Codetector: A Framework for Zero-Shot Detection of AI-Generated Code

Nadim Adham[1(✉)], Henning Duwe[1], and Holger H. Hoos[1,2]

[1] AI Methodology, RWTH Aachen University, Aachen, Germany
nadim.adham@rwth-aachen.de, {duwe,hh}@aim.rwth-aachen.de
[2] LIACS, Leiden University, Leiden, The Netherlands

Abstract. Generative artificial intelligence (AI) models are advancing rapidly, and their ability to generate code has significant implications for software development. Their use in code generation raises concerns about plagiarism, malware generation and dataset contamination. Previous work proposed methods to address these issues, using developments from the field of natural language detection. However, due to the infancy of the field, there is a deficit in established benchmarks and datasets, making a comprehensive comparison between detection methods difficult.

In this work, we investigated the efficacy of five existing zero-shot detection methods on AI-generated code. To do so, we used 17 large language models to generate and analyse 113 776 code samples from six common datasets and sources. By collecting new, previously unseen data with their respective time-stamps, we address the issue of data leakage, i.e., the exposure of LLMs to testing data during pre-training. Our proposed framework enables easy integration of new LLMs and datasets, facilitating the generation, detection and analysis of AI-generated code. Following this, we examined the impact of different hyperparameters used during code generation (such as the temperature) on detection performance. Our code is available under https://github.com/ Progyo/Codetector.

Keywords: Zero-shot Detection · LLMs · Code Generation · Framework

1 Introduction

Development in AI has made significant strides within the last ten years, and AI techniques are now being integrated into many systems we interact with, ranging from search engines to weather pattern prediction and more. Additionally, in recent years, an increasing number of AI systems have been developed to not only process data but to also generate content. Large language models (LLMs), in

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-032-09192-5_11.

Y. Zhang et al. (Eds.): LION 2025, LNCS 15745, pp. 157–173, 2026.
https://doi.org/10.1007/978-3-032-09192-5_11

particular, have shown a rapid rate of improvement in a multitude of tasks including several related to coding. Despite their popularity and success, LLMs come with risks: From malicious actors creating backdoors through dataset contamination [18], to careless users generating plagiarised or dangerous code [9,12,17]. Moreover, if left unchecked, such AI-generated content (AIGC) will bleed into the training sets of future models, possibly leading to phenomena such as model collapse, where models degrade due to training on their own synthetic outputs, causing them to converge into narrower modes and decrease in diversity [1]. These challenges underscore the urgent need for AI detection systems. However, AIGC detection, particularly for code, remains relatively immature, with the earliest work we found dating to 2023, allowing little time for the development of benchmarks or datasets [23,24].

Frequently, code detection performance is benchmarked using output from a single LLM [9,17,23], which can be problematic, especially for zero-shot methods that perform best when detecting text generated by the same base model [14]. This is also a common pitfall for fine-tuned detection methods that tend to specialise in the corpus or domain they were trained on, leading to weakened cross-domain performance [24]. Thus, focusing on the detection performance of code generated by a single model cannot accurately represent the performance of a given detection method. In addition to the models used to generate AIGC, the underlying data sources may also impact detection performance. All in all, we aimed to answer the following research questions in this work:

1. RQ1: Can zero-shot detection methods from other domains be effectively applied to code detection?
2. RQ2: What is the impact of data leakage on detection performance?
3. RQ3: How do generation parameters of LLMs, such as temperature, affect detection performance?

To answer these questions, we evaluated five existing zero-shot detection methods on AI-generated code from 17 LLMs. This code was generated based on samples scraped from different sources, such as LeetCode[1] and Stack Overflow.[2] After running the generated samples through our detection pipeline, individual detection values were assigned based on specific method-model combinations and stored as "detection samples". We also analysed the impact of these different data sources on detection performance and propose an end-to-end framework for generating and detecting AIGC. Overall, our contributions can be summarised as follows:

– We developed a framework called "Codetector" that enables the modular integration of new data sources and LLMs for generating and detecting AIGC.
– We evaluated the effectiveness of five existing zero-shot methods for detecting AI-generated code on 113 776 code samples, generated by 17 LLMs, and analysed 12 294 388 detection samples.

[1] https://leetcode.com/.
[2] https://stackoverflow.com/.

- We empirically demonstrate that the performance of existing zero-shot detection methods can be improved by using new, unseen samples for generation and detection.
- On our processed dataset, newer detection methods achieved AUROC scores of up to 97% with an average of around 83%, showing that some detectors can be directly applied to code detection.

The remainder of this paper is structured as follows: In Sect. 2, we introduce key concepts and existing work in AI-generated code detection. Following this, in Sect. 3, we briefly present the framework that we developed. In Sect. 4, we present the generated and detection datasets and the necessary steps we took to produce them. We then analyse the results in Sect. 5 and determine possible impact factors on detection scores. Finally, we summarise our findings in Sect. 6.

2 Background and Related Work

Detecting AIGC can be defined as a binary classification task where AIGC is typically assigned as the positive class, while human-generated content is assigned the negative class [24]. Therefore, the false positive rate (FPR) is the rate of incorrectly labelled human-generated samples; conversely, the true positive rate (TPR) is the rate of correctly labelled AIGC samples. Plotting the relation between the TPR and FPR for varying detection thresholds gives the receiver operating characteristic (ROC) curve of a given detector. The area under the ROC curve (AUROC) is a value ranging between 0 and 1 that indicates how well the detector can separate the two classes [17].

Contemporary work focuses on the task of detecting AI-generated natural language texts, leaving out a large pool of questions specific to the detectability of AI-generated code. A few studies have begun investigating this area, with Wang *et al.* [23] being the first to compare commercial and open-source detection methods for AI-generated code. They showed poor performance across the board, which raised the question of whether machine-generated code is more difficult to detect than machine-generated text in natural language, and entirely different techniques are needed. Several subsequent studies [9,20,25] have provided evidence that AIGC detection, in the context of code, may be more feasible than initially thought.

Much of the current research for code detection occurs in the context of education, usually with a focus on detecting submissions in beginner-level programming courses. Beginners often write inefficient code, a trait some detection methods inadvertently exploit [9]. It has been observed that ChatGPT [15,16] writes code utilising practices akin to those of a professional programmer, which stands out from the submissions of novices [9]. A lot of research focuses on zero-shot detection methods, because they do not require any fine-tuning, making them cheaper to iterate and develop. Ideally, a robust zero-shot method would also perform well across different model outputs, a challenge that current methods still face [14,24].

Some research investigated the performance of DetectGPT [14] in detecting modified Python code submissions with accuracies between 40–54%, reflecting the poor performance in previous literature [17,23]. DetectGPT4Code [25] investigates the detection capabilities of code generated by GPT-3.5 and GPT-4, reaching an AUROC of up to 86% when detecting Python samples generated by GPT-4. Meanwhile, CodeDetectGPT [20] replaces the costly LLM perturbation model of DetectGPT with a randomised algorithm that perturbs the sample by inserting spaces and newlines. This method achieved AUROC scores up to 99% in white-box settings at a temperature of 0.2, though performance dropped by 27% at 1.0 [20].[3] This raises an important question: Does detection performance consistently decrease with higher temperatures across all zero-shot detection methods? Overall, the current literature deviates from previous expectations, suggesting the plausibility of reliably detecting machine-generated code.

A common shared trait of the aforementioned studies is the near-exclusive focus on ChatGPT-generated code.[4] It is important to note that the tested versions of ChatGPT have been shown to have strong memorisation tendencies when completing code-oriented tasks [22]. This may have had an impact on the resulting AUROC scores reported in several studies by affecting their TPR and highlights the importance of choosing suitable datasets to generate and select code samples for detection from. While ChatGPT is among the most popular services used for tasks such as code completion, it is of utmost importance to evaluate the detection performance across a larger set of LLMs, to better gauge the state of the art in AIGC detection. In particular, as more services begin to build and adopt proprietary models, the need for strong cross-model detection methods will only increase. This fact, alongside the seemingly haphazard selection of code samples, is the motivation behind the extensive analysis and scrutiny of dataset sources and LLMs discussed in our work described in the following.

3 Framework

To generate the datasets used for code detection, we developed a framework called "Codetector". Our framework consists of two pipelines: A dataset generation pipeline and a detection pipeline. The dataset generation pipeline allows for the modular and easy integration of new data sources (Github, Stack Overflow, etc.) and LLMs used for code generation. The detection pipeline allows for the modular implementation of different detection methods and the LLMs they use. This design stands in contrast to many monolithic implementations of detection methods, where functionality is hard-coded and activated via command line arguments. The individual use cases and repositories utilised by our pipelines are unit-tested to ensure proper functionality. Another goal in designing both

[3] In our dataset, we utilise the temperatures 0.2 and 0.97. The latter was intended to be 1.0 to match the literature. This small difference was noticed too late in the generation process but arguably does not distort our findings.

[4] Or, more precisely, on code produced by models such as GPT-3.5 and GPT-4.

pipelines was for the implementation to be agnostic to the underlying frameworks used, such as PyTorch and TensorFlow, as well as higher-level frameworks, such as Huggingface Transformers.

While this model framework was initially developed for code generation and detection, it supports natural language samples as well. The impact of the underlying datasets, as well as other factors, are discussed in Sect. 5.

3.1 Dataset Generation Pipeline

The efficacy of various detection methods is strongly dependent on the quality and diversity of the samples contained in the datasets. Using our framework, new datasets and sources can easily be implemented by either extending existing dataset class formats like XML and Apache Parquet[5] or by implementing the required methods specified by the dataset abstract class. Additionally, datasets can be converted between different file formats out of the box, allowing data to be stored in both compressed and human-readable formats. Various filters can also be applied to datasets to prevent unwanted samples from being used for generation. Moreover, LLMs that inherit from the BaseModel class can be marked as generators by implementing the abstract GeneratorMixin. For further implementation details or examples regarding datasets and data sources, we refer to the IPython notebooks and the supplied source code.

3.2 Detection Pipeline

Once the dataset generation pipeline has produced a large corpus of human and machine-generated samples, these samples can be used for detection. Our framework currently supports single and two-model zero-shot detectors, allowing for flexibility in detection approaches. New detection methods can be implemented by extending either of the abstract classes, depending on the use case. To ensure compatibility within the detection pipeline, all LLMs used for detection must implement the DetectorMixin. LLMs that are used for both generating and detecting samples simply need to implement both mixins, reducing code redundancy while maintaining versatility.

4 Dataset Collection and Generation

A major contribution of our work is the collection and generation of a large corpus of human- and AI-generated code samples, using 17 models capable of code generation, of which the majority are open-source. It is important to note that we refrained from using more closed-source models for two reasons. Firstly, without access to the model weights, only a black-box analysis would have been possible, a known weakness of current zero-shot detection methods. Secondly, by using open-source models, we were able to reduce the research costs and

[5] https://parquet.apache.org/.

direct them to other purposes, such as generating output from a more expensive reasoning model (o1-mini). These models were chosen based on size, training date, coding capabilities and prior use in research.

Several factors were considered during the collection of this large corpus. Many earlier studies neglect the possible impact of data leakage in their detection datasets [20,23]. Evaluation benchmarks are often purged before pre-training, yet copies sometimes slip into the training data, affecting code quality scores [22]; thus, the impact of data leakage on detection performance must be considered as well. Evaluation benchmarks, such as APPS and CodeSearchNet, are often used as a source for generating AIGC [13,20,23,25]. When using samples seen in training, the chances of regurgitating fragments or entire snippets of code verbatim during generation increases [3,10]. This increase in false negatives would, in turn, reduce the TPR. It therefore makes sense to keep track of the inception dates of code samples contained in a dataset as well as the training cut-off date of the LLMs used to generate samples.

The length of the code samples also has a significant impact on the maximum achievable AUROC score [4]. Thus, the samples across different code detection datasets should also be similarly distributed in length for a fair comparison between them. How this was achieved in our work is explained in more detail in Subsect. 4.2. In addition to this, some related work sets a maximum output length for generated code samples [20]. This induces an upper bound for the AUROC score of the best detector and a pessimistic precedent for code detection capabilities [4,19].

4.1 Data Sources

To generate a diverse set of data that mimics real-life use cases, various data sources were compiled. These sources include common datasets used by several earlier studies on detection, such as APPS and CodeSearchNet, as well as code samples that were scraped from the web to further increase diversity. To investigate the impact of possible data leakage, two disjunct sets of data were generated with code samples from two different periods: One prior to the release of ChatGPT in November 2022 (the "Pre" datasets) and the other from August 2023 to April 2024 (the "Post" datasets). August 2023 was chosen as the lower bound for the "Post" datasets by determining the most recent training date of the 11 LLMs that were initially chosen for analysis. As a result, the samples in the "Post" datasets should be novel to the LLMs, ensuring that they have not been seen during training. The generated samples of the six remaining LLMs, including o1 mini, were added later to the final dataset but excluded from the dataset analysis. The goal was to inspect whether older code samples have a noticeable impact on the AUROC scores of detection methods. Another factor to consider here is the prevalence of machine-generated code in some of these sources, which may be marked as human-generated, increasing the number of false positives. The chance of this occurring rises substantially with samples collected after the release of ChatGPT.

Stack Overflow. Stack Overflow is one of the largest repositories of publicly available code in the world. Its strength lies in the large number of code snippets where functionality is often explained in natural language in the accompanying post, providing a great starting point for generating prompts. Despite this advantage, Stack Overflow is filled with non-functional code, hence why the code is in a post on the website. It is therefore reasonable to only consider code from accepted solutions or the most upvoted ones. This narrows down the number of applicable code snippets but increases the average quality of the code. After the initial parsing and filtering stage, the posts can then be labelled. For labelling, GPT-3.5-turbo was instructed to return a JSON-formatted string describing the functionality of the code and the programming language. The so-called "Stack Overflow Pre" dataset consists of 2 587 804 posts between August 1st, 2015, and April 7th, 2016. The "Stack Overflow Post" dataset consists of 791 122 posts between August 1st, 2023, and April 7th, 2024. Labelling all of these would be costly and excessive, as the goal for each dataset is 1700 samples.[6] Moreover, since the code is unknown, an issue arises: The distribution of programming languages in the respective datasets is hidden and is only revealed after labelling has been completed. To address this, code samples from the parsed XML files were sampled to approximate a log-normal distribution with a total of 30 000 samples each. This gives enough headroom for the target programming language distributions to be reached. Similar precautions were taken for the following three data sources.

LeetCode. The performance of an LLM on LeetCode is often used as a metric to gauge a model's coding capabilities and understanding of diverse problem sets, given that it consists of short code samples ranging from easy to hard difficulties. Unlike Stack Overflow, LeetCode consists solely of functional code, making it a favourable candidate as a source dataset. Although many web-scraped LeetCode datasets exist containing thousands of problem sets with solutions written in different programming languages, they cannot be used directly for generation for the following reasons. A key goal of this paper is to investigate the impact of possible data leakage on detection performance. This requires each problem to be timestamped to claim with high certainty that the problem is novel to the LLMs being used to generate new samples. Unfortunately, LeetCode problems are chronologically numbered but not timestamped. Using the Wayback Machine[7] to access previous versions of LeetCode's sitemap XML files and linear regression, an estimate for the problem numbers matching the November 2022 and August 2023 cut-off could be made. Problem set 2517 is estimated to be the upper bound for the pre-ChatGPT dataset and problem set 2883 is the lower bound for the August 2023 dataset.

[6] We chose this number based on computational estimates and budget constraints.

[7] https://web.archive.org/. The Wayback Machine archives versions of accessible websites on the web.

APPS and CodeSearchNet. Automated Programming Progress Standard (APPS) [8] and CodeSearchNet [11] are two popular datasets used to generate AIGC samples for detection [13,20,23,25]. APPS contains 10 000 Python problems at varying levels of complexity. These problems are used to test a model's natural language to code translation abilities [8]. In addition to the reference code, APPS supplies over 130 000 test cases for testing the generated code. These were not used in this paper but could be a way to filter out low-quality code. CodeSearchNet is comprised of 2 326 976 documented functions. These functions were sourced from non-fork GitHub repositories that were popular and had permissive licenses [11]. For comparison across multiple datasets, only the Python subset containing 503 502 samples was used.

4.2 Dataset Processing

It has been observed that some programming languages are more difficult to detect than others. For example, various detectors have been observed to report strictly higher AUROC scores on Python code samples from CodeContest than on Java samples [25]. Other work suggests that the detectability of samples written in different programming languages varies across detection methods [23]. This discrepancy further highlights the importance of closely inspecting the sources and distributions of the code samples. If one dataset contained more of a difficult-to-detect language, this could skew the detection score and lead to misleading results. Similarly, code length has an impact on the maximum achievable AUROC score of a detector [4,19], and by having two different code length distributions, the performance will be intrinsically different between the two datasets leading to inconclusive results. To investigate the impact of the underlying data sources on detection capabilities, the disjunct datasets must have similar distributions in both programming language and code length to ensure a fair comparison.

To address this issue, we developed a utility class called the "dataset helper" to generate a code length distribution based on the intersection of two separate data sources. This was calculated on a programming language basis to mitigate the aforementioned issue. The distribution was then sampled to fit a scaled log-normal distribution. A log-normal distribution was chosen, because it closely mirrors length distributions found in reality, and many unfiltered distributions exhibit similar characteristics. Additionally, this choice increases the presence of longer samples in the tail. A uniform distribution is infeasible, due to the large number of longer samples that would need to be generated, which would create an overly optimistic portrayal of code detection given its high mean code length. After limiting the distributions to a log-normal distribution, the Python code length histograms of the Stack Overflow, APPS, and CodeSearchNet datasets are identical.

Using the final distributions, a list of code sample hashes can be generated. These hashes, derived from the SHA256 hash of the UTF-8 string representation of each sample, serve as compact identifiers. They can be stored in a file and used as a filter during dataset loading to exclude other samples, ensuring adherence

to the intended distribution. Notably, problem difficulty was not factored into sample selection in the generation pipeline, though it is reasonable to assume that problems requiring longer solutions tend to be more complex.

4.3 Generated Dataset

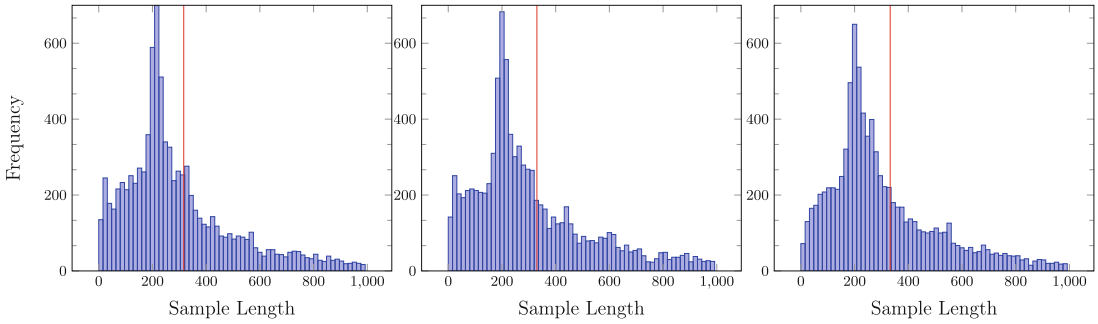


Fig. 1. Generated Python code length distribution of APPS (Left), CodeSearchNet (Middle), and Stack Overflow Post (Right) with a bin size of 16. Length in number of tokens using the cl100k base TikToken tokenizer.

In total, we generated 113 776 code samples across 6 datasets and 17 LLMs. It is important to note that samples with less than eight tokens were discarded, since they were either empty strings or filled with white space. Discarding samples from the dataset introduces the notion of imbalance between human and AIGC samples in the dataset. Comparing the human samples to samples generated by a single LLM, we can calculate an average human-to-generated ratio (see Table 1). To prevent class imbalance, it is important that the ratio stays close to 1 between the samples. Where possible, the parameters for temperature and top-p were set to 0.97 and 0.95, respectively. Temperature is a floating point value applied to the softmax function in the final layer of an LLM that flattens or sharpens the probability distribution of the next token.[8] Top-p sampling (or nucleus sampling) is a randomised sampling strategy that samples from the smallest subset of tokens with the largest probability whose sum exceeds the top-p value [5].

The temperature value and sampling strategy used by the LLM can have a large impact on the detectability of a code sample [20]. A lower temperature value sharpens the conditional probability distribution of the next token leading to a more deterministic generated output. This makes detection easier in a white box context because it increases the probability of a base model "recognizing" its own samples. Increasing the temperature has the opposite effect. Similarly, decreasing the top-p value increases determinism and vice versa. To verify the impact of these parameters, 26 636 additional CodeSearchNet samples were generated with varying temperature and top-p combinations. To compare results, some values

[8] Specifically LLMs utilising a decoder-only transformer architecture.

Table 1. List of the data sources, total human- and machine-generated samples, and average human-to-generated ratio (per LLM). E.g., For APPS, we have 561 human-generated samples. Each LLM has slightly fewer than 561, resulting in 9149 machine-generated samples and an average ratio of 1:0.96. *Machine-generated total contains additional samples generated at varying temperature and top-p values.

Data source	HumanTotal	Machine-Generated Total	Avg. Ratio
Stack Overflow (Pre + Post)	3400	25 544 + 27 243	1:0.94
APPS	561	9149	1:0.96
CodeSearchNet*	561	35 925	1:0.98
LeetCode (Pre + Post)	1000	8025 + 7890	1:0.94

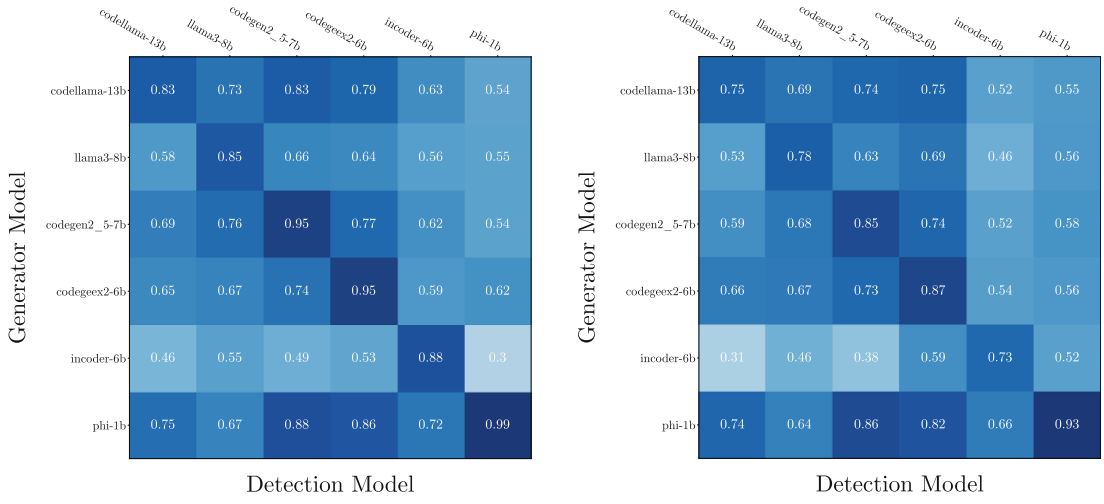
were taken from previous work [20]. A total of four combinations of $T = 0.97$, $T = 0.2$, $p = 0.95$, and $p = 0.5$ were tested.

After generation, it is to be expected that some variance between the dataset distributions is introduced. Since models behave differently from one another and the code may be semantically similar but structurally different, the code lengths will vary as well. Although the distributions are not identical, the mean did not deviate significantly (see Fig. 1). The generated output was then analysed using various zero-shot detection methods within the framework developed in this paper. Both the generated data and the real values assigned by various zero-shot detection methods are published as separate datasets alongside this paper.

4.4 Computational Requirements

Throughout the creation of the datasets and results presented in this paper, we used a high performance cluster to run the LLMs during generation and detection. We ran the generation and detection phases sequentially across 10 parallel instances, each with their own H100 GPU, resulting in approximately 1042 compute hours being spent overall. Around 112 h were spent generating the samples, and another 930 running the detection methods across various model combinations. We determined the batch size, among other optimisation parameters, by running preliminary benchmarks to maximize token throughput. It should be noted that our framework automatically groups samples by size to reduce unnecessary padding added during batching that wastes memory. On average it took around 3.5 s to generate a sample and around 270 ms per detection sample. Although we had access to 96 GB of VRAM per instance, we used quantised models to allow for larger batch sizes to increase inference speed. The continuous progress of LLM quantisation will allow these models to be run on lower-end hardware, making zero-shot detection more viable in the future. The impact of quantisation on detection performance, however, is unknown and remains an open research question.

5 Analysis



(a) Fast-DetectGPT (b) Binoculars

Fig. 2. Cross-model performance on two datasets. (a) Fast-DetectGPT on Stack Overflow Post and (b) Binoculars detector on APPS.

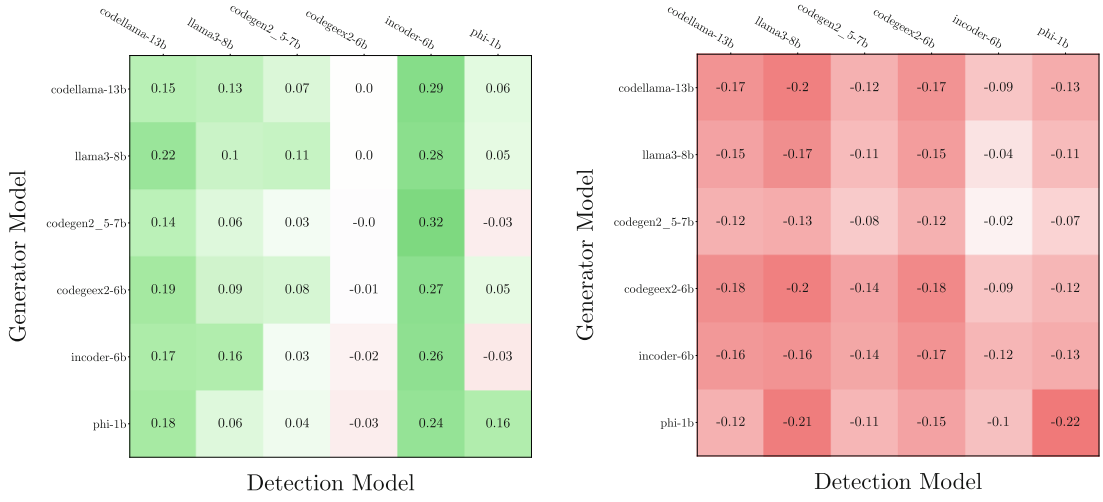
It is crucial to investigate the impact of various factors pertaining to the properties of the samples themselves. Key factors include sample length, programming language, and generation parameters, like temperature and top-p value. This yielded 12 294 388 detection samples from five detection methods [2,6,7,21] and 16 LLMs.[9] To examine whether data leakage affects detection performance, we analysed a subset of detection models trained before our cut-off date, focusing on samples from 11 of the 17 LLMs. While initially working on constructing the source datasets and theorising the impact of data leakage, we hypothesised that introducing previously seen samples could have two opposing effects. Hypothesis A: Detection performance increases because the underlying LLM "recognises" its own generated samples better. Hypothesis B: Detection performance decreases because the generated samples closely match the original human samples, making differentiation more difficult. Our goal was to highlight the importance of dataset selection and generation, urging standardisation in the field through benchmarks and frameworks to allow for a more reliable comparison between detection methods and to answer to following research questions:

RQ1: Can Zero-Shot Detection Methods from Other Domains be Effectively Applied to Code Detection?

Heat maps are a useful way to visualise the cross-model performance of a detection method. It should be noted that, to maintain readability of the data con-

[9] o1-mini cannot be used as a detection model because OpenAI do not expose the logits produced by the LLM that are required for the investigated methods.

tained in the figures, we only present six of the LLMs in the following figures. We chose to represent only the largest model of a given family and remove instruct versions from the figures. After inspecting the heat maps of several detection methods and datasets, it becomes clear that current methods still generally perform better in a white box context compared to a black box one. This is evident from the darker diagonal in the heat maps (see Fig. 2). Alongside the code and datasets, we provide a large collection of complete figures, including various heat maps of detection method and dataset combinations, which can also be generated using the supplied IPython notebook files in the code repository. Overall, there is strong evidence supporting the applicability of existing detection methods for AI-generated code.



(a) Rank (b) Entropy

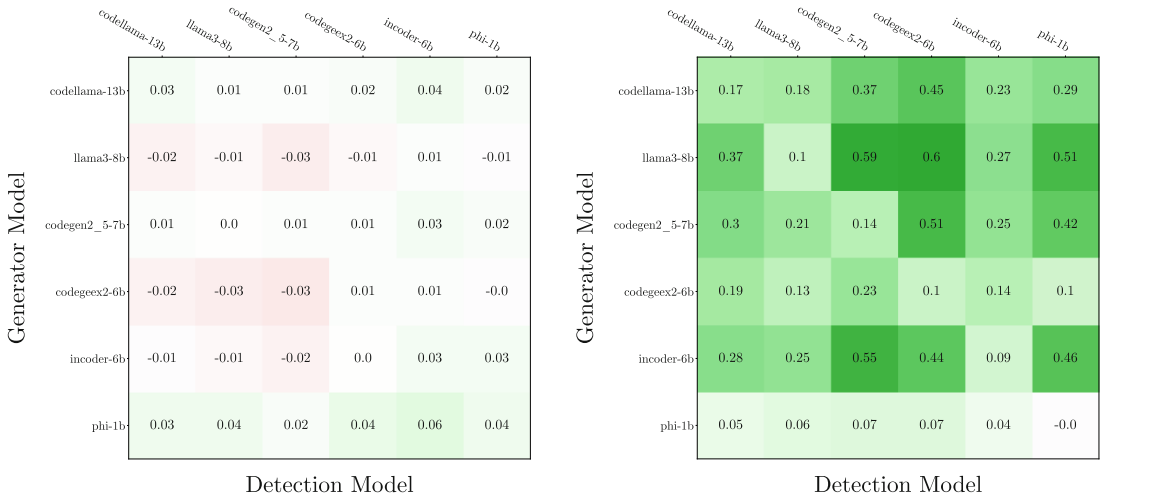
Fig. 3. (a) Python-only cross-model performance comparison between CodeSearchNet and Stack Overflow Post using rank (b) Cross-model performance comparison between LeetCode Pre and LeetCode Post using entropy. Green indicates an improvement in AUROC. Red indicates a decrease in AUROC. (Color figure online)

One can also see a large discrepancy in AUROC scores between detection methods. Directly comparing AUROC scores without considering the underlying ROC curve can be misleading, as AUROC alone does not reflect key detector characteristics, such as the FPR-TPR relationship at fixed thresholds. Filtering AIGC from future datasets requires a high TPR, as being overly cautious is preferred, while the FPR is less relevant. Conversely, detecting AIGC in student submissions requires a low FPR at the cost of letting some AIGC evade detection. Thus, we also investigate the ROC curves. Only curves that dominate others can be called superior detectors. Inspecting several ROC curves with varying detection methods, generator and base model combinations reveals a clear trend: Generally, the larger the difference between the AUROC scores, the greater the likelihood that the ROC curve dominates the other, meaning that the TPR lies above the other across all FPRs. This is notable because it allows us to define

a heuristic that we call a "difference heat map" to better visualise differences in AUROC scores. The difference heat map visualises performance improvements of dataset B over A (A vs B) by calculating B - A per tile, highlighting positive differences in green and negative in red. In particular, the larger the absolute difference the more accurate the heuristic.

RQ2: What Is the Impact of Data Leakage on Detection Performance?

Figure 3 highlights the difference in AUROC between datasets before and after the cut-off date using the same detection methods. Although most detection methods showed an improvement, supporting hypothesis B, the entropy detector often decreased with novel samples (see Fig. 3).



(a) Log-likelihood (b) $T = 0.97$, $p = 0.95$ vs $T = 0.2$, $p = 0.5$

Fig. 4. (a) Cross-model performance comparison between Stack Overflow Pre and Stack Overflow Post using log-likelihood (b) Python-only cross-model performance comparison between CodeSearchNet at varying temperature and top-p using Fast-DetectGPT. Green indicates an improvement in AUROC. Red indicates a decrease in AUROC. (Color figure online)

Unlike when comparing Stack Overflow Post to APPS and CodeSearchNet, there was no significant change in detection performance between Stack Overflow Pre and Post (see Fig. 4a). While contradictory to both our hypotheses, this is not an inexplicable result. Firstly, less than 1% of samples were used in both of the 250-day periods of Stack Overflow Pre and Post. Additionally, the labelling process helped mitigate dataset contamination by encouraging novel outputs that better reflect the model's true coding abilities. This likely impacted how recognisable these samples were, even if seen during training. Since these steps were applied to both datasets, the generated samples should be of similar quality leading to the observed results. These findings may change if the natural language-to-code generation performance continues to increase or methods like retrieval

augmented generation (RAG) are utilised during generation. Future work may look further into the impact of rephrasing problem descriptions from existing datasets on detection performance. The minimal performance difference between Stack Overflow Pre and Post contrasts with the significant variations observed between APPS, CodeSearchNet, and Stack Overflow Post, as well as LeetCode Pre and Post. This highlights a key finding: for current LLMs, collecting newly labelled data outside of benchmarks and common training datasets is just as crucial as the cut-off date.

RQ3: How Do Generation Parameters of LLMs, Such as Temperature, Affect Detection Performance?

Difference heatmaps can also be used to examine how detection methods perform under varying generation parameters. Since AUROC values across different top-p and temperature configurations are derived from the same human reference samples, comparing them is meaningful. The difference heat maps indicate that lower top-p and temperature values resemble the results of newly labelled data (see Fig. 4b). In most cases, the AUROC score of detection methods increased significantly across the board except for entropy detection. These results further underscore the need for standardised detection datasets and frameworks, as variations in generation parameters can further complicate the comparability of presented results.

Additionally, difference heat maps can also be used to investigate the performance difference of detection methods across different programming languages. While the programming language distributions also follow a log-normal distribution, it is advised to cautiously interpret these results, as the distributions differ far more across programming languages than across datasets. However, these significant differences in detection performance still raise the question of the cause underlying this discrepancy. Hoq *et al.* [9] suggest that syntactic differences are a sufficiently strong indicator for differentiating AI-generated code from student submissions. This is partially substantiated in our dataset by our token frequency analysis, where it can be seen that some tokens are preferred by either humans or LLMs. While this may explain one possible factor for the variation in detection performance between different programming languages, further work investigating aspects such as the influence of training data or programming ability of an LLM in a specific language is required.

6 Conclusions

In this paper, we investigated the efficacy of existing zero-shot methods for detection of AI-generated code by assembling a large corpus of 113 776 samples generated by 17 LLMs. Using the generated code samples, we then produced 12 294 388 detection samples, by applying five detection methods and 16 of the LLMs.

We achieved this by developing a framework called "Codetector" that facilitates the modular integration of new data sources and LLMs for generating and detecting AI-generated content. The goal of our unified framework is to benefit the task of AIGC detection by allowing for an easier and more reliable comparison between detection methods. Using our framework, we gathered a mixture of newly labelled and existing human-generated code samples from various online sources, ensuring similar programming languages and length distributions. These samples were then used to generate new ones with the integrated LLMs. This careful scrutiny of collected and generated samples allowed us to not only show the feasibility of detecting AI-generated code but also to closely compare the detection results between subsets of our large set of samples.

We found that both old and new zero-shot detection methods often perform better when using new, unseen samples during generation and detection, with some detector-generator combinations showing an absolute AUROC increase of over 40%. This was achieved by using samples created after the model cut-off date or by relabelling existing samples. By doing so, we achieved AUROC scores of up to 97% with an average of around 83% using newer detection methods, despite using high temperature and top-p values of 0.97 and 0.95, respectively, during generation. In addition to the impact of diverse datasets and possible data leakage, we showed the significant impact of generation parameters on detection performance.

We hope that our findings on how various factors bearing on dataset creation impact detectability will further motivate the development of unified benchmarks for AIGC detection. Additionally, we aim for our unified framework to facilitate a more reliable comparison between detection methods, ultimately accelerating the development of novel detection techniques.

Acknowledgments. All simulations were performed with computing resources granted by RWTH Aachen University under project thes1707. This work was supported by the Alexander-von-Humboldt Foundation. Finally, we would like to thank Marie Anastacio for helpful feedback on the paper.

References

1. Alemohammad, S., et al.: Self-consuming generative models Go MAD. In: The Twelfth International Conference on Learning Representations (ICLR), pp. 1–13 (2024)
2. Bao, G., et al.: Fast-DetectGPT: efficient zero-shot detection of machine-generated text via conditional probability curvature. In: The Twelfth International Conference on Learning Representations (ICLR), pp. 1–13 (2024)
3. Carlini, N., et al.: Extracting training data from large language models. In: 30th USENIX Security Symposium, pp. 2633–2650 (2021)
4. Chakraborty, S., et al.: Position: on the possibilities of AI-generated text detection. In: International Conference on Machine Learning (ICML), pp. 1–15 (2024)
5. Finlayson, M., et al.: Closing the curious case of neural text degeneration. In: The Twelfth International Conference on Learning Representations (ICLR), pp. 1–12 (2024)

6. Gehrmann, S., et al.: GLTR: statistical detection and visualization of generated text. In: 57th Conference of the Association for Computational Linguistics (ACL), pp. 111–116 (2019)
7. Hans, A., et al.: Spotting LLMs with binoculars: zero-shot detection of machine-generated text. In: International Conference on Machine Learning (ICML), pp. 1–20 (2024)
8. Hendrycks, D., et al.: Measuring coding challenge competence with APPS. In: Neural Information Processing Systems Track on Datasets and Benchmarks, pp. 1–12 (2021)
9. Hoq, M., et al.: Detecting ChatGPT-generated code submissions in a CS1 course using machine learning models. In: 55th ACM Technical Symposium on Computer Science Education (SIGCSE), pp. 526–532 (2024)
10. Hu, X., et al.: RADAR: robust AI-text detection via adversarial learning. In: Advances in Neural Information Processing Systems (NeurIPS), pp. 1–12 (2023)
11. Husain, H., et al.: CodeSearchNet challenge: evaluating the state of semantic code search. arXiv preprint arXiv:1909.09436, pp. 1–6 (2019)
12. Khoury, R., et al.: How secure is code generated by ChatGPT? In: IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp. 2445–2451 (2023)
13. Li, B., et al.: Resilient watermarking for LLM-generated codes. arXiv preprint arXiv:2402.07518, pp. 1–18 (2024)
14. Mitchell, E., et al.: DetectGPT: zero-shot machine-generated text detection using probability curvature. In: International Conference on Machine Learning (ICML), pp. 24950–24962 (2023)
15. OpenAI, et al.: GPT-4 technical report. arXiv preprint arXiv:2303.08774, pp. 1–78 (2023)
16. Ouyang, L., et al.: Training language models to follow instructions with human feedback. In: Advances in Neural Information Processing Systems (NeurIPS), pp. 1–15 (2022)
17. Pan, W.H., et al.: Assessing AI detectors in identifying AI-generated code: implications for education. In: 46th International Conference on Software Engineering, pp. 1–11 (2024)
18. Qiang, Y., et al.: Learning to poison large language models during instruction tuning. arXiv preprint arXiv:2402.13459, pp. 1–14 (2024)
19. Sadasivan, V.S., et al.: Can AI-generated text be reliably detected? arXiv preprint arXiv:2303.11156, pp. 1–16 (2023)
20. Shi, Y., et al.: Between lines of code: unraveling the distinct patterns of machine and human programmers. arXiv preprint arXiv:2401.06461, pp. 1–12 (2024)
21. Solaiman, I., et al.: Release strategies and the social impacts of language models. arXiv preprint arXiv:1908.09203, pp. 1–71 (2019)
22. Tian, H., et al.: Is ChatGPT the ultimate programming assistant - how far is it? arXiv preprint arXiv:2304.11938, pp. 1–22 (2023)

23. Wang, J., et al.: An empirical study to evaluate AIGC detectors on code content. In: 39th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 844–856 (2024)
24. Wu, J., et al.: A survey on LLM-generated text detection: necessity, methods, and future directions. Comput. Linguist., 1–66 (2025)
25. Yang, X., et al.: Zero-shot detection of machine-generated codes. arXiv preprint arXiv:2310.05103, pp. 1–11 (2023)