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

Nadim Adham[1 [0009−0004−6161−1750]], Henning Duwe[1 [0009−0008−4361−1203]], and Holger H. Hoos[1,2 [0000−0003−0629−0099]]

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

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 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 [25], to careless users generating plagiarised or dangerous

code [13,16,24]. 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 [2]. 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 [31,32].

Frequently, code detection performance is benchmarked using output from a single LLM [13,24,31], which can be problematic, especially for zero-shot methods that perform best when detecting text generated by the same base model [19]. 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 [32]. 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[3] and Stack Overflow.[4] 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.
- We empirically demonstrate that the performance of existing zero-shot detection methods can be improved by using new, unseen samples for generation and detection.

[3] https://leetcode.com/.
[4] https://stackoverflow.com/.

– 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 Section 2, we introduce key concepts and existing work in AI-generated code detection. Following this, in Section 3, we briefly present the framework that we developed. In Section 4, we present the generated and detection datasets and the necessary steps we took to produce them. We then analyse the results in Section 5 and determine possible impact factors on detection scores. Finally, we summarise our findings in Section 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 [32]. 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 [24].

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.* [31] 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 [13,28,33] 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 [13]. It has been observed that ChatGPT [22,23] writes code utilising practices akin to those of a professional programmer, which stands out from the submissions of novices [13]. 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 [19,32].

Some research investigated the performance of DetectGPT [19] in detecting modified Python code submissions with accuracies between 40-54%, reflecting

the poor performance in previous literature [24,31]. DetectGPT4Code [33] 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 [28] 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 [28].[5] 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. For a more extensive overview of related work, see Table 2 in Appendix B.

A common shared trait of the aforementioned studies is the near-exclusive focus on ChatGPT-generated code.[6] It is important to note that the tested versions of ChatGPT have been shown to have strong memorisation tendencies when completing code-oriented tasks [30]. 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 pipelines was for the implementation to be agnostic to the underlying frameworks

[5] 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.

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

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 Section 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[7] or by implementing the required methods specified by the dataset abstract class. An example of how this can be achieved is shown in Appendix C.3. 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

[7] 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; for a detailed list of the models used, see Table 3.

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 [28,31]. Evaluation benchmarks are often purged before pre-training, yet copies sometimes slip into the training data, affecting code quality scores [30]; 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 (see Table 2). When using samples seen in training, the chances of regurgitating fragments or entire snippets of code verbatim during generation increases [4,14]. 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 [5]. 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 Subsection 4.2. In addition to this, some related work sets a maximum output length for generated code samples [28]. This induces an upper bound for the AUROC score of the best detector and a pessimistic precedent for code detection capabilities [5,27].

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 (see Table 2), 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 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 (see Appendix C.6). 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.[8] 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[9] 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. For more details see Appendix C.4.

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

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

APPS and CodeSearchNet. Automated Programming Progress Standard (APPS) [12] and CodeSearchNet [15] are two popular datasets used to generate AIGC samples for detection (see Table 2). 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 [12]. 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 [15]. 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 [33]. Other work suggests that the detectability of samples written in different programming languages varies across detection methods [31]. 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 [5,27], 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. The final distribution of programming languages can be seen in Table 5.

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 (see Appendix C.5). A log-normal distribution was chosen, because it closely mirrors length distributions found in reality, and many unfiltered distributions exhibit similar characteristics (see Appendix C.1). 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 (see Figure 9).

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

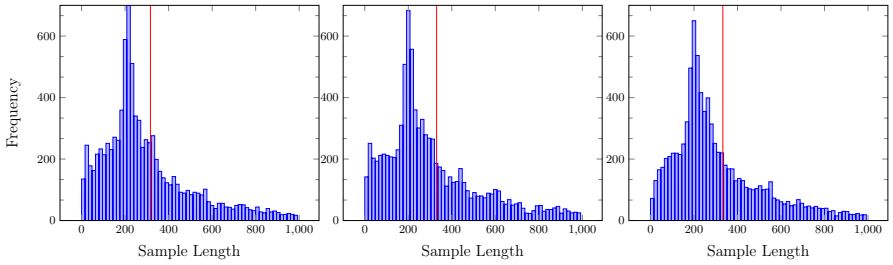


Fig. 1: Generated Python code length distribution of APPS (Left), CodeSearchNet (Middle), and Stack Overflow Post (Right) with a bin size of 16. Length in number of tokens using the cl100k base TikToken tokenizer.

In total, we generated 113 776 code samples across 6 datasets and 17 LLMs; a list of the LLMs used is provided in Table 3. 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. For a more detailed list of the generated sample counts, see Table 4 in the Appendix. 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.[10] 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 [6].
The temperature value and sampling strategy used by the LLM can have a large impact on the detectability of a code sample [28]. 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

[10] 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	**Human**	**Machine-Generated**	**Avg. Ratio**
	Total	**Total**	
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

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 were taken from previous work [28]. A total of four combinations of $T = 0.97$, $T = 0.2$, $p = 0.95$, and $p = 0.5$ were tested.

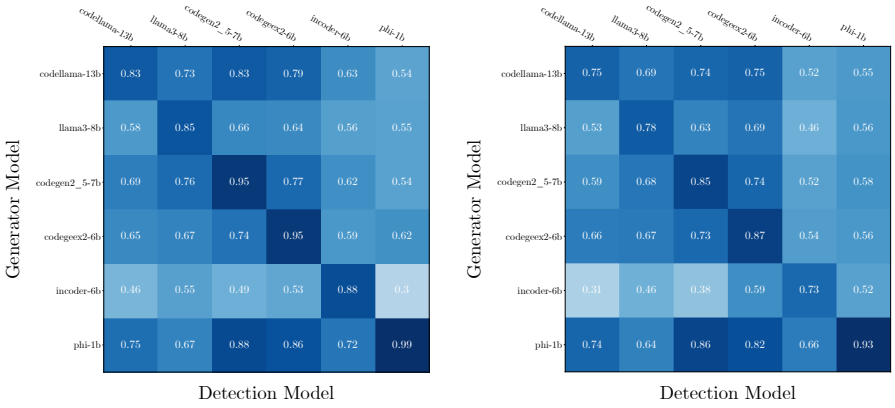
After generation, it is to be expected that some variance between the dataset distributions is introduced. Since models behave differently from one another and the code may be semantically similar but structurally different, the code lengths will vary as well. Although the distributions are not identical, the mean did not deviate significantly (see Figure 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 hours 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 seconds to generate a sample and around 270 milliseconds per detection sample. Although we had access to 96GB 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.

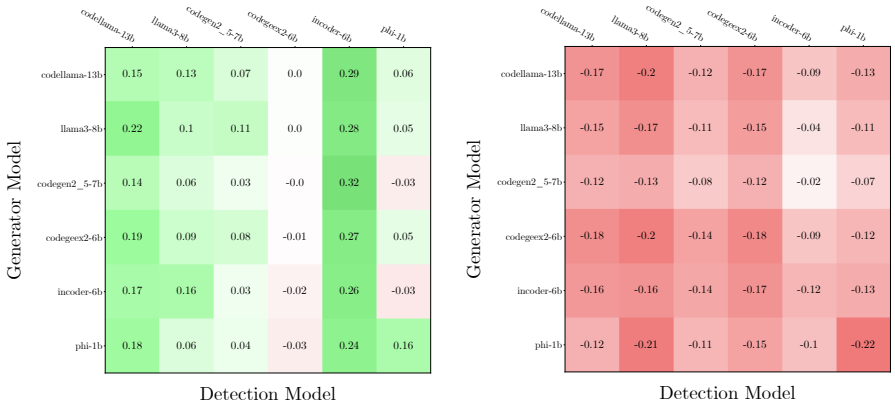
It is crucial to investigate the impact of various factors pertaining to the properties of the samples themselves. Key factors include sample length, programming language, and generation parameters, like temperature and top-p value. This yielded 12 294 388 detection samples from five detection methods (see Appendix A) and 16 LLMs.[11] 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

[11] 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.

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 contained 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. The full versions of the figures can be found in Appendix D.4. 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 Figure 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

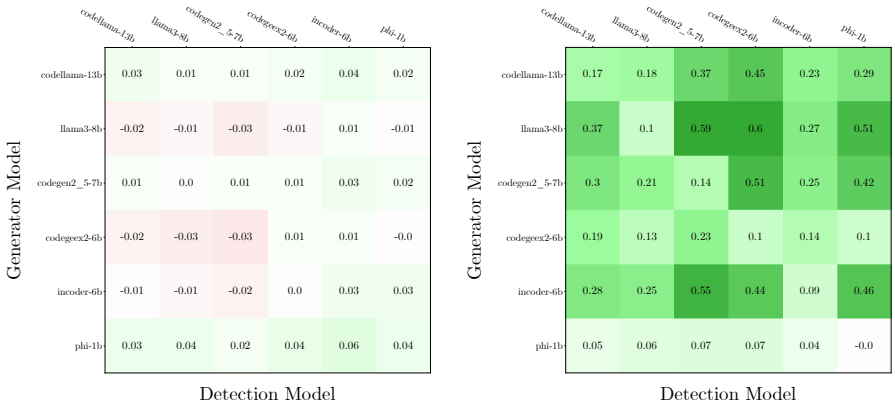
Fig. 3: (a) Python-only cross-model performance comparison between CodeSearchNet and Stack Overflow Post using rank (b) Cross-model performance comparison between LeetCode Pre and LeetCode Post using entropy. Green indicates an improvement in AUROC. Red indicates a decrease in AUROC.

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 (see Figure 12 in the Appendix), 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 Figure 3).

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 Figure 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.



(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.

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 Figure 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 (see Appendix D.2). 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.* [13] 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 (see Appendix D.1), 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. Abdin, M.I. et al.: Phi-3 technical report: A highly capable language model locally on your phone. (Preprint) arXiv:2404.14219 pp. 1–24 (2024)
2. Alemohammad, S. et al.: Self-consuming generative models go MAD. In: The Twelfth International Conference on Learning Representations (ICLR). pp. 1–13 (2024)
3. 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)
4. Carlini, N. et al.: Extracting training data from large language models. In: 30th USENIX Security Symposium. pp. 2633–2650 (2021)
5. Chakraborty, S. et al.: Position: On the possibilities of ai-generated text detection. In: International Conference on Machine Learning (ICML). pp. 1–15 (2024)
6. 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)
7. Fried, D. et al.: Incoder: A generative model for code infilling and synthesis. In: The Eleventh International Conference on Learning Representations (ICLR). pp. 1–14 (2023)
8. 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)
9. Grattafiori, A. et al.: The llama 3 herd of models. (Preprint) arXiv:2407.21783 (2024)
10. Gunasekar, S. et al.: Textbooks are all you need. (Preprint) arXiv:2306.11644 pp. 1–24 (2023)
11. 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)
12. Hendrycks, D. et al.: Measuring coding challenge competence with APPS. In: Neural Information Processing Systems Track on Datasets and Benchmarks. pp. 1–12 (2021)
13. 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)
14. Hu, X. et al.: RADAR: Robust AI-text detection via adversarial learning. In: Advances in Neural Information Processing Systems (NeurIPS). pp. 1–12 (2023)
15. Husain, H. et al.: Codesearchnet challenge: Evaluating the state of semantic code search. (Preprint) arXiv:1909.09436 pp. 1–6 (2019)
16. 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)
17. Li, B. et al.: Resilient watermarking for LLM-generated codes. (Preprint) arXiv:2402.07518 pp. 1–18 (2024)
18. Lozhkov, A. et al.: Starcoder 2 and the stack v2: The next generation. (Preprint) arXiv:2402.19173 pp. 1–54 (2024)
19. 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)

20. Nijkamp, E. et al.: Codegen: An open large language model for code with multi-turn program synthesis. In: The Eleventh International Conference on Learning Representations (ICLR). pp. 1–13 (2023)
21. OpenAI et al.: OpenAI o1 system card. (Preprint) arXiv:2412.16720 pp. 1–52 (2024)
22. OpenAI et al.: GPT-4 technical report. (Preprint) arXiv:2303.08774 pp. 1–78 (2023)
23. 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)
24. 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)
25. Qiang, Y. et al.: Learning to poison large language models during instruction tuning. (Preprint) arXiv:2402.13459 pp. 1–14 (2024)
26. Rozière, B. et al.: Code llama: Open foundation models for code. (Preprint) arXiv:2308.12950 pp. 1–26 (2023)
27. Sadasivan, V.S. et al.: Can AI-generated text be reliably detected? Preprint arXiv:2303.11156 pp. 1–16 (2023)
28. Shi, Y. et al.: Between lines of code: Unraveling the distinct patterns of machine and human programmers. (Preprint) arXiv:2401.06461 pp. 1–12 (2024)
29. Solaiman, I. et al.: Release strategies and the social impacts of language models. (Preprint) arXiv:1908.09203 pp. 1–71 (2019)
30. Tian, H. et al.: Is ChatGPT the ultimate programming assistant - How far is it? (Preprint) arXiv:2304.11938 pp. 1–22 (2023)
31. 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)
32. Wu, J. et al.: A survey on LLM-generated text detection: Necessity, methods, and future directions. Computational Linguistics pp. 1–66 (2025)
33. Yang, X. et al.: Zero-shot detection of machine-generated codes. (Preprint) arXiv:2310.05103 pp. 1–11 (2023)
34. Yu, Z. et al.: Wavecoder: Widespread and versatile enhancement for code large language models by instruction tuning. In: 62nd Annual Meeting of the Association for Computational Linguistics (ACL). pp. 5140–5153 (2024)
35. Zhao, H. et al.: Codegemma: Open code models based on Gemma. (Preprint) arXiv:2406.11409 pp. 1–11 (2024)
36. Zheng, Q. et al.: Codegeex: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X. In: 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD. pp. 5673–5684 (2023)

A Zero-Shot Detection Methods

All of the zero-shot detection methods investigated in this paper utilise probability distributions generated by underlying LLMs. To be precise, the detection methods operate with the non-normalised vector outputs (logits) of each input token, resulting in a matrix with dimensions *vocabulary size* $\times$ *input token count*. When normalised, each column can be seen as the conditional probability distribution of the tokens at that position.

The main goal of this paper was to determine the efficacy of various zero-shot detectors from the domain of natural language detection for code detection. Due to computational constraints, we could only thoroughly analyse a few. Despite older methods showing weaker detection performance in code detection, they are included here as a reference for both the progression of the field and as a comparison to detection scores found in the literature.

Log-Likelihood

One of the first proposed methods of detecting AIGC was using the log-likelihood of a sample given an LLM [8,29]. Intuitively this makes sense, as a model should assign a high log-likelihood to samples it has generated and lower values to foreign samples. This introduces a common problem amongst many detection methods. Under this assumption, the LLM cannot differentiate between a human sample and a foreign machine-generated one created by another LLM. This should in theory lead to a weak cross-model/black-box performance of this detection method. This is substantiated during analysis in Section 5. Under the hood, the log-likelihood of a model is determined by calculating the negative loss of the model. This is because negative cross-entropy (loss) is equivalent to log-likelihood.

Entropy

Entropy detection utilises a LLM's predicted entropy of a given sample [8]. A high entropy is equivalent to a lower perplexity, which is often used to measure how "unexpected" the input is to the model [11]. This should not be confused with cross-entropy despite being functionally similar. The key difference is that only the logits of the LLM are utilised in the calculation of entropy. In cross-entropy, the target labels derived from the sample are used as well.

Rank

The rank a LLM assigns a sample is determined by the average token position in the LLM's normalised output logit [8]. Simply put, at each token position of the sample, the LLM assigns the token a rank in order of likelihood that it would output it. A rank of $n = 1$ indicates that the token is the last one the LLM would have chosen, while a rank of $n = vocab\_size$ would indicate that the LLM would have chosen that exact token [8].

DetectGPT

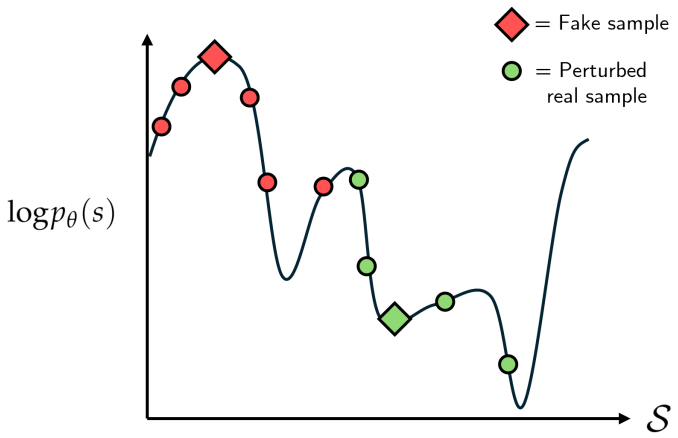


Fig. 5: Image depicting log-likelihood curvature utilised by DetectGPT [19]. Here, $\mathcal{S}$ represents the space of all possible samples. Fake samples are sampled from $p_\theta(s)$ while real samples are sampled from $p_h(s)$. Figure adapted from DetectGPT [19].

DetectGPT [19] was the first of its kind to introduce the notion of detection via perturbation. Perturbation methods are more effective because they sample neighbourhoods instead of singular points in the model's probability space. This gives rise to the possibility of detecting patterns, such as negative curvature discovered by DetectGPT, in these high-dimensional spaces. The negative curvature observation is attributed to the way LLMs sample their probability distribution. AIGC samples tend to lie on maxima in the model's log-likelihood space and perturbed samples lower around them [19]. Human samples on the other hand do not follow the model's distribution and will often lie in more rugged areas in the model's log-likelihood space (see Figure 5). To obtain a high resolution of the neighbouring topology, a large number of perturbations need to be generated and evaluated. The slow-down is two-fold when another LLM generates the perturbations. During preliminary analysis, it was determined that DetectGPT performed significantly worse than other newer detection methods on a test set. Because of this and its slow detection speed, it was removed from the paper.

Fast-DetectGPT

Fast-DetectGPT [3] extends DetectGPT's findings. Like its predecessor, it is a perturbation-based model that utilises the curvature of samples in a probability space to aid in detection but differs in the space that it probes. Instead of determining the curvature in the log-likelihood space of the model, the curvature is estimated in the conditional probability space. The paper goes into detail about the rationale. Simply put, instead of generating perturbations and evaluating them separately, the posterior of the conditional probability can be modified instead [3]. This is easy to do as this is simply a resampling of the probabilities integrated into the logits of the LLM. That means that the logits only need to be generated once instead of hundreds of times compared to DetectGPT [3]. Unlike in log-likelihood space, it was observed that AIGC tends to lie in points with positive curvature. Due to the increase in efficiency, 10,000 samples can be generated to better approximate the curvature. This leads to a superior performance over DetectGPT. The paper also derives an analytical form of the resampling step that further speeds up the detection process. This has led to a measured speed-up of 340 times over regular DetectGPT [3].

Binoculars

Binoculars [11] is a novel zero-shot detection method that departs from perturbation-based detection methods. The authors of Binoculars highlight the issue of human written prompts impacting model perplexity, which they name "the capybara problem". Note that: Here, perplexity is defined as an exponentiated cross-entropy [11]. In their paper, they give an example of a prompt that elicits a response that would have high perplexity, should the prompt be omitted, which would lead to misclassification by perplexity-based detectors. Binoculars addresses this issue by introducing cross-perplexity normalisation. The idea is to reset the perplexity baseline using a secondary observer model. Their Binoculars score is the ratio of perplexity to cross-perplexity [11]. The score can be reinterpreted as the log-likelihood divided by the summed negative cross-entropy of the sample. Despite its seeming simplicity, Binoculars outperforms many of the SoTA detection methods in the domain of natural language detection. It is also important to note that the values output by binoculars have been flipped from the reference implementation. We did this due to the average AUROC being significantly below 0.5 indicating that the detector was consistently assigning samples to the wrong class. This effectively means that $1 - AUROC$ was calculated in the heat maps.

B Additional Tables

Table 2: List of the datasets, models (Generator + Detector), and the detection type (Zero-shot, classifier, etc.) of various code detection papers.

Paper	Detection Type	Datasets/ Sources	Programming Languages	Models
Evaluating AIGC Detectors on Code Content [31]	Zero-shot, Proprietary	Stack Overflow, CodeSearchNet, APPS, CONCODE	Python, Java, Javascript, PHP, Ruby, Go	ChatGPT
Assessing AI Detectors in Identifying AI-Generated Code: Implications for Education [24]	Zero-shot	Python Coding Questions (Quescol), Coding Problems (Kaggle), Leetcode	Python	ChatGPT
Detecting ChatGPT-Generated Code Submissions in a CS1 Course Using Machine Learning Models [13]	Fine-tuned	CodeWorkout	Java	ChatGPT
Resilient Watermarking for LLM-Generated Codes [17]	Watermarking	MBPP, APPS	Python	GPT-3.5, GPT-4, StarCoder
Zero-Shot Detection of Machine-Generated Codes [33]	Zero-shot	CodeContests, APPS	Python, Java	GPT-3.5-turbo, GPT-4, text-davinci-003
Between Lines of Code: Unraveling the Distinct Patterns of Machine and Human Programmers [28]	Zero-shot	CodeSearchNet, The Stack	Python	Incoder, Phi-1, StarCoder, WizardCoder, CodeGen2, CodeLlama

Table 3: Summary of the LLMs used in the pipeline. (Yes) indicates that the option exists but was not used.

Model	**Parameters**	**Multilingual**	**Instruct**	**Release/ Cut-off**	**Dataset**	**Human-Eval**	**Paper**
Code Llama	7B, 13B	Yes	Yes	July 2023	Public Online Data	34.8%, 42.7%	[26]
Llama 3	8B	Yes	Yes	July 2023	Public Online Data	62.2%	[9]
CodeGen 2.5	7B	Yes	(Yes)	July 2023	The Stack v1.2	28.36%	[20]
CodeGeeX2	6B	Yes	No	July 2023	The Pile, CodeParrot, GitHub	35.9%	[36]
StarCoder 2	3B, 7B	Yes	No	February 2024	The Stack v2	31.7%, 35.4%	[18]
CodeGemma	7B	Yes	Yes	April 2024	Public Online Data	56.1%	[35]
WaveCoder-Ultra	6.7B	Yes	(Yes)	April 2024	CodeSearch-Net (Subset)	79.9%	[34]
Incoder	1.3B, 6.7B	Yes	No	April 2022	GitHub, Stack-Overflow	8%, 15%	[7]
Phi-3 mini	3.8B	Yes	Yes	April 2024	Public Online Data, Synthetic Data	58.5%	[1]
Phi-1	1.3B	No	No	June 2023	The Stack v1.2	45%	[10]
o1 mini	-	Yes	Yes	October 2023	Public Online Data, Proprietary Data	92.4%	[21]

Table 4: Overview of the number of generated samples per generator and dataset.

Model	**Stack Overflow Pre**	**Stack Overflow Post**	**APPS**	**CodeSearch -Net**	**Leetcode Pre**	**Leetcode Post**
Human	1700	1700	561	561	500	500
Codellama-13b	1697	1696	561	559	495	497
Codellama-instruct-13b	1662	1662	460	526	464	445
Llama3-8b	1688	1688	540	508	497	483
Llama3-instruct-8b	1678	1680	552	556	497	489
Codellama-7b	1698	1698	560	559	500	500
Codellama-instruct-7b	1679	1680	478	542	498	469
Codegen2_5-7b	1700	1700	554	561	499	500
Codegeex2-6b	1697	1699	561	558	500	500
Starcoder2-7b	1700	1700	561	561	500	500
Codegemma-instruct-7b	1685	1687	497	542	467	454
Wavecoderultra-7b	1611	1610	522	476	482	450
Incoder-6b	1700	1700	555	561	500	500
Phi3mini4k-instruct-4b	1692	1693	559	557	498	494
Starcoder2-3b	1700	1699	561	561	500	500
Phi-1b	257	251	527	540	109	93
Incoder-1b	1700	1700	545	561	500	500
o1-mini	-	1700	556	561	500	500

Table 5: Distribution of programming languages across various datasets.

Programming Languages	**APPS**	**CodeSearchNet**	**Stack Overflow (Pre/Post)**	**Leetcode (Pre/Post)**
Python	561	561	561	195
Java	-	-	340	139
Javascript	-	-	255	37
C#	-	-	255	-
C++	-	-	228	129
Go	-	-	34	-
Rust	-	-	27	-
Total	561	561	1700	500

C Additional Information: Datasets

Here, we expand on key topics discussed in the paper, providing specific implementation details, dataset insights, and essential procedures to offer a deeper understanding of our approach.

C.1 Unfiltered Distributions

The following figures represent the raw sample distributions of the datasets before the processing stage. Here the importance of the processing stage is highlighted, as the discrepancy between the various datasets becomes apparent.

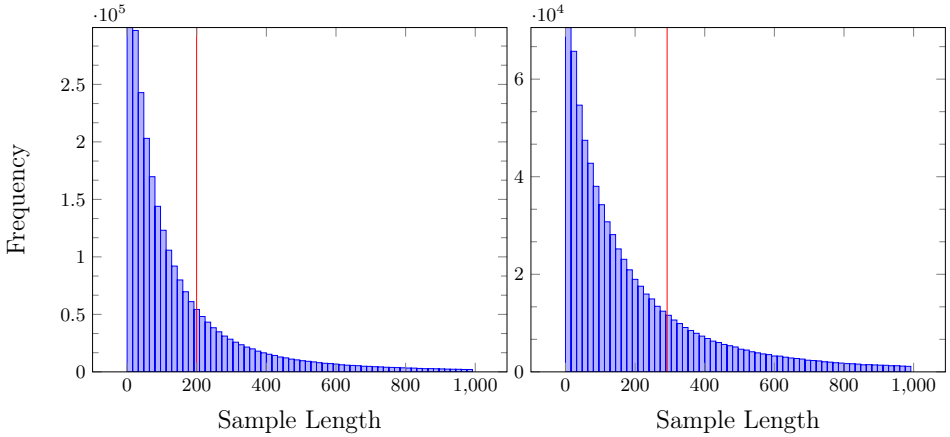


Fig. 6: Code length distribution of Stack Overflow code samples with a bin size of 16. Stack Overflow Pre (Right) and Stack Overflow Post (Left). Length in tokens using the cl100k base TikToken tokenizer.

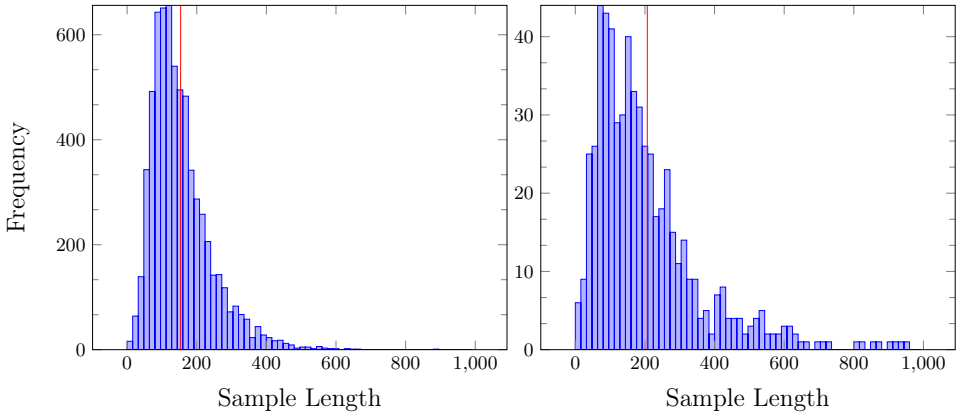


Fig. 7: Code length distribution of Leetcode code samples with a bin size of 16. Leetcode Pre (Left) and Leetcode Post (Right). Length in tokens using the cl100k base TikToken tokenizer.

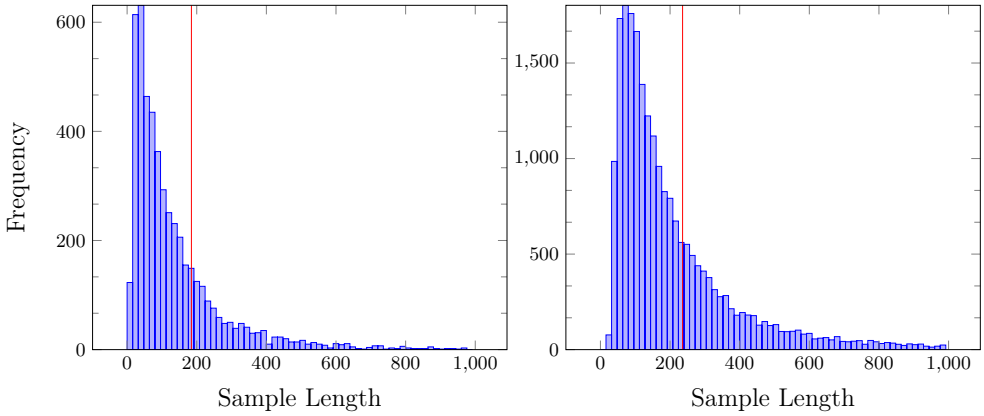


Fig. 8: Code length distribution of APPS (Left) and CodeSearchNet (Right) with a bin size of 16. Length in tokens using the cl100k base TikToken tokenizer.

C.2 Filtered Distributions

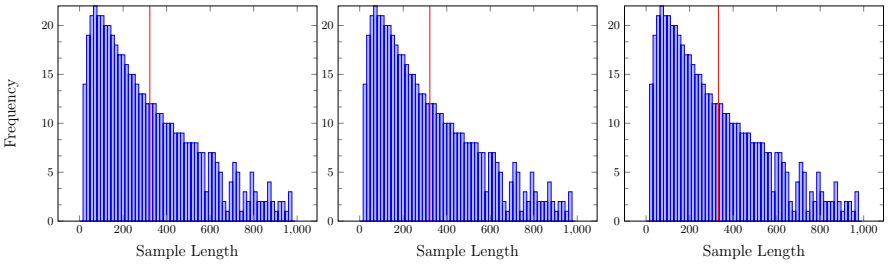


Fig. 9: Python code length distribution of APPS (Left), CodeSearchNet (Middle), and Stack Overflow Post (Right) with a bin size of 16. Length in tokens using the cl100k base TikToken tokenizer.

C.3 Framework Dataset Example

```python
from codetector.samples import CodeSample
from codetector.dataset import XMLDataset

class TestXML(XMLDataset):

    def getContentType(self):
        #Define the object type contained in the dataset.
        #This allows for serialisation into arbitrary file formats.
        return CodeSample

    def preProcess(self):
        #Called when loading a dataset for the first time.
        pass

    def getTag(self):
        #The tag of the dataset.
        return 'test_xml'
```

Fig. 10: Example implementation of a XML dataset that contains code samples.

C.4 LeetCode Problem Set Bounds Calculation and Data Collection

As previously stated, to be able to utilise samples from LeetCode, we had to determine what samples were produced before and after our cut-off date. We achieved this by collecting existing sitemap XML files archived by the Wayback Machine and counting the number of problems available at that date. Plotting and fitting a linear function to the data, we get the data presented in Figure 11. 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.

Additionally to the problem set bound calculation, we had to build a custom webscraper to collect the newest LeetCode problems that were not contained in any existing publicly available datasets. We achieved this by utilising a selenium webdriver that would use the sitemap to go to the new problems, extract the problem description and other metadata; then it would download several top solutions and extract the code and programming language from the posts.

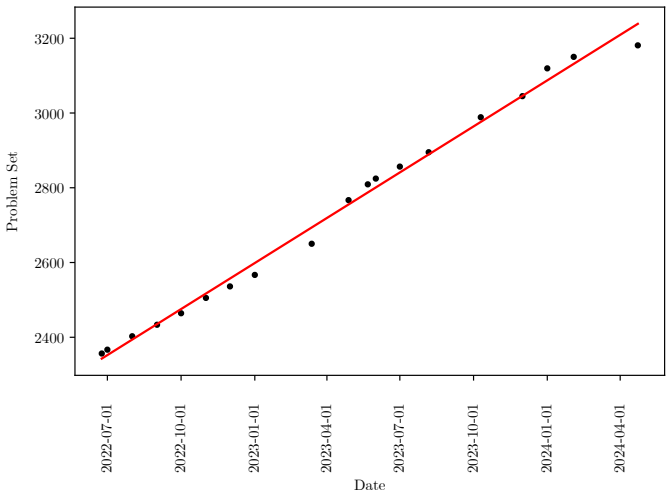


Fig. 11: Scatter plot of LeetCode sitemap dates and the number of available problems at that date. Red line represents a linear regression used to later determine cut-off dates.

C.5 Log-Normal Distribution

To closely mirror code length distributions found in reality, the code samples were sampled to match a log-normal distribution. This was achieved using the following formula:

$$f_{dist}(binEdge) = \left\lfloor \frac{1}{\frac{binEdge+k}{c} \cdot \sigma\sqrt{2\pi}} e^{-\frac{\ln^2(\frac{binEdge+k}{c} - \mu)}{2\sigma^2}} \cdot d \right\rfloor \tag{1}$$

The values $\sigma = 2.1$, $\mu = 1.3$, $c = 169.4$, $k = 217$ were used for all distributions. d was chosen accordingly so that:

$$\sum_{i=0}^{63} f_{dist}(16i) = \text{Programming language target total} \tag{2}$$

For example, a target total of 561 samples for Python was chosen. After determining a suitable value for d, the distribution overlap can be generated and fitted to a scaled log-normal distribution.

C.6 Prompts

Labelling Prompt The following system prompt was given to GPT-3.5-turbo to label samples from the Stack Overflow datasets.

```
You are Code GPT. You will receive a title and a long text
↪  containing lots of technical information and possibly
↪  code.You will output a JSON formatted string with the
↪  following exact format where you parse the code, if present,
↪  from the given text. Additionally, you should add the
↪  programming language (using the given labels) and semantic
↪  meaning of the code if present. Keep the semantic meaning
↪  short but long enough to describe the functionality (maximum
↪  three sentences). When describing the semantic, write it in
↪  the style of a comment describing the code (Do not format it
↪  as a comment). The programming language labels are the
↪  following: python, javascript, c, c++, java, rust, go, c#.
↪  If a language is not in the given list of labels, label it
↪  as generic.

The format for the JSON is:
{"programming-language": <The programming language of the code>,
↪  "semantic": <The meaning behind the code>}

Return the JSON {"no-code": true} if the text contains no code.

Follow the exact instructions above.
```

Generation Prompts In the completion and instruction prompts, used to generate the code samples, the values inside of $<\ldots>$ were substituted. The completion prompt was additionally converted to a comment format using the appropriate styling for the language.

*Completion

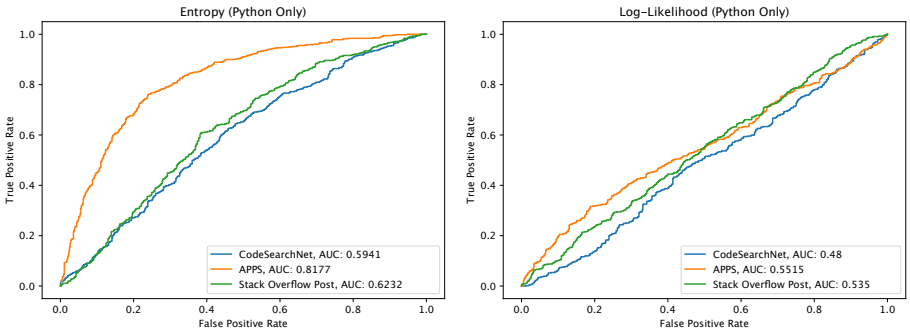
```
language: <The programming language>
<The prompt>
```

*Instruction

```
In <natural language name of programming language>, write some
↪  code with the following functionality: <The prompt>.
Do not explain the code, only return the code and comments if
↪  necessary. DO NOT REPEAT YOURSELF! Only explain things in
↪  the comments of code. Otherwise do not write any plain text.
↪  Start with the code here:
```

D Additional Information: Analysis

Here, we provide further insights into the properties of the generated samples and LLMs, analysing ROC curves, relative token frequencies, the impact of programming languages on detection, and correlations between various factors and AUROC.



(a) Dominating (b) Non-dominating

Fig. 12: Receiver Operating Characteristic (ROC) curves of various detection methods. (a) ROC curve of entropy detector with Phi-3 4k Mini 3.8B as the generator model and StarCoder 2 7B as the base model showing dominance with a large AUROC difference (b) ROC curve of log-likelihood detector with Code Llama 13B as both a base model and generator model showing no dominance with a small AUROC difference across datasets.

D.1 Token Frequency Analysis

Additionally, we inspected the token frequency distributions of the generated samples. To visualise the preference of specific tokens that LLMs may have, we plotted the individual LLM token frequencies against their human frequency (see Figure 13). Values above the identity indicate a stronger preference for the LLM over human samples and vice versa for below. The top 15 tokens were highlighted using a simple heuristic based on the product between the perpendicular distance to the identity and the distance to the origin. The tokenizer plays a large role in the distribution and selection of tokens in the plot. To enhance human readability, we utilised the Natural Language Toolkit (NLTK) TreebankWordTokenizer tokenizer that uses regular expressions to break down strings into token sequences. Inspecting the various plots, a few observations can be made. Generally, the majority of the LLMs share a large number of the top tokens assigned by the heuristic. They also have the tendency to utilise more comment tokens like '//' or '#' than humans. The token 'the' is often to be

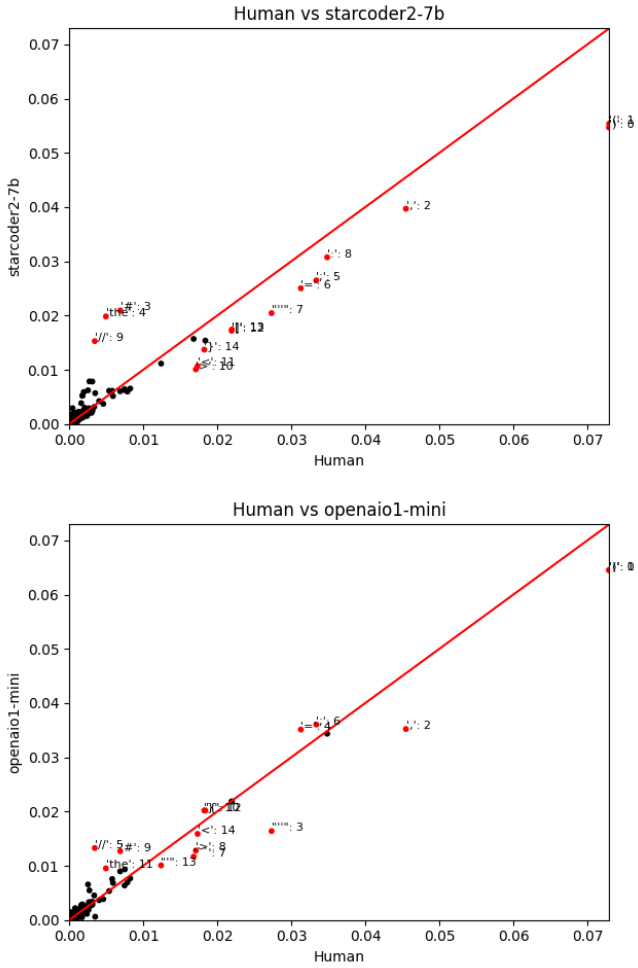
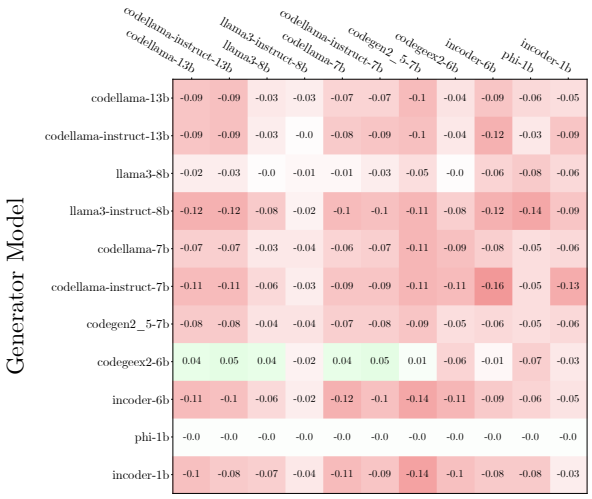


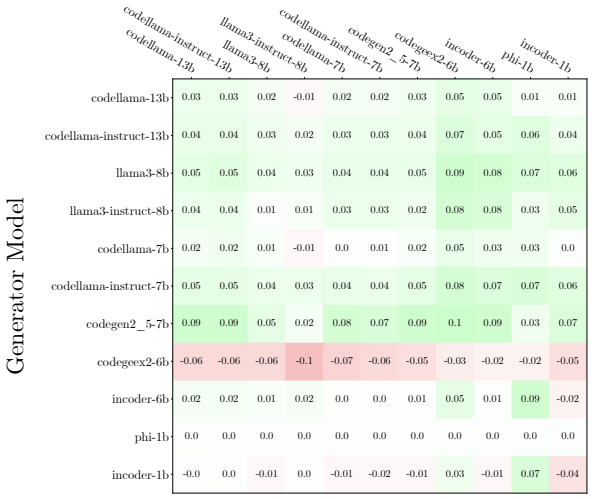
Fig. 13: Correlation plot between the relative token frequencies of AIGC from two LLMs against human token frequencies.

found in close proximity indicating that the token is probably frequently used inside of comments. Like the rest of the figures presented in this paper, the token frequency plots are supplied as SVGs and PNGs, alongside the code to generate them from scratch.

D.2 Programming Language Detection Comparison



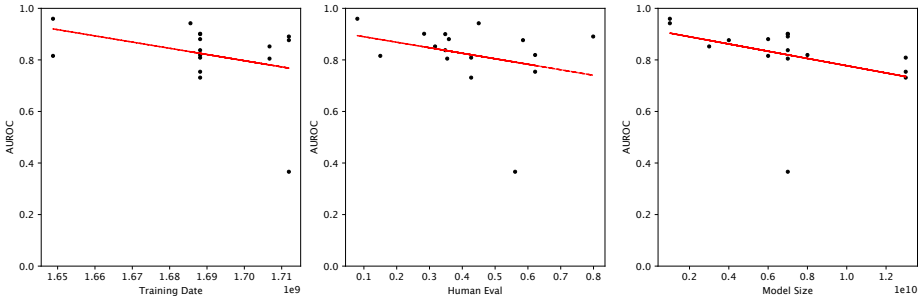
(a) C# vs Python



(b) Java vs Python

Fig. 14: (a) Cross-model performance comparison between C# and Python using Binoculars (b) Cross-model performance comparison between Java and Python using entropy detector.

D.3 AUROC Correlation



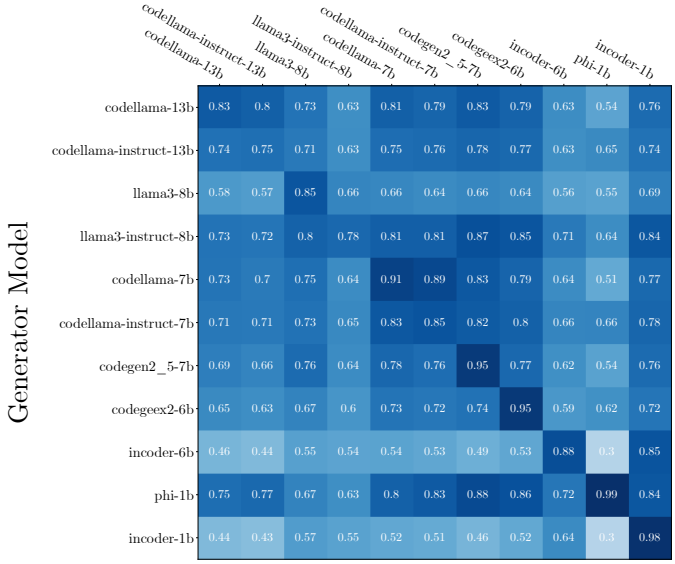
(a) Training Date (b) Human Eval (c) Model Size

Fig. 15: Correlation plot between AUROC scores and various properties. AUROC scores were calculated using Binoculars in a white box context.

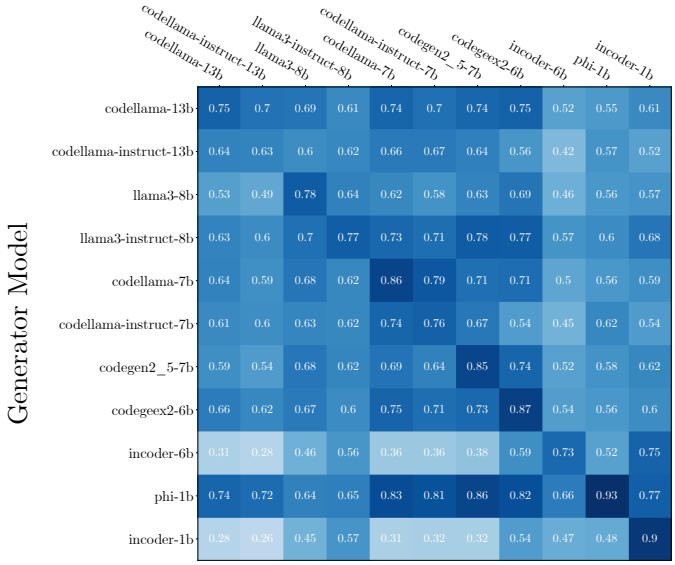
Plotting AUROC scores against various factors offers a way to extrapolate the future prospects of detection in the field. To be precise, we investigated the relation of AUROC to training date, HumanEval scores, and model size. Newer detection methods, such as Fast-DetectGPT and Binoculars, show a slight downward trend for models with either newer, larger, or higher HumanEval scores (see Figure 15). However, it is important to note that the newest, largest and best model tested, o1 mini, is among one of the "easiest" models to detect.

D.4 Full-sized Heat maps

We append a list of full-sized heat maps used in the paper. For heat maps that contain values of all 17 LLMs please see our GitHub repository.

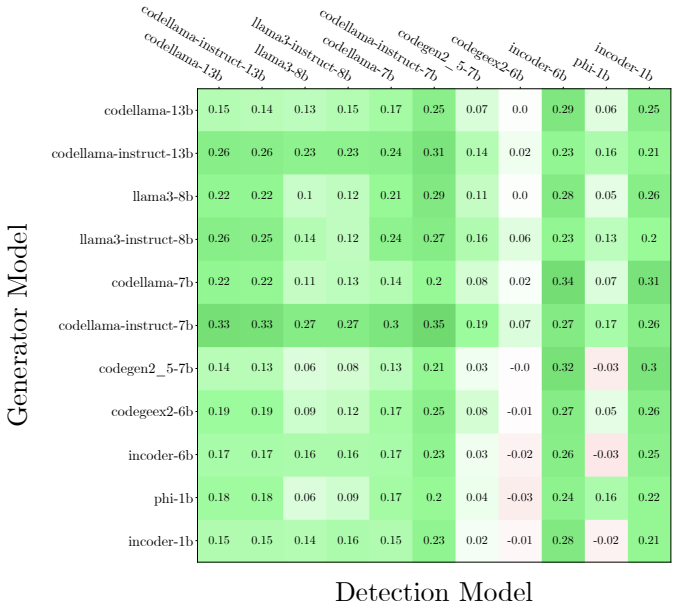


(a) Fast-DetectGPT

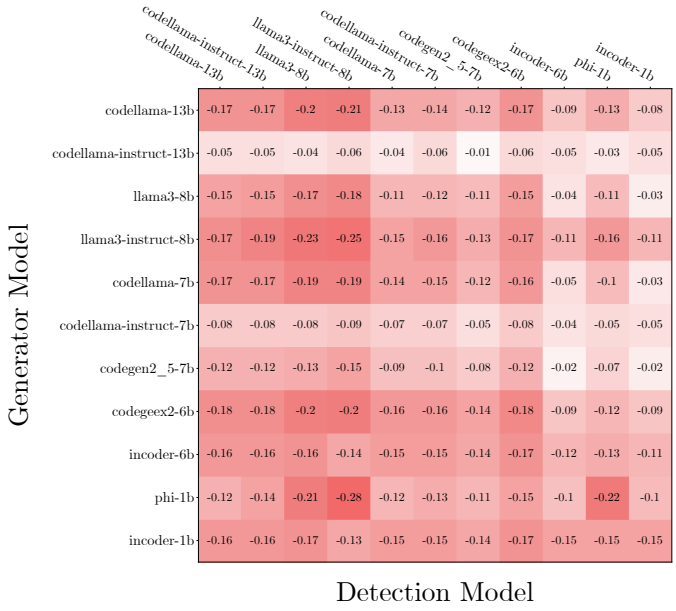


(b) Binoculars

Fig. 16: Cross-model performance on two datasets. (a) Fast-DetectGPT on Stack Overflow Post and (b) Binoculars detector on APPS.

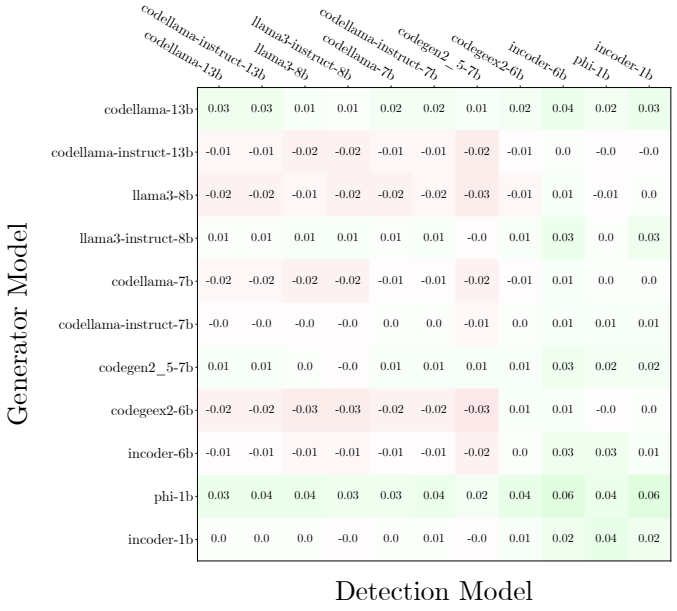


(a) Rank



(b) Entropy

Fig. 17: (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.



(a) Log-likelihood

(b) $T = 0.97$, $p = 0.95$ vs $T = 0.2$, $p = 0.5$

Fig. 18: (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.