the latter case.

As the picture shows, we obtain a significant loss in accuracy (-17.93% in avg. $\text{t-statistic} = 8.82$ with 95% CI 12.4 to 21.3 and $p < 0.001$). We believe this phenomenon is tightly coupled with the methodological choice of producing AI-written snippets via code translation and would not be observable under different conditions. We consider that we have properly mitigated this by training our main classifier to be multilingual across multiple translation provenances.

Still, the accuracy differences are interesting *per se* and pave the way to dedicated future work. We hypothesize that different ways of inputting—and prompting—the generative model, both to generate the training datasets and to use the obtained classifiers, can influence its performance, possibly leading to detection evasion.

(3) Reproducibility issues. As is way too common in empirical software engineering [15], studies tend to under-report the details needed to fully replicate their empirical findings, particularly so when AI-training is involved. When replicating baseline results from previous work, we tried our best, but some artifacts were not available and had to be reimplemented from scratch. This work contributes to raising the bar of replicability for AI code stylometry by relying on and producing only openly available artifacts.

Aside from the above, in Section III, we highlighted two other factors that can influence detection accuracy: dataset size and task balancing. We did not explore systematically how either of them influence the results; it is left as future work.

b) What about ChatGPT?: ChatGPT is one of the most popular generative AI tools on the market, for both generating natural language text and program code. We could not use ChatGPT in our experiments without undermining their replicability—which is one of our goals and differentiating aspects w.r.t. previous work. Still, it is interesting to verify how our reproducible classifiers, trained using only openly available data, fare against ChatGPT (whose model and training data are undisclosed).

We tested our multilingual classifier (trained on the H-AIRosettaMP dataset) on the Human-AI dataset by Oedingen et al. [33], whose AI part consists of Python code snippets generated by ChatGPT (at the time and version of their experiments) starting from natural language prompts. Without any fine-tuning on ChatGPT data, our multilingual classifier achieved 72.8% accuracy, with a -6.3% drop from the results obtained on our dataset. The accuracy gap is more significant when compared to the results reported by Oedingen et al. [33]: -25.2% with our model on their dataset. Still, our classifier results are way above the chance (50% for 2 classes), multilingual, and reproducible.

The accuracy gap can be due to several factors: the LLM used to generate the AI snippets, the different inputs and prompts used during generation, hyperparameter tuning, etc. Ultimately, the important question here is whether, in the upcoming arm race between AI code generation and AI code stylometry (to detect it), we can rely on closed models and datasets or not. We argue we should not and propose a new baseline for reproducible, multilingual, AI code stylometry with this work.

VI. Limitations

External validity: Our approach to the generation of AI-written snippets in the novel H-AIRosettaMP dataset is reproducible code translation. Our results could be impacted by that choice. We have explored the possibility only superficially, by verifying how our classifier performs on the dataset of Oedingen et al. [33], generated using ChatGPT on natural language prompts (hence: not code translation), with good results (cf. Section V). A more thorough analysis of how our results generalize to other prompting techniques (e.g., natural language prompts) and LLMs is left as future work. Note that we share this threat with all related work and that it is impossible to fully mitigate this threat encompassing closed LLMs (like ChatGPT) without sacrificing reproducibility.

Reliability: We address reliability threats in the usual way by releasing a comprehensive replication package covering all experiments discussed in the paper (see the Data availability statement at the end). In this respect, we fare better than all previous works by relying only on openly available datasets and components, including third-party LLMs.

VII. Related work

a) Code stylometry: Oman and Cook [34] were the first to introduce the notion of code stylometry. They hypothesized that each author is recognizable by a unique coding style called "fingerprint". In their pioneering work, they approached the task using cluster-based classification, introducing an unsupervised technique for inferring the code author.

Caliskan-Islam et al. [20] were the first to use both syntactical features (from ASTs) and lexical features (from concrete syntax trees) for code stylometry. They showed how a random forest classifier can take advantage of both kinds of information to achieve an accuracy of 53.91% for the Python language across 229 different authors. Other works followed the Caliskan approach, e.g., Dauber et al. [12].

More recently, the emergence of word embeddings [28] introduced a shift in the representation of author fingerprints from classical machine-learning techniques to deep-learning ones. Deep learning approaches [3], [6], [22] led to better author style representation capabilities, most notably by leveraging LSTM and code2vec [2] architectures. In terms of code stylometry accuracy, this resulted in a bump up to 95.90% with 70 different authors [6]. These architectures represent source code using both syntactical and lexical features.

Our work in this paper is a specific instance of the code stylometry task, where we aim to distinguish a specific "AI author" (a code LLM) from human authors. To that end we introduce the use of a transformer-based architecture [41], novel for the code stylometry task. Contrary to more traditional code stylometry work, we do not rely on syntactical features, but solely on lexical features (token stream).

b) AI detection for natural language: Köbis and Mossink [21] observed first how the generative capabilities of LLMs make it difficult to distinguish their (natural language, in this case) output from human-authored text, paving the way to research on the topic.

Early studies [5], [14], [17] approached the problem of AI-generated natural language using stochastic approaches. They generated AI-labeled samples with GPT-2 [36] and reached accuracies up to 93% [5].

Liao et al. [25] introduced the use of BERT [13] for recognizing AI-generated natural language. They demonstrated that this approach results in superior accuracy (96.7%) compared to traditional machine learning approaches: +7% w.r.t. XGBoost (decision tree) to a fine-tuned BERT architecture.

Mitchell et al. [29] used an approach based solely on probabilities sampled from a generative model, reaching 86% AUROC. Mitrović et al. [30] compared the performance of different approaches that do not need fine-tuning. They show that supervised techniques perform better, with a 14% accuracy increase from a perplexity-based approach (84%) to DistillBERT (98%).

With respect to these works, we focus on AI *code* stylometry, rather than natural language. We adopt an LLM-based approach (like [25], [30]), fine tuning the T5plus [42] LLM. For dataset generation, we use the open-weight StarCoder2 [27] code LLM.

c) AI code stylometry: Hoq et al. [18] looked at the problem of AI *code* stylometry, for educational plagiarism detection in the context of university computer science class. Their dataset consists of: (1) student-written Java code from a publicly-available dataset encompassing multiple problems with human solutions, and (2) solutions to the same problems generated by ChatGPT. Their approach relied on both syntactical and lexical features extracted from the code, fed to both traditional machine learning techniques (random forest) and deep learning ones, such as code2vec [2]. They reached accuracies up to 95% (with code2vec).

Both Bukhari et al. [7] and Idialu et al. [19] followed the same approach on different datasets and programming languages. The former focused on the C language using the Lost at C dataset [40] and Codex [8] for AI-generated snippets. The latter looked at Python code from the CodeChef learning platform, and generated the AI solutions with GPT-4. Both studies used random forest classification, reaching respectively 92% accuracy and 91% F1-score.

Yang et al. [43] replicated for AI code stylometry the probability-based methodology introduced by Mitchell et al. [29] for natural language. They focused on Java and Python, data coming from multiple LLMs from OpenAI, reaching AUCs up to 86.01% for Python and 77.42% for Java.

Oedingen et al. [33] analyzed the discrepancies between fine-tuned methodologies and zero-shot approaches (like Yang et al. [43]), showing how the latter struggle to achieve competitive performances. Using traditional machine learning techniques (XGB with TF-IDF features), they achieved impressive results (98% accuracy) on the detection of Python code generated by ChatGPT.

Rahman et al. [37] followed a similar approach, using a different (but still proprietary) LLM for code generation: Claude 3 haiku [4]. They reached 82% accuracy on Python.

Li et al. [23] introduced the use of translations (from either natural language specifications or existing code) to generate the AI-labeled part of training datasets for AI code stylometry. Using translation, they aim to reduce the chance of producing code already present in the training dataset of the code LLM that is to be recognized as an AI author. They considered C++ and Java languages (*separately*), using both ChatGPT and GPT-4 as generative models. They used random forest classification, with only lexical features, achieving 93% and 97.8% accuracy for C++ and Java, respectively.

We depart from previous AI code stylometry work in three ways: (1) we apply for the first time a transformer architecture to the AI code stylometry task; (2) we are able to recognize AI-written code across 10 different programming languages with a single model, achieving an average accuracy of 84.1%; (3) we rely only on openly-available data and code, enabling scientific reproducibility and future reuse of our work.

d) GPTSniffer: Independently and in parallel to our work, Nguyen et al. [31] introduced GPT Sniffer, that tackles the task of detecting AI-generated coming from a different perspective than code stylometry, but also using a transformer-based classifier. Their work is focused on the Java language, uses ChatGPT as a generative model, and considers the impact of different data source domains, such as programming books and data representative of real use-case scenarios that encompass a mixture of snippets from GitHub repositories and generated from ad-hoc queries. GPTSniffer performs really well in domain (100% F1 score training and testing with data provenance from a Java programming book), degrades a lot on out-of-domain (56% F1 with real use-case data for testing), and improves again on a mixed training set (94% F1) and data alteration techniques (96% F1). In comparison to GPTSniffer, we rely exclusively on open models for data generation and classification, ensuring experiment reproducibility, and supports a larger classification scope of 10 distinct programming languages with a single model.

VIII. Conclusions

AI code stylometry consists of automatically detecting whether an input piece of source code was authored by an AI (e.g., Copilot or ChatGPT) or a human. In this paper, we took a fresh look at the problem by revisiting assumptions made in previous work.

First, we solved the problem in a multilingual setting, supporting 10 different popular programming languages achieving high average accuracy (84.1%) with a single transformed-based classifier, a novel architecture for this task.

Second, our experiments are fully reproducible. As building blocks, we use only openly available data and components, including our code LLM: StarCoder2. We release openly all our artifacts: the novel H-AIRosettaMP dataset consisting of 121 247 code snippets in 10 languages, partly human-written (from Rosetta Code) and partly AI-written (via cross-language code translation); checkpoint of our trained model; and an open source CLI tool to use it in practice.

As future work we plan to analyze how snippet generation impacts detection accuracy, covering: prompt engineering, used code LLM, the provenance language for code translation, as well as starting from natural language prompts.

Data availability

A full replication package containing the dataset, a checkpoint of our multilingual model, a command-line tool for using it on selected code snippets, as well as the source code used to run all the experiments presented in this paper is available from Zenodo at https://doi.org/10.5281/zenodo.13908858.

REFERENCES

[1] Anisha Agarwal, Aaron Chan, Shubham Chandel, Jinu Jang, Shaun Miller, Roshanak Zilouchian Moghaddam, Yevhen Mohylevskyy, Neel Sundaresan, and Michele Tufano. Copilot evaluation harness: Evaluating llm-guided software programming. *CoRR*, abs/2402.14261, 2024.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, 2019.

[3] Bander Alsulami, Edwin Dauber, Richard E. Harang, Spiros Mancoridis, and Rachel Greenstadt. Source code authorship attribution using long short-term memory based networks. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*, volume 10492 of *Lecture Notes in Computer Science*, pages 65–82. Springer, 2017.

[4] Anthropic. The claude 3 model family: Opus, sonnet, haiku. https://www.anthropic.com/claude-3-model-card, 2024.

[5] Anton Bakhtin, Sam Gross, Myle Ott, Yuntian Deng, Marc'Aurelio Ranzato, and Arthur Szlam. Real or fake? learning to discriminate machine from human generated text. *CoRR*, abs/1906.03351, 2019.

[6] Egor Bogomolov, Vladimir Kovalenko, Yurii Rebryk, Alberto Bacchelli, and Timofey Bryksin. Authorship attribution of source code: a language-agnostic approach and applicability in software engineering. In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 932–944. ACM, 2021.

[7] Sufiyan Bukhari, Benjamin Tan, and Lorenzo De Carli. Distinguishing AI- and human-generated code: A case study. In Santiago Torres-Arias, Marcela S. Melara, Laurent Simon, Nikos Vasilakis, and Kathleen Moriarty, editors, *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses, SCORED 2023, Copenhagen, Denmark, 30 November 2023*, pages 17–25. ACM, 2023.

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[9] Rosetta Code. Rosetta code. https://rosettacode.org, Jul 2022. [Online; accessed 1-July-2022].

[10] Roberto Di Cosmo and Stefano Zacchiroli. Software heritage: Why and how to preserve software source code. In Shoichiro Hara, Shigeo Sugimoto, and Makoto Goto, editors, *Proceedings of the 14th International Conference on Digital Preservation, iPRES 2017, Kyoto, Japan, September 25-29, 2017*, 2017.

[11] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. GitHub Copilot AI pair programmer: Asset or liability? *J. Syst. Softw.*, 203:111734, 2023.

[12] Edwin Dauber, Aylin Caliskan, Richard E. Harang, Gregory Shearer, Michael J. Weisman, Frederica Free-Nelson, and Rachel Greenstadt. Git blame who?: Stylistic authorship attribution of small, incomplete source code fragments. *Proc. Priv. Enhancing Technol.*, 2019(3):389–408, 2019.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[14] Sebastian Gehrmann, Hendrik Strobelt, and Alexander M. Rush. GLTR: statistical detection and visualization of generated text. pages 111–116, 2019.

[15] Jesús M. González-Barahona and Gregorio Robles. Revisiting the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Inf. Softw. Technol.*, 164:107318, 2023.

[16] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196, 2024.

[17] Atsumu Harada, Danushka Bollegala, and Naiwala P. Chandrasiri. Discrimination of human-written and human and machine written sentences using text consistency. In *2021 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*, pages 41–47, 2021.

[18] Muntasir Hoq, Yang Shi, Juho Leinonen, Damilola Babalola, Collin F. Lynch, and Bita Akram. Detecting chatgpt-generated code in a CS1 course. In Steven Moore, John C. Stamper, Richard Jiarui Tong, Chen Cao, Zitao Liu, Xiangen Hu, Yu Lu, Joleen Liang, Hassan Khosravi, Paul Denny, Anjali Singh, and Christopher Brooks, editors, *Proceedings of the Workshop on Empowering Education with LLMs - the Next-Gen Interface and Content Generation 2023 co-located with 24th International Conference on Artificial Intelligence in Education (AIED 2023), Tokyo, Japan, July 7, 2023*, volume 3487 of *CEUR Workshop Proceedings*, pages 53–63. CEUR-WS.org, 2023.

[19] Oseremen Joy Idialu, Noble Saji Mathews, Rungroj Maipradit, Joanne M. Atlee, and Meiyappan Nagappan. Whodunit: Classifying code as human authored or GPT-4 generated- A case study on codechef problems. In Diomidis Spinellis, Alberto Bacchelli, and Eleni Constantinou, editors, *21st IEEE/ACM International Conference on Mining Software Repositories, MSR 2024, Lisbon, Portugal, April 15-16, 2024*, pages 394–406. ACM, 2024.

[20] Aylin Caliskan Islam, Richard E. Harang, Andrew Liu, Arvind Narayanan, Clare R. Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 255–270. USENIX Association, 2015.

[21] Nils C. Köbis and Luca Mossink. Artificial intelligence versus maya angelou: Experimental evidence that people cannot differentiate ai-generated from human-written poetry. *Comput. Hum. Behav.*, 114:106553, 2021.

[22] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. Building implicit vector representations of individual coding style. In *ICSE '20: 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020*, pages 117–124. ACM, 2020.

[23] Ke Li, Sheng Hong, Cai Fu, Yunhe Zhang, and Ming Liu. Discriminating human-authored from chatgpt-generated code via discernable feature analysis. In *34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023 - Workshops, Florence, Italy, October 9-12, 2023*, pages 120–127. IEEE, 2023.

[24] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason T. Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you! *Trans. Mach. Learn. Res.*, 2023, 2023.

[25] Wenxiong Liao, Zhengliang Liu, Haixing Dai, Shaochen Xu, Zihao Wu, Yiyang Zhang, Xiaoke Huang, Dajiang Zhu, Hongmin Cai, Tianming

Liu, and Xiang Li. Differentiate chatgpt-generated and human-written medical texts. *CoRR*, abs/2304.11567, 2023.

[26] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[27] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian J. McAuley, Han Hu, Torsten Scholak, Sébastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, and et al. Starcoder 2 and the stack v2: The next generation. *CoRR*, abs/2402.19173, 2024.

[28] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun, editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013.

[29] Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D. Manning, and Chelsea Finn. Detectgpt: Zero-shot machine-generated text detection using probability curvature. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 24950–24962. PMLR, 2023.

[30] Sandra Mitrovic, Davide Andreoletti, and Omran Ayoub. Chatgpt or human? detect and explain. explaining decisions of machine learning model for detecting short chatgpt-generated text. *CoRR*, abs/2301.13852, 2023.

[31] Phuong T. Nguyen, Juri Di Rocco, Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Massimiliano Di Penta. Gptsniffer: A codebert-based classifier to detect source code written by chatgpt. *J. Syst. Softw.*, 214:112059, 2024.

[32] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. An empirical comparison of pre-trained models of source code. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 2136–2148. IEEE, 2023.

[33] Marc Oedingen, Raphael C. Engelhardt, Robin Denz, Maximilian Hammer, and Wolfgang Konen. Chatgpt code detection: Techniques for uncovering the source of code. *CoRR*, abs/2405.15512, 2024.

[34] Paul W. Oman and Curtis R. Cook. Programming style authorship analysis. In Arthur M. Riehl, editor, *Computer Trends in the 1990s - Proceedings of the 1989 ACM 17th Annual Computer Science Conference, Louisville, Kentucky, USA, February 21-23, 1989*, pages 320–326. ACM, 1989.

[35] Marvin Wener Paul Jansen, Rob Goud. Tiobe - the software quality company. https://www.tiobe.com/tiobe-index/. Accessed: 2024-05-13.

[36] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[37] Musfiqur Rahman, SayedHassan Khatoonabadi, Ahmad Abdellatif, and Emad Shihab. Automatic detection of llm-generated code: A case study of claude 3 haiku. *arXiv preprint arXiv:2409.01382*, 2024.

[38] Jie Ren, Han Xu, Pengfei He, Yingqian Cui, Shenglai Zeng, Jiankun Zhang, Hongzhi Wen, Jiayuan Ding, Hui Liu, Yi Chang, and Jiliang Tang. Copyright protection in generative AI: A technical perspective. *CoRR*, abs/2402.02333, 2024.

[39] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950, 2023.

[40] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. Lost at C: A user study on the security implications of large language model code assistants. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 2205–2222. USENIX Association, 2023.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[42] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 1069–1088. Association for Computational Linguistics, 2023.

[43] Xianjun Yang, Kexun Zhang, Haifeng Chen, Linda R. Petzold, William Yang Wang, and Wei Cheng. Zero-shot detection of machine-generated codes. *CoRR*, abs/2310.05103, 2023.

[44] Albert Ziegler. GitHub Copilot research recitation — parrot or crow? a first look at rote learning in GitHub Copilot suggestions. https://github.blog/ai-and-ml/github-copilot/github-copilot-research-recitation/, 2021. Accessed: 2024-09-25.