A Simple, Yet Effective Approach to Finding Biases in Code Generation

Spyridon Mouselinos
University of Warsaw
s.mouselinos@uw.edu.pl

Mateusz Malinowski
DeepMind
mateuszm@deepmind.com

Henryk Michalewski
Google, University of Warsaw
henrykm@google.com

Abstract

Recently, high-performing code generation systems based on large language models have surfaced. They are trained on massive corpora containing much more natural text than actual executable computer code. This work shows that current code generation systems exhibit undesired biases inherited from their large language model backbones, which can reduce the quality of the generated code under specific circumstances.

To investigate the effect, we propose the "block of influence" concept, which enables a modular decomposition and analysis of the coding challenges. We introduce an automated intervention mechanism reminiscent of adversarial testing that exposes undesired biases through the failure modes of the models under test. Finally, we demonstrate how our framework can be used as a data transformation technique during fine-tuning, acting as a mitigation strategy for these biases.

1 Introduction

Large language models (LLM) have recently demonstrated their ability to generate code (Li et al., 2022; Brown et al., 2020; Wang et al., 2021) or solve challenging programming/math tasks on par with human coders (Li et al., 2022; Lewkowycz et al., 2022b; Chowdhery et al., 2022a); these models are trained with the data-driven paradigm. On the other hand, an increasing body of work also questions whether the data-driven approach leads to acquiring reasoning skills (Piekos et al., 2021; Zhang et al., 2022; Mouselinos et al., 2022), showing that if left alone, it might not be sufficient for achieving truly human-level performance on tasks such as logical or visual reasoning. In many studied cases, models still rely on various hints in their reasoning process. This work extends the results above, i.e., the lack of reasoning capabilities, to the code generation domain. More specifically, we devise a framework that automatically identifies subtle cues a code generation model might exploit. Changes or removal of those cues stands as a reasoning test towards the generational capabilities of the model at hand.

We presume that the reasoning process of code generation models should remain invariant under changes that still provide enough context or pose little, if any, additional challenge to a human coder. To this end, we propose an automatic and model-agnostic framework that modifies the following: (1) function names, (2) keywords in a problem specification, and (3) examples provided in the problem prompt. We refer to these parts as Blocks-Of-Influence; see Figure 1. Each block contributes partially to the context needed for correct completion. We show that minor modifications of these blocks are sufficient to "fool" LLM-based code generation methods.

Our results reveal biases such as keyword preference and memorization effects, which can be identified across multiple models. During our experiments, we ensure that any modifications maintain the global semantics of the coding challenge. This is achieved through a context-aware filtering mechanism that guarantees any information altered or removed still exists and/or can be deducted from the remaining unaltered part.

Contributions. The main contributions of our work can be summarized in three points.
First, we propose a novel automated framework that identifies possible biases in code generation models. Our framework removes subtle hints, introducing minimal changes such as keyword replacement or partial code-block omission, ultimately acting as an adversarial test. Since the framework operates on a data level, it is agnostic to the model's structure and internal workings. The framework can be easily adjusted to any input format or programming language.
Second, we introduce the "*Blocks of Influence*" concept. We suggest that every instance of a typical

Findings of the Association for Computational Linguistics: ACL 2023, pages 11299–11329

coding challenge can be analyzed into three parts (blocks). Each part is correlated with a different method of hinting and is used as a target of our transformations. A model's reasoning process is informed by all three blocks, making them perfect analyzing tools for cases of failing code generation. ***Third***, we explore new ways of mitigating biases during code generation. In Section 6, we study the effects of adversarial training against our proposed perturbations, and the benefits of including examples with longer descriptions during fine-tuning. Our results show that combining these techniques leads to more accurate code completions.

2 Related Work

Our approach is inspired by works of various research directions, which we briefly describe here.

Solving coding and math challenges. The emergent abilities of large language models to generate, summarize and translate textual information, have recently sparked interest in their aptitude for math, logic, and programming challenges. Tasks such as code-completion (Chen et al., 2021; Shin et al., 2019; Hendrycks et al., 2021a; Li et al., 2022), code summarization and code translation (Lu et al., 2021) have been proposed, with models constantly progressing towards near-human performance. Similarly, (Hendrycks et al., 2021b; Saxton et al., 2019; Ling et al., 2017; Amini et al., 2019) have proposed tests measuring a model's ability to perform math and logic, ranging from school problems to competition-grade challenges. Impressive results in multiple programming languages have also been achieved by decoder-only works (Brown et al., 2020; Chen et al., 2021). Fried et al. (2022) created the first generative model to perform infilling using a novel masking objective. Finally, massive-scale models such as (Chowdhery et al., 2022b; Lewkowycz et al., 2022a) demonstrated breakthrough capabilities in language, reasoning, and code tasks achieving state-of-the-art performance in multiple domains simultaneously.

Social biases in large language models. Trained on ever-increasing amounts of publicly available data, large language models have been studied for adopting social biases commonly found among humans. Wallace et al. (2019) show that generative models can be conditioned to produce toxic content, with the use of nonsense, adversarial prefixes. Similarly, Liang et al. (2021) suggest that models might adopt biases and social stereotypes found among their training data and provide ways to apply fairness during generation. Countermeasures have been proposed by (Zhao et al., 2021; Liu et al., 2022), claiming that sanitized zero-shot examples contribute to mitigating biases during generation.

Probing reasoning through cognitive biases. There have been notable attempts to systemize intelligence and reasoning as concepts (Legg, 2008; Chollet, 2019), yet a few recent works try to approach reasoning, through the analysis of failure modes, caused by biases in deep learning models. Glockner et al. (2018) suggest that natural language inference systems can be easily fooled with a single hypernym/hyponym swap, exhibiting an bias towards specific word choices. Similarly, Lin et al. (2020) prove that numerical commonsense reasoning in LLMs is heavily biased by adjectives describing the object of interest. Concerns against the current data-driven methods have been expressed by Razeghi et al. (2022), pointing out that LLMs are more accurate on mathematical challenges that involve terms significantly more frequently in their pre-training dataset. Piekos et al. (2021) claim that LLMs can answer math and logic questions without understanding the rationale behind them, relying blindly on the existence of specific keywords. We place our work in this line of research, provoking and studying the failures of LLMs under reasoning-heavy coding tasks. Our main goal consists of identifying cognitive bias sources, i.e., words, structures, or co-occurrence patterns, that exist in current LLMs, and lead to systematic failures of rationale.

Adversarial methods and Language Processing. NLP community developed excellent methods to prepare adversarial tasks, including the TextAttack framework (Morris et al., 2020) and sophisticated techniques to elicit adversarial examples from humans, as in Talmor et al. (2022), though our work seems to be the first focused on the disciplined construction of adversarial examples for code.

3 Benchmarks

In this section, we describe the datasets used in our experiments. We employed widely used coding challenges HumanEval (HE) and MBPP and a more complex dataset with lengthy descriptions of problems (DMCC). More information about the datasets can be found in the Appendix 10.2.

HumanEval (HE). This is a human-curated problem-solving dataset described in Chen et al. (2021). It consists of 164 original programming challenges assessing language comprehension, al-

```
def rev_list_take2(a: List):
    "Reverse the list and return the second item."
    Examples:
    >>> rev_list_take2(['a',2,'apple'])
    2
```

```
def func(a: List):
    "Reverse the list and return the second item."
    Examples:
    >>> rev_list_take2(['a',2,'apple'])
    2
```

Figure 1—**Left:** The three blocks of influence: *Name Block* in red, *Description Block* in green and *Example Block* in blue. **Right:** We demonstrate three possible transformations, one for each block: Swap the function name with "func", remove keywords, and remove examples. Transformations can be applied alone or in combinations of two as described in Section 5.2

gorithms, and simple mathematics. Each problem is presented as an incomplete function, accompanied by a docstring. The docstring contains the task and a few example cases. For each task, we are provided with a set of unit tests. A task is considered solved when all unit tests are passed.

Mostly Basic Python Problems (MBPP). Introduced in Austin et al. (2021), it contains 974 short Python functions designed to be solved by entry-level programmers. Contrary to HumanEval, each task is given through a text description rather than a docstring. Since there are no input-output examples in the prompt, we generate 3 valid pairs using the code solutions provided. c MBPP challenges models to perform tasks of imperative control flow, requiring loops and conditionals.

DeepMind Code Contests (DMCC). Is the highly challenging dataset proposed by Li et al. (2022). The dataset includes problems from the Codeforces platform (Mirzayanov, 2020), Description2Code (Caballero, 2016), and CodeNet (Puri et al., 2021). We used challenges written in the Python3 language of the training split for our experiments. DMCC contains long descriptions of the problems and input-output examples of the functions to be completed.

In this work, DMCC is used for its long context properties during experiments of augmented fine-tuning (Table 5). Models presented in our work achieve zero or near-zero scores on it; hence it is excluded from our perturbation analysis, with HumanEval and MBPP being more suitable targets.

4 Evaluation

Models. In our experimental setup, we test five models representing different approaches to code generation. CodeParrot (Tunstall et al., 2022a) comes with an open-source dataset and can be easily used for fine-tuning experiments due to its size. Its smaller variant (110M) achieves competitive results to other open-source LLMs at larger parameter budgets. By exploring its dataset, we tested our hypothesis that function names act as biases during code generation. Models can be heavily inspired by similarly named snippets in their training set and resort to copying whole or parts of the solution instead of performing reasoning. (See Appendix A.9) We also test the Incoder (Fried et al., 2022) model, which is trained under a novel bi-directional causal objective, being able to handle context more efficiently than its causal counterparts. Against our initial hypothesis, our methods cause significant performance drops despite the model's enhanced context-understanding capabilities (Table 3). The Bloom model (Mitchell et al., 2022) exhibits emergent abilities in multiple domains by training on massive multilingual and multi-purpose content. Despite not being a code generation model, it performs equally well with code-specific models in the same parameter budget. Theoretically, bias effects can be reduced when a model is exposed to diverse training examples. Our experiments reveal that this is still not the case under our setup, and post-training solutions are explored. CodeGen (Nijkamp et al., 2022) is a high-performing model trained in natural language understanding and code. We test its Mono variant, further fine-tuned on the Python language. Finally, we have the powerful Codex model, which can tackle most of the proposed coding challenges in the HumanEval and MBPP datasets. A list of the tested models, as well as KeyBert (Grootendorst, 2020) that is used in our framework, can be found in Table 1.

Model Name	Sizes Used
KeyBert (Grootendorst, 2020)	2M
Codeparrot (Tunstall et al., 2022a)	110M / 350M*/ 1.5B
InCoder (Fried et al., 2022)	1.6B / 6B
CodeGen (Nijkamp et al., 2022)	350M / 6B
Bloom (Mitchell et al., 2022)	560M* / 1.7B / 176B†
Codex (v1 / v2) (Chen et al., 2021)	∼175B† (Estimated)

Table 1: Models used: (*) refers to fine-tuned and (†) to API.

Performance metrics. We evaluate the functional correctness of the generated programs with the pass@k metric, introduced in Kulal et al. (2019). This metric serves as an estimator of a model's generative capabilities under a specific budget. In Chen et al. (2021), authors propose an updated un-

biased version that we adopt throughout the rest of this work. To avoid any confusion, we calculate pass@k at exactly k attempts. The average of ten runs with different seeds is presented for all experiments in Table 3. We use sampling temperatures of 0.2 / 0.8 for pass@1 / pass@100, which are the optimal values across the tested models.

5 Method

5.1 Blocks of Influence

Our method treats each coding challenge as a combination of three distinct but complementary blocks rather than a single, homogeneous input. We refer to them as *Blocks of Influence* and correlate each with a different source of bias during code generation. Taking as an example Figure 1, we challenge the model to complete a function that reverses a list and then returns its second item.

Name Block. The first block of influence, marked in red, informs the model about the function name and the names and types of the input arguments. Let us assume that initially, a model generates correct solutions to a problem. However, the model fails when we rename the function name to something unrelated to the task, e.g., *"fun"*. This failure mode indicates that neither the problem description was understood nor the model could extract a reasoning pattern from the given usage examples. We associate such cases with memorization effects, where the model relies heavily on the function name, replicating snippets from its training dataset with the same or similar names.

Description Block. The problem description stands as the second block, marked in green. Here the model is expected to form a solution by utilizing its natural language understanding capabilities. We observe that removing specific keywords from the problem description can lead to catastrophic results in model performance. It is vital that removing these keywords must not degrade the description semantics, and any information lost should be recoverable from the rest of the context. For example, in Figure 1, the removal of the word pair "the list" creates a description that is still well understandable by a human coder. We challenge the model to deduct the missing context from the word "list" in the function name and the input list type in the example given. The inability to recover the missing context is associated with an inherent preference bias, where the model relies on superficial lexical clues or frequently co-occurring terms seen during training rather than the given context to "mentally" fill any gaps.

Example Block. As the final block, we consider the examples after the problem description. They act as demonstrations, guiding the model to specific reasoning patterns. Let us consider a scenario where models cannot generate correct code when examples are absent. Arguably, more than the task and given inputs alone were needed for the model to form a proper problem understanding. In this failure mode, the provided examples act as a "reasoning tie-breaker" between proposed solutions the model can generate. Generated solutions are not entirely irrelevant but a relatively poor interpretation of the problem. For example, in Figure 2, when stripped of its examples, the model still exhibits signs of task understanding (i.e., comparing element difference to a threshold, iterating over elements). However, combining these logic parts in a meaningful manner is complex enough that the model requires additional examples to filter out faulty strategies. We associate such effects with poor reasoning abilities.

5.2 Framework

The first step involves splitting a coding challenge into the three *Blocks of Influence*. For this purpose, we utilize a regular expression module that searches for common patterns of each block's start or end. (e.g., *Name Block*: "def (...):", *Description Block*: " or """, *Example Block*: "Examples:" or $>$ / $\gg$ followed by usage of the function name).

As the next step, the *Description Block* is further analyzed to identify possible hinting keywords. Ideally, we are interested in unigrams or bigrams that provide excess information towards completing the coding task. For keyword identification, we use KeyBert (Grootendorst, 2020), an LLM tasked to perform keyword extraction and word similarity. We proceed to fine-tune KeyBert on the open-source CodeParrot dataset (Tunstall et al., 2022a) so that more code-specific suggestions are provided. For each candidate keyword, we calculate its embedding similarity with the set of words: [Python, Programming, Code, Variable], again through KeyBert. Words with cosine similarity scores under 0.7 for all the items of the set are unrelated to coding and thus filtered out. However, carelessly removing keywords can lead to non-interesting drops in performance associated with removing crucial information rather than hinting effects. Thus, an additional context-aware filtering stage is employed

to validate that any information lost can be retrieved from the remaining coding challenge.

During this stage, we compute each candidate keyword's embedding similarity with every non-potential keyword token. The keyword is marked as valid for removal if at least one "close" word is identified. Again, we consider "close" keywords with a similarity score larger than 0.7. If a keyword exists in multiple locations, the first instance is not marked as valid for removal, while the rest are. When a keyword happens to be an argument type (i.e., list, integer, tuple), we additionally look for instances of that type in the examples or name block. In case of a match, the keyword is safe for removal. Equivalent information already exists in the context. As the final step, we chose between the following transformations:

Drop one. Removes one of the provided keywords from the *Description Block*. The transformation is repeated N times where N is the number of identified keywords.

Drop all. Removes all the provided keywords simultaneously from the *Description Block*

Drop examples. Removes all the provided examples from the *Example Block*.

Anonymize. Replaces the function name with an arbitrary token. We use *"func"* in our experiments. Note that the function name is also replaced in the provided examples, so no information leak occurs. We also tested whether the choice of *"func"* may potentially bear some intrinsic adversarial effect associated with the training data. We experimented with other word choice replacements (*"action"*,*"do_stuff"*, *"XYZ"*) and got the same results. Furthermore, we identified instances where the function name, although closely correlated to the task at hand, if it was to be taken as the sole source of information, could instead be misleading, signifying the need for proper context understanding by the tested models (See Appendix 10.8).

For example, let us use our framework on the challenge presented in Figure 1. At the first stage, KeyBert would have identified the following keywords: [Reverse, list, return, second]. Among these, the word *second* does not pass the first filtering stage with over 0.7 similarity score against our set. In the second stage, each word would be compared against all the existing tokens. *Reverse* and *return* will not be associated with other tokens. *List* will be identified in the function name and input argument type. Also, since *list* is also a python keyword, it will be matched against the list type of the input given in the examples. This leaves *list* as the only available keyword for removal. If keyword drop would be combined with anonymization, the drop would still be valid since the information would still be available in the examples and input type.

These transformations test the hypotheses we associate with each block, as presented in Section 5.1. Removing possible hints leads to performance drops between the original and modified challenges, revealing underlying biases in the models' logic Arguably, any of our suggested transformations can destroy local semantics. However, we take significant measures to ensure that global semantics is preserved and enough information exists towards its solution. This is also why we refrain from performing simultaneous transformations in the *Example Block* and *Description Block*, or all of the *Blocks of Influence* together; a model stripped of all necessary information cannot generate a proper solution. To quantify the possible degree of ambiguity our transformations introduce, we employ the LM critic test, inspired by the work of (Yasunaga et al., 2021; Yasunaga and Liang, 2021): We collect a random sample of 200 coding challenges from the HumanEval and MBPP. Each challenge is then transformed according to the methods presented in Table 2. Afterwards, for both the original and every modified version of a challenge, we calculate their log probability score using a large language model. The core idea is that the model will act as a soft critic, ranking model inputs by their overall plausibility. Modified inputs that seem "off" to the critic and are partially understood will be assigned a log probability score far lower than the unmodified ones. Since this criterion is based on local neighborhood optimality, only moderate changes are allowed between the challenges under comparison. For example, two completely different but syntactically and semantically correct text snippets can have similar log probability scores. During their comparison, however, we would have violated the locality assumption, and no conclusions could be drawn about their contents. As our critic, we employ the Codex-v2 model (Chen et al., 2021). We calculate log probability similarity as: $Sim = 100 - \frac{LogP_{Method} - LogP_{Original}}{LogP_{Original}}$.
Table 2 shows that our transformations do not introduce drastic changes to the coding challenge.

Method	Similarity (%)	
	w/ CAF	w/o CAF
Original	100.0 (± 0.0)	100.0 (± 0.0)
Anonymization	98.5 (± 1.2)	98.5 (± 1.2)
Drop One	97.3 (± 1.5)	84.2 (± 2.2)
Drop All	95.3 (± 1.9)	80.3 (± 2.8)
Anonymization + Drop One	95.8 (± 1.4)	80.9 (± 2.3)
Anonymization + Drop All	94.6 (± 2.3)	78.4 (± 3.1)

Table 2: Similarity scores for different methods of our framework with (w/) and without (w/o) the proposed context-aware filtering mechanism (CAF). Results of 200 samples are presented.

Even in the most aggressive transformation of *Anonymization + Drop All*, the critic assigns over 94% similarity between code challenges affected by it versus their original form. For comparison, removing the context-aware filtering stage, leads to only 78% similarity in the case of *Anonymization + Drop All* transformation. We believe this is a fair indicator that the tested models observe inputs of similar quality and comprehensibility during our experiments. Note that we omit results for the *Drop Examples* method. In this case, the log probabilities will significantly change since we remove many tokens, which violates the method's locality prerequisite.

6 Experiments

6.1 Results on Block Transformations

The main results of our experiments are presented in Table 3. Despite their simplicity, our transformations cause consistent drops in performance across different model sizes on both datasets.[1] Mere anonymization causes drops of 19% on average in both Pass@1 and Pass@100 metrics, validating our claims of memorization effects. Single (*Drop One*) and full keyword removal (*Drop All*) reduce models' performance by 15% and 22% on average, suggesting their inability to deduct the missing context from *Name Block* and *Example Block*. Instead, models rely on generating arbitrary, commonly used snippets that vaguely fit for the task. Especially interesting are the cases of *Drop Examples* and *Anonymize + Drop Examples*, with 15% and 25% average drops. Both transformations remove the information provided by the docstring examples, with the latter having the additional restriction of an anonymized function. With the *Description Block* unmodified in both cases, these transformations target the models' abilities to create solutions based on their natural language understanding. The combination of anonymization with the drop of all keywords (*Anonymize + Drop All*) seems to be the most challenging transformation overall, with drops of approximately 40%. Its primary purpose is to assess the model's capability of deducting the missing context of the *Description Block* by only observing patterns in the examples. These observations suggest a clear model preference over its sources of information, with the task description being the primary one. Thus, when a model exhausts its ability to understand the task, it exploits similarities of the function name with previously seen code solutions. Simultaneously, the model's reasoning relies on the example demonstrations, which, as seen from (*Anonymize + Drop All*), are not always able to provide clear directives.

6.2 Towards Bias Mitigation

Inspired by the field of adversarial training, we decided to investigate the effects of using our framework transformations as training augmentations. To this end, we apply our framework to examples of the MBPP challenge and use them as a fine-tuning dataset for three different Codeparrot models. We use HumanEval as our test dataset, which bears no overlap with the MBPP. In this way, our models have not seen examples of the test set during their training or fine-tuning steps. In Table 4, we compare the results of our models before and after fine-tuning. Models benefit from the introduction of augmented examples and partially recover from failure modes caused by the need to rely on hints. The larger the model, the more its abilities benefit. We believe this effect is closely related to large language models' scaling reasoning capabilities and their parameter size. The need to rely on hints can be attributed to low data quality or lack of task-specific inductive biases. However, the capacity to properly understand coding tasks is undoubtedly there. To improve the code generation abilities of models, we thus suggest exposing them to challenges that push their deductive and reasoning abilities. We decided to repeat the experiments, but without including any of our data augmentation techniques during fine-tuning. We observe that under this setup, models do not exhibit any significant improvement against our method's perturbations. Our suggested data augmentations that push the reasoning limits of the models are

[1] We present a full table of results, including Codeparrot (110M), CodeGen(350M), Bloom(1.7B) and Codex(v1) in Appendix 10.5

Table 3: Model results on Human Eval and MBPP.

	Codeparrot (1.5B)				Incoder (1.6B)				CodeGen-Mono (6B)			
	Human Eval		MBPP		Human Eval		MBPP		Human Eval		MBPP	
Method	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)
Original	4.1	17.8	6.1	31.2	11.3	24.2	14.6	56.7	26.1	65.8	42.3	77.3
Drop One	3.9	13.2	4.2	26.8	10.5	22.3	11.5	45.4	18.4	39.3	25.2	65.7
Drop All	3.6	11.1	3.9	21.7	9.7	17.6	12.8	42.1	13.9	34.8	22.4	57.7
Drop Ex	3.7	14.3	5.3	27.5	11.3	22.2	14.4	43.8	20.4	42.3	27.2	61.7
Anon	3.8	12.5	4.7	23.2	9.1	21.8	11.3	45.2	18.2	37.3	24.0	65.6
Anon+Drop One	3.3	9.5	3.9	20.2	7.4	21.5	10.5	44.9	12.6	24.6	15.8	58.6
Anon+Drop All	2.1	8.9	3.9	17.9	6.3	17.5	8.0	41.3	11.5	23.1	14.9	46.3
Anon+Drop Ex	3.7	11.8	4.6	22.8	8.7	21.3	11.2	43.5	16.0	28.3	18.2	60.7
	Incoder (6B)				Codex (v2)				Bloom (176B)			
	Human Eval		MBPP		Human Eval		MBPP		Human Eval		MBPP	
Method	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)
Original	15.2	47.0	19.4	65.1	49.4	91.4	60.1	86.3	16.4	57.2	20.8	62.4
Drop One	12.1	35.3	18.9	52.6	36.0	86.2	56.0	79.2	12.8	48.6	15.8	51.4
Drop All	10.2	28.2	15.6	47.0	37.1	73.7	52.1	69.5	11.5	40.2	14.2	44.4
Drop Ex	12.7	29.5	17.4	50.3	41.4	81.0	48.8	70.7	15.2	43.3	15.8	50.1
Anon	11.6	32.9	14.8	50.7	44.5	90.4	57.9	81.7	14.0	48.3	15.1	51.2
Anon+Drop One	8.1	30.6	13.5	46.7	29.8	74.4	51.2	69.5	12.8	41.9	13.6	46.8
Anon+Drop All	7.5	25.2	11.2	38.9	24.2	68.7	47.2	63.8	10.3	36.8	12.6	38.4
Anon+Drop Ex	11.2	28.1	14.5	50.2	34.1	72.5	42.6	70.5	14.0	39.8	14.3	47.8

Table 4: HumanEval results of fine-tuning Codeparrot on the MBPP dataset with (A) or with no (NA) augmentations: Regular finetuning does not contribute to bias removal, achieving similar results against the perturbations. However, our suggested augmentations lead to higher model performance, especially in the pass@100 metric. The average of 15 runs is presented. Bold marks statistically significant improvements under the T-Test (Before versus After-A) with $a = 0.95$.

	Codeparrot - 110M				Codeparrot - 350M				Codeparrot - 1.5B			
	Pass@1 (T=0.2)		Pass@100 (T=0.8)		Pass@1 (T=0.2)		Pass@100 (T=0.8)		Pass@1 (T=0.2)		Pass@100 (T=0.8)	
Method	Before	After NA / A	Before	After NA / A	Before	After NA / A	Before	After NA / A	Before	After NA / A	Before	After NA / A
Original	3.8	3.7 / 3.7	12.7	12.1 / 12.1	3.8	3.7 / 3.7	13.9	13.7 / 13.7	4.1	4.1 / 4.1	17.8	17.8 / 17.8
Drop One	3.3	3.2 / **3.6**	9.7	9.7 / **10.4**	3.3	3.3 / **3.6**	11.9	11.9 / **12.3**	3.9	3.9 / **4.0**	13.2	13.2 / **14.1**
Drop All	3.1	3.1 / 3.1	7.2	7.2 / **7.9**	3.2	3.2 / 3.2	10.1	10.0 / **10.7**	3.6	3.6 / 3.7	11.1	11.1 / **12.3**
Drop Ex	3.8	3.7 / 3.7	9.9	9.9 / **10.2**	3.8	3.8 / 3.7	12.9	12.9 / 12.9	3.7	3.7 / 3.7	14.3	14.3 / **15.1**
Anon	3.4	3.4 / **3.5**	8.7	8.7 / **9.1**	3.6	3.6 / 3.6	11.6	11.6 / **12.2**	3.8	3.8 / 3.9	12.5	12.5 / **13.8**
Anon+Drop One	3.0	2.8 / **3.4**	7.5	7.5 / **7.9**	3.0	2.8 / **3.5**	8.2	8.2 / **9.4**	3.3	3.3 / **3.5**	9.5	9.5 / **10.5**
Anon+Drop All	1.9	1.9 / 2.0	6.9	6.9 / 6.9	2.0	2.0 / **2.2**	8.1	8.0 / 8.3	2.1	2.1 / **2.4**	8.9	8.8 / 9.4
Anon+Drop Ex	3.4	3.3 / 3.4	8.7	8.7 / **9.0**	3.6	3.6 / 3.6	10.7	10.7 / **11.8**	3.7	3.7 / 3.7	11.8	11.8 / **13.7**

thus a valid alternative to simple fine-tuning.

6.3 Effects of Longer Context

When causally training on coding datasets, models condition on multiple functions and declarations in the same file. The input is a conglomerate of rapidly changing contexts, with each function or class being a self-contained entity. Subsequently, a model is accustomed to localizing its focus when trained on such data. As an extension to our previous experiment, we measure the effects of using a long description dataset, DMCC, as a fine-tuning target. By training on long descriptions of natural language, we promote the context-deducting skills of the model under test. A model able to widen its focus can avoid distractions caused by missing keywords. Efficient context understanding will replace not rely heavily on internal biases. We choose Bloom as the model under test since it was not explicitly tuned for code generation but rather general language understanding. In Table 5, we present results of fine-tuning on MBPP, modified by our framework. We observe similar performance improvements as in Table 4. We ex-

Table 5: HumanEval results of fine-tuning Bloom (560M) on the modified MBPP and long-description DMCC dataset with (A) or without (NA) augmentations: Model exhibits increased performance under the combined augmentation schema against perturbations that challenge language understanding. The average of 15 runs is presented. Bold marks statistically significant improvements under the T-Test (Before versus +DMCC-A) with $a = 0.95$.

	Pass@1 (T=0.2)			Pass@100 (T=0.8)		
Method	Before	+MBPP NA / A	+DMCC NA / A	Before	+MBPP NA / A	+DMCC NA / A
Original	3.7	3.6 / 3.6	3.6 / 3.6	12.1	12.1 / 12.1	12.0 / 12.0
Drop One	3.1	3.1 / 3.6	3.1 / **3.6**	10.3	10.3 / 10.9	10.3 / **10.9**
Drop All	2.4	2.3 / 2.4	2.3 / **2.9**	9.2	9.1 / 9.1	9.1 / **9.7**
Drop Ex	3.0	3.0 / 3.0	3.0 / 3.0	11.0	11.0 / 11.3	11.0 / **11.5**
Anon	2.5	2.5 / 3.0	2.6 / **3.6**	10.7	10.7 / 10.9	10.8 / **11.3**
Anon+Drop One	1.9	1.9 / 2.3	1.9 / **2.4**	7.8	7.8 / 9.1	7.8 / **9.7**
Anon+Drop All	1.8	1.8 / 1.8	1.8 / **2.3**	7.0	7.0 / 7.2	7.0 / **8.3**
Anon+Drop Ex	2.4	2.4 / 2.9	2.4 / **3.0**	9.7	9.7 / 10.3	9.7 / **11.4**

periment again, this time combining both MBPP and DMCC examples. We show that incorporating examples of more extended context leads to even better performance against transformations targeting the *Description Block* and language understanding. Similar experiments were conducted with the CodeParrot variants but were unfruitful. We attribute this to the restricted focus regarding training data (exclusively Python3 code) and architectural differences between the models. We believe that the merging benefits of our two proposed setups can serve as an interesting direction towards model resilience in code generation scenarios.

7 Conclusions

We present a simple approach to isolate cues and benchmark the reasoning of code generation models through input-level transformations. Our method treats code examples as a combination of three blocks, each providing different cues to the model. We show that minor transformations can lead models to failure, signifying the existence of biases. Our framework can automatically identify and remove keywords responsible for indirect hinting. We show that popular models with solid results on challenging coding challenges are susceptible to our tests, with their performance degrading noticeably. Moreover, we studied the effects of utilizing our proposed transformations during the fine-tuning of a model. Models can benefit from our proposed changes, with the effect proportional to their parameter size. We believe that, despite their success, code generation systems with LLMs as backbones inherit some of their biases and modes of failure. Training on structured and well-documented code, combined with our proposed techniques, is a promising direction towards reliable code generation. Although an ideal fit for competition-style challenges, our method can be extended to support less formatted high-quality codebases (e.g. GitHub repositories). For a short analysis see Section 10.1 of the Appendix.

```python
from typing import List
def has_close_elements(numbers: List, threshold: float):
    "Check if in the given list of numbers, are any two numbers closer
    ↪  to each other than given threshold."
    Examples:
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0,2.0,3.0,4.0,5.0,2.0], 0.3)
    True
    return any(abs(x-y) < threshold for x,y in zip(numbers,
    ↪  numbers[1:]))
```

```python
from typing import List
def has_close_elements(numbers: List, threshold: float):
    "Check if in the given list of numbers, are any two numbers closer
    ↪  to each other than given threshold."
    for i in range(len(numbers) -1):
        if abs(numbers[i] - numbers[i+1]) < threshold:
            return True
        return False
```

Figure 2—Example removal reveals poor reasoning (*Example drop* / Codex-v1): The model initially exhibits signs of task comprehension (top), generating a correct solution. Removing the examples, however, reveals a lack of proper reasoning; Although the model still understands that it has to compare numbers, it resorts to a naive sequential check instead of comparing each available pair (bottom).

```python
from typing import List
def string_xor(a:str, b:str) -> str:
    "Input is two strings a and b consisting only of 1s and 0s. Perform
    ↪  binary XOR on these inputs and return the result as a string.
    Examples:
    >>> string_xor('010','110')
    '100'
    return ''.join([str(int(a,2) ^ int(b,2)) for a,b in zip(a,b)])
```

```python
from typing import List
def string_xor(a:str, b:str) -> str:
    "Input is a and b consisting only of 1s and 0s. Perform  XOR on
    ↪  these inputs and return the result.
    Examples:
    >>> string_xor('010','110')
    '100'
    return ''.join([str(int(a) ^ int(b))])
```

Figure 3—Keyword hinting (*Drop All* / Bloom 175B): After the removal of keywords, the context remains intact: The *two strings* keyword can be assumed by observing the function arguments, and the *binary/string* keywords by the examples and return type signature of the function. Nevertheless, the model fails to generate a correct solution (bottom).

```python
def pairs_sum_to_zero(l):
    "Pairs_sum_to_zero takes a list of integers as input. It returns
    ↪  True if there are two distinct elements in the list that sum to
    ↪  zero, and False otherwise."
    Examples:
    >>> pairs_sum_to_zero([1,3,5,8])
    False
    >>> pairs_sum_to_zero([1,3,-2,1])
    False
    >>> pairs_sum_to_zero([2,4,-5,3,5,7])
    True
    for i in range(len(l)):
        for j in range(i+1, len(l)):
            if l[i] + l[j] == 0:
                return True
    return False
```

```python
def func(l):
    "Func takes a list of integers as input. It returns True if there
    ↪  are two distinct elements in the list that sum to zero, and
    ↪  False otherwise."
    return any(sum(x) == 0 for x in l)
```

Figure 4—*Anonymize + Drop Examples* / Incoder 6B: Using only the problem description, the model creates partially informed subparts (`any` derives from *"if there are"*, `sum(x) == 0` from *"sum to zero"*, and `for x in l` from *"elements in the list"*) that are not combined correctly to solve the task (bottom), signifying that hints from the function name / examples were used in the correct solution (top).

8 Limitations

Some limitations and possible research directions exist in our work. Our study focuses on the Python3 programming language, with many coding challenges existing in different popular choices (e.g., C, C++, Java, Scala). Although the Blocks of Influence identification mechanism could be easily

adapted to each case, an off-the-shelf application of our framework in another language would lead to errors.

Similarly, the framework assumes that each coding challenge will be in a "competition-style" format, meaning that a proper problem description, in-docstring examples, and each input types are present for each example. In Appendix Section 10.1, we present how an adaptation to less formatted codebases would be possible, but for now, we leave it as a future investigation.

Finally, there is no guarantee that the improved performance against the suggested perturbations reflects an equivalent performance increase in real-world code assistant applications. Real-time coding suggestions and completions that are more user aligned are out of the scope of this work.

9 Risks and Ethical Considerations

Our research aims to discover and remove biases in code-generation scenarios through adversarial intervention. However, we acknowledge that insecure or malicious code can still be generated after finetuning with our suggested augmentations. Furthermore, our work is focused only on cognitive biases that affect the reasoning and logic behind the coding process of large language models. Social biases and stereotypes can still appear when general-purpose LLMs such as Codex or Bloom are used in typical text generation scenarios. Signs of robustness against our methods are not to be confused with indicators of other forms of biases not existent.

Acknowledgements

All experiments were performed using the Entropy cluster funded by NVIDIA, Intel, the Polish National Science Center grant UMO-2017/26/E/ST6/00622 and ERC Starting Grant TOTAL. The work of Spyridon Mouselinos and Henryk Michalewski was supported by the Polish National Science Center grant UMO-2018/29/B/ST6/02959.

References

Aida Amini, Saadia Gabriel, Shanchuan Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. 2019. Mathqa: Towards interpretable math word problem solving with operation-based formalisms. *CoRR*, abs/1905.13319.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

E. Caballero. 2016. Description2code dataset, 8 2016.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

François Chollet. 2019. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022a. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022b. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Max Glockner, Vered Shwartz, and Yoav Goldberg. 2018. Breaking NLI systems with sentences that require simple lexical inferences. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 650–655, Melbourne, Australia. Association for Computational Linguistics.

Maarten Grootendorst. 2020. Keybert: Minimal keyword extraction with bert.

Sylvain Gugger, Lysandre Debut, Thomas Wolf, Philipp Schmid, Zachary Mueller, and Sourab Mangrulkar. 2022. Accelerate: Training and inference at scale made simple, efficient and adaptable. https://github.com/huggingface/accelerate.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021a. Measuring coding challenge competence with apps. *NeurIPS*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021b. Measuring mathematical problem solving with the math dataset. *NeurIPS*.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Shane Legg. 2008. *Machine super intelligence*. Ph.D. thesis, Università della Svizzera italiana.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. 2022a. Solving quantitative reasoning problems with language models.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. 2022b. Solving quantitative reasoning problems with language models. *arXiv preprint arXiv:2206.14858*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode.

Paul Pu Liang, Chiyu Wu, Louis-Philippe Morency, and Ruslan Salakhutdinov. 2021. Towards understanding and mitigating social biases in language models. In *International Conference on Machine Learning*, pages 6565–6576. PMLR.

Bill Yuchen Lin, Seyeon Lee, Rahul Khanna, and Xiang Ren. 2020. Birds have four legs?! NumerSense: Probing Numerical Commonsense Knowledge of Pre-Trained Language Models. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6862–6868, Online. Association for Computational Linguistics.

Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. 2017. Program induction by rationale generation: Learning to solve and explain algebraic word problems. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 158–167, Vancouver, Canada. Association for Computational Linguistics.

Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2022. What makes good in-context examples for GPT-3? In *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, pages 100–114, Dublin, Ireland and Online. Association for Computational Linguistics.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664.

M. Mirzayanov. 2020. Codeforces: Results of 2020.

Margaret Mitchell, Giada Pistilli, Yacine Jernite, Ezinwanne Ozoani, Marissa Gerchick, Nazneen Rajani, Sasha Luccioni, Irene Solaiman, Maraim Masoud, Somaieh Nikpoor, Carlos Muñoz Ferrandis, Stas Bekman, Christopher Akiki, Danish Contractor, David Lansky, Angelina McMillan-Major, Tristan Thrush, Suzana Ilić, Gérard Dupont, Shayne Longpre, Manan Dey, Stella Biderman, Douwe Kiela, Emi Baylora, Teven Le Scao, Aaron Gokaslan, Julien Launay, and Niklas Muennighoff. 2022. The world's largest open multilingual language model: Bloom.

John X. Morris, Eli Lifland, Jin Yong Yoo, and Yanjun Qi. 2020. Textattack: A framework for adversarial attacks in natural language processing. *CoRR*, abs/2005.05909.

Spyridon Mouselinos, Henryk Michalewski, and Mateusz Malinowski. 2022. Measuring clevrness: Blackbox testing of visual reasoning models. *ICLR: International Conference on Learning Representations*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint*.

Piotr Piekos, Henryk Michalewski, and Mateusz Malinowski. 2021. Measuring and improving bert's mathematical abilities by predicting the order of reasoning. *ACL: Association for Computational Linguistics*.

Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, and Ulrich Finkler. 2021. Project codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. *CoRR*, abs/2105.12655.

Yasaman Razeghi, Robert L. Logan, Matt Gardner, and Sameer Singh. 2022. Impact of pretraining term frequencies on few-shot reasoning.

Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. Zero-offload: Democratizing billion-scale model training.

David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. 2019. Analysing mathematical reasoning abilities of neural models. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Eui Chul Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov. 2019. Program synthesis and semantic parsing with learned code idioms. *Advances in Neural Information Processing Systems*, 32.

Alon Talmor, Ori Yoran, Ronan Le Bras, Chandra Bhagavatula, Yoav Goldberg, Yejin Choi, and Jonathan Berant. 2022. Commonsenseqa 2.0: Exposing the limits of AI through gamification. *CoRR*, abs/2201.05320.

Lewis Tunstall, Leandro von Werra, and Thomas Wolf. 2022a. Natural language processing with transformers.

Lewis Tunstall, Leandro von Werra, and Thomas Wolf. 2022b. Natural language processing with transformers.

Eric Wallace, Shi Feng, Nikhil Kandpal, Matt Gardner, and Sameer Singh. 2019. Universal adversarial triggers for attacking and analyzing NLP. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2153–2162, Hong Kong, China. Association for Computational Linguistics.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Michihiro Yasunaga, Jure Leskovec, and Percy Liang. 2021. Lm-critic: Language models for unsupervised grammatical error correction. In *Empirical Methods in Natural Language Processing (EMNLP)*.

Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. In *ICML*, pages 11941–11952.

Honghua Zhang, Liunian Harold Li, Tao Meng, Kai-Wei Chang, and Guy Van den Broeck. 2022. On the paradox of learning to reason from data. *arXiv preprint arXiv:2205.11502*.

Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate before use: Improving few-shot performance of language models. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 12697–12706. PMLR.

10 Appendix

10.1 Extension to open-source code

Although an ideal fit for competition-style challenges, our method can be extended to support less formatted high-quality codebases (e.g. GitHub repositories). Large files can be broken down into individual functions/classes, each further analyzed into Blocks of Influence. In such codebases, function names should be closely relevant to their purpose. The existence of meaningful docstrings is crucial, the absence of which promotes more memorization and biases as we exhibited. Moreover, the input/output checks contained in function unit tests can be repurposed as function examples. Keywords can be chosen similarly, with the context being co-informed by both local and larger scopes.

10.2 Information on Models and Datasets

Model Name	Link	LICENSE
KeyBert (Grootendorst, 2020)	https://github.com/MaartenGr/KeyBERT	MIT
Codeparrot (Tunstall et al., 2022b)	https://huggingface.co/codeparrot/codeparrot	Apache License 2.0
InCoder (Fried et al., 2022)	https://github.com/dpfried/incoder	CC-BY-NC 4.0
CodeGen (Nijkamp et al., 2022)	https://github.com/salesforce/CodeGen	BSD 3-Clause
Bloom (Mitchell et al., 2022)	https://huggingface.co/bigscience/bloom	BigScience RAIL License v1.0
Codex-V2 (Chen et al., 2021)	https://beta.openai.com/	N/A

Table 6: URL and Licenses of used Models.

Dataset Name	Link	LICENSE
CodeParrot Dataset (Tunstall et al., 2022a)	https://huggingface.co/datasets/codeparrot/codeparrot-clean	Apache License 2.0
HumanEval (Chen et al., 2021)	https://github.com/openai/human-eval	MIT
MBPP (Austin et al., 2021)	https://github.com/google-research/google-research/tree/master/mbpp	CC BY 4.0
DMCC (Li et al., 2022)	https://github.com/deepmind/code_contests	Apache License 2.0

Table 7: URL and Licenses of used Datasets.

Name	#Problems	#Tests per Problem	Avg. desc. length	Avg. keywords
HumanEval (Chen et al., 2021)	164	8	449	4
MBPP (Austin et al., 2021)	1000	3	235	4
DMCC (Train / Python3) (Li et al., 2022)	8139	85	1480	9

Table 8: Datasets used in experiments. We present the number of problems, number of tests per problem, average length of the challenge description and average distinct keywords identified by our framework.

For all of our perturbation experiments, we utilize the abovementioned models, and we comply with their respective licenses and intended use (generating code completions in python3). This also stands true for Codeparrot and Bloom, for which we create fine-tuned versions. Furthermore, we do not plan to repack or redistribute any of the used datasets. We plan to release the codebase of this work as an open-source project.

10.3 Information on Experimental Setup

Our experimental setup consisted of 4x NVIDIA V100 GPUs. Regarding the results of Table 3, the computing time of each table entry was influenced by: the model size, the k value of pass@k metric (number of generations), the perturbation method, and the dataset tested. Specifically for the drop one / anonymize + drop one methods, the experiment was repeated N times, where N corresponds to the number of keywords identified. This results in approximately four times slower experiments for those perturbations since in both HumanEval and MBPP, four keywords on average per problem were identified (see Table 8). API calls to Codex and Bloom models were subject to throttling limits, and waiting loops

were introduced to avoid interruptions of service. The total experiment time resulted in approximately 500 hours.

Regarding the finetuning experiments of Table 4, we trained Codeparrot Models with the AdamW optimizer at a learning rate of 1e-5, batch size of 64, weight decay of 0.01, and constant learning rate schedule. The same hyperparameters were chosen as well in the case of the MBPP-only experiment of the Bloom Model in Table 5. When both MBPP and DMCC datasets were combined, a learning rate of 3e-5 and a batch size of 256 were used. The hyperparameters were chosen after a grid search on the following choices: Weight decay (0.01 / 0.0), Learning Rate: (1e-6,1e-5,3e-5,5e-5,1e-4), Schedule: (Constant, Cosine). The batch size was chosen proportionally to the overall dataset length. All models were trained with the Accelerate library (Gugger et al., 2022) and Zero-3 (Ren et al., 2021) partitioning schema. Regarding the training objective, we used a custom causal language modeling loss. The loss was calculated only on the generated tokens corresponding to the problem solution and not on any tokens belonging to the problem description or examples. We used a random validation split of 10% and validation loss for all experiments as our metric for early stopping.

10.4 Qualitative Examples

We present examples of code generation failures caused by our framework across different models and scenarios. On each pair, the left image represents the original, unmodified challenge alongside the correctly generated solution. The right image contains the modified version of the challenge and the incorrect completion.

```python
def is_equal_to_sum_even(n):
    "Evaluate if n can be written as the sum of exactly 4 positive even
    ↪  numbers."
    Examples:
    >>> is_equal_to_sum_even(4)
    False
    >>> is_equal_to_sum_even(8)
    True
    if n % 2 != 0:
        return False
    for i in range(1, n // 2 + 1):
        for j in range(1, n // 2 + 1):
            for k in range(1, n // 2 + 1):
                for l in range(1, n // 2 + 1):
                    if i + j + k + l == n:
                        return True
    return False
```

```python
def func(n):
    "Evaluate if n can be written as the sum of exactly 4 positive even
    ↪  numbers."
    Examples:
    >>> func(4)
    False
    >>> func(8)
    True
    if n % 2 != 0:
        return False
    if n < 8:
        reuturn False
    if n == 8:
        return True
    if n > 8:
        return func(n - 2)
```

Figure 5—Instance of anonymization on Codex-V1

```python
def remove_vowels(text):
    "Remove_vowels is a function that takes a string and
    ↪  returns the string without vowels"
    >>> remove_vowels('')
    ''
    >>> remove_vowels('abcdef')
    'bcdf'
    >>> remove_vowels('aaaaa')
    ''
    >>> remove_vowels('aaBAAA')
    'B'
    vowels = 'aeiouAEIOU'
    return ''.join([c for c in text if c not in vowels])
```

```python
def remove_vowels(text):
    "Remove_vowels is a function that takes a string and
    ↪  returns the string without vowels"
    vowels = ['a','e','i','o','u']
    for vowel in vowels:
        text = text.replace(vowel, '')
    return text
```

Figure 6—Instance of dropping the prompt examples on Codex-V2

```
def do_algebra(operator, operand):
    "Given two lists operator, and operand. The first List has
    ↪  basic algebra operations, and the second list is a
    ↪  List of integers. Use the two given lists to build the
    ↪  algebric expression and return the evaluation of this
    ↪  expression.
    The basic algebra operations:
    Addition ( + )
    Subtraction ( - )
    Multiplication ( * )
    Floor division ( // )
    Exponentiation ( ** )

    Example:
    operator['+', 'x', '-']
    array = [2, 3, 4, 5]
    result = 2 + 3 * 4 - 5
    => result = 9

    result = operand[0]
    for i in range(len(operator)):
        if operator[i] == '+':
            result += operand[i+1]
        elif operator[i] == '-':
            result -= operand[i+1]
        elif operator[i] == '*':
            result *= operand[i+1]
        elif operator[i] == '//':
            result //= operand[i+1]
        elif operator[i] == '**':
            result **= operand[i+1]
    return result
```

```
def do_algebra(operator, operand):
    "Given two lists operator, and operand. The first List has
    ↪  basic algebra operations, and the second list is a
    ↪  List of integers. Use the two given lists to build the
    ↪  algebric expression and return the evaluation of this
    ↪  expression.
    The basic algebra operations:
    Addition ( + )
    Subtraction ( - )
    Multiplication ( * )
    Floor division ( // )
    Exponentiation ( ** )

    return eval(str(operand[0]) + operator[0] +
    ↪  str(operand[1]) + operator[1] +  str(operand[2]) +
    ↪  operator[2] + str(operand[3]) + operator[3] +
    ↪  str(operand[4]) + operator[4] + str(operand[5]))
```

Figure 7—Instance of dropping the prompt examples on CodeParrot-1.7B

```
def square_nums(nums):
    "Find squares of individual elements in a list using the
    ↪  lambda function."
    Examples:
    >>> square_nums([1, 2, 3, 4, 5, 6, 7, 8, 9, 16])
    [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
    >>> square_nums([10,20,30])
    [100,400,900]
    return list(map(lambda x: x**2, nums))
```

```
def func(nums):
    "Find squares of individual elements using the lambda
    ↪  function."
    Examples:
    >>> func([1, 2, 3, 4, 5, 6, 7, 8, 9, 16])
    [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
    >>> func([10,20,30])
    [100,400,900]
    s=0
    for i in range(len(nums)):
        if nums[i] == 0:
            s+=1
        elif s % 2 == 0:
            s+=1
        else:
            s+=1
    return s
```

Figure 8—Instance of anonymization and keyword drop on Bloom-1.7B

```
def iscube(a):
    "Write a function that takes an integer a and returns True
    ↪  if this integer is a cube of some integer number."
    Examples:
    iscube(1)   ==> True
    iscube(2)   ==> False
    iscube(-1)  ==> True
    iscube(64)  ==> True
    iscube(0)   ==> True
    iscube(188) ==> False

    return a == 0 or a == 1 or (a > 0 and a**(1/3) ==
    ↪  int(a**(1/3)))
```

```
def iscube(a):
    "Write a function that takes a and returns True if a cube
    ↪  of some integer number."
    Examples:
    iscube(1)   ==> True
    iscube(2)   ==> False
    iscube(-1)  ==> True
    iscube(64)  ==> True
    iscube(0)   ==> True
    iscube(188) ==> False

    if (a ** 3 == a):
        return True
    else:
        return False
```

Figure 9—Instance of keyword drop on Bloom-175B

```
def solve(s):
    "You are given a string s. If s[i] is a letter, reverse
    ↪  its case from Lower to upper or vise versa, otherwise
    ↪  keep it as it is. If the string contains no letters,
    ↪  reverse the string. The function should return the
    ↪  resulted string."
    Examples:
    >>> solve("1234") = "4321"
    >>> solve("ab") = "AB"
    >>> solve("#a@C") = "#A@c"
    if not s:
        return s
    if not any(c.isalpha() for c in s):
        return s[::-1]
    res = ''
    for c in s:
        if c.isalpha():
            res += c.swapcase()
        else:
            res += c
    return res
```

```
def solve(s):
    "You are given a string s. If s[i] is a letter, case from
    ↪  lower to upper or vise versa, otherwise keep it as it
    ↪  is. If the string contains no letters, reverse the
    ↪  string. The function should return the resulted
    ↪  string."
    Examples:
    >>> solve("1234") = "4321"
    >>> solve("ab") = "AB"
    >>> solve("#a@C") = "#A@c"
    if not s.isalpha():
        return s[::-1]
    else:
        return s.swapcase()
```

Figure 10—Instance of keyword drop on Incoder-1.6B

10.5 Quantative Results

We present our full results table, including the CodeParrot(110M) and Codex(v1) results. Note here that experiments involving the large version of the Bloom Model were done once in the case of pass@100 metric due to restrictions with the API request limits.

Model	Method of Attack	Human Eval		MBPP	
		Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)
Codeparrot (110M) (Tunstall et al., 2022b)	Original	3.8	12.7	5.1	26.2
	Drop One	3.3 (±0.1)	9.7 (±0.3)	4.1 (±0.1)	16.3 (±0.5)
	Drop All	3.1 (±0.1)	7.2 (±0.5)	3.9 (±0.1)	15.7 (±0.7)
	Drop Ex	3.8 (±0.0)	9.9 (±0.2)	5.0 (±0.0)	18.4 (±0.3)
	Anon	3.4 (±0.1)	8.7 (±0.2)	4.4 (±0.1)	16.1 (±0.5)
	Anon+Drop One	3.0 (±0.1)	7.5 (±0.5)	4.0 (±0.1)	13.6 (±1.1)
	Anon+Drop All	1.9 (±0.2)	6.9 (±0.5)	3.9 (±0.2)	12.0 (±1.5)
	Anon+Drop Ex	3.4 (±0.1)	8.7 (±0.3)	4.3 (±0.2)	16.1 (±0.8)
CodeGen-Mono (350M) (Nijkamp et al., 2022)	Original	12.7	35.2	19.2	59.4
	Drop One	7.1(±0.1)	31.1(±0.4)	10.7 (±0.1)	42.9 (±0.7)
	Drop All	5.5(±0.1)	24.5(±0.6)	9.4 (±0.2)	38.5 (±0.7)
	Drop Ex	8.3(±0.1)	29.8(±0.4)	11.7 (±0.1)	43.6 (±0.7)
	Anon	6.1(±0.2)	29.8(±0.5)	12.6 (±0.1)	42.6 (±0.8)
	Anon+Drop One	4.8(±0.2)	28.4(±0.6)	7.6 (±0.2)	39.4 (±1.3)
	Anon+Drop All	3.4(±0.2)	22.1(±0.7)	6.8 (±0.3)	35.8 (±1.6)
	Anon+Drop Ex	5.2(±0.2)	29.2(±0.6)	7.3 (±0.2)	40.5 (±1.1)
Codeparrot (1.5B) (Tunstall et al., 2022b)	Original	4.1	17.8	6.1	31.2
	Drop One	3.9 (±0.1)	13.2 (±0.4)	4.2 (±0.2)	26.8 (±0.8)
	Drop All	3.6 (±0.3)	11.1 (±0.6)	3.9 (±0.2)	21.7 (±1.1)
	Drop Ex	3.7 (±0.0)	14.3 (±0.2)	5.3 (±0.0)	27.5 (±0.7)
	Anon	3.8 (±0.1)	12.5 (±0.2)	4.7 (±0.2)	23.2 (±0.9)
	Anon+Drop One	3.3 (±0.2)	9.5 (±0.7)	3.9 (±0.1)	20.2 (±1.5)
	Anon+Drop All	2.1 (±0.3)	8.9 (±1.1)	3.9 (±0.2)	17.9 (±1.8)
	Anon+Drop Ex	3.7 (±0.2)	11.8 (±0.9)	4.6 (±0.1)	22.8 (±0.9)
Bloom (1.7B) (Tunstall et al., 2022b)	Original	4.3	14.6	6.6	37.2
	Drop One	3.0 (±0.2)	12.2 (±0.6)	2.7 (±0.3)	27.6 (±1.2)
	Drop All	2.4 (±0.3)	9.8 (±0.9)	2.6 (±0.3)	24.2 (±1.8)
	Drop Ex	3.6 (±0.1)	12.8 (±0.5)	3.1 (±0.2)	29.0 (±0.9)
	Anon	3.6 (±0.1)	11.6 (±0.5)	3.1 (±0.1)	27.5 (±1.1)
	Anon+Drop One	2.4 (±0.3)	9.1 (±1.1)	2.4 (±0.5)	25.3 (±1.8)
	Anon+Drop All	1.8 (±0.5)	8.5(±1.3)	2.0 (±0.6)	23.1 (±2.3)
	Anon+Drop Ex	3.4 (±0.2)	11.6 (±0.6)	3.0 (±0.3)	26.7 (±1.3)
Incoder (1.6B) (Fried et al., 2022)	Original	11.3	24.2	14.6	56.7
	Drop One	10.5 (±0.1)	22.3 (±0.9)	11.5(±0.4)	45.4 (±1.1)
	Drop All	9.7 (±0.3)	17.6 (±1.2)	12.8 (±0.6)	42.1 (±1.9)
	Drop Ex	11.3 (±0.2)	22.2 (±1.5)	14.4(±0.3)	43.8 (±0.7)
	Anon	9.1 (±0.1)	21.8 (±0.8)	11.3 (±0.5)	45.2 (±0.8)
	Anon+Drop One	7.4 (±0.7)	21.5 (±1.8)	10.5 (±0.6)	44.9 (±2.4)
	Anon+Drop All	6.3 (±0.9)	17.5 (±2.2)	8.0 (±0.8)	41.3(±2.5)
	Anon+Drop Ex	8.7 (±0.5)	21.3 (±1.6)	11.2 (±0.5)	43.5(±1.0)

Table 9: First part of results on Human Eval and MBPP datasets, for four tested models.

Model	Method of Attack	Human Eval		MBPP	
		Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)
Incoder (6B) (Fried et al., 2022)	Original	15.2	47.0	19.4	65.1
	Drop One	12.1 (±0.3)	35.3 (±1.2)	18.9 (±0.5)	52.6 (±1.1)
	Drop All	10.2 (±0.5)	28.2 (±1.4)	15.6 (±0.5)	47.0 (±1.9)
	Drop Ex	12.7 (±0.3)	29.5 (±0.9)	17.4 (±0.3)	50.3 (±0.7)
	Anon	11.6 (±0.2)	32.9 (±0.9)	14.8 (±0.6)	50.7 (±0.8)
	Anon+Drop One	8.1 (±0.7)	30.6 (±1.7)	13.5 (±0.7)	46.7 (±2.4)
	Anon+Drop All	7.5 (±1.3)	25.2 (±2.3)	11.2 (±1.1)	38.9 (±2.5)
	Anon+Drop Ex	11.2 (±0.4)	28.1 (±1.1)	14.5 (±0.5)	50.2 (±1.0)
CodeGen-Mono (6B) (Nijkamp et al., 2022)	Original	26.1	65.8	42.3	77.3
	Drop One	18.4 (±0.3)	39.3 (±0.9)	25.2 (±0.5)	65.7 (±1.2)
	Drop All	13.9 (±0.4)	34.8 (±1.3)	22.4 (±0.6)	57.7 (±1.6)
	Drop Ex	20.4 (±0.3)	42.3 (±1.1)	27.2 (±0.5)	61.7 (±1.1)
	Anon	18.2 (±0.3)	37.3 (±1.0)	24.0 (±0.5)	65.6 (±1.3)
	Anon+Drop One	12.6 (±0.5)	24.6 (±1.4)	15.8 (±0.7)	58.6 (±2.2)
	Anon+Drop All	11.5 (±0.8)	23.1 (±1.9)	14.9 (±0.8)	46.3 (±2.6)
	Anon+Drop Ex	16.0 (±0.5)	28.3 (±1.6)	18.2 (±0.7)	60.7 (±1.8)
Codex (v1) (Chen et al., 2021)	Original	39	82.9	51.7	83.4
	Drop One	29.2 (±0.2)	78 (±1.3)	48.3 (±0.4)	78.7 (±1.0)
	Drop All	30 (±0.4)	67.2 (±1.7)	33.9 (±0.8)	67.3 (±1.9)
	Drop Ex	32.9 (±0.1)	73.7 (±1.1)	42.1 (±0.2)	70.1 (±0.9)
	Anon	35.3 (±0.1)	81.7 (±1.2)	50.8 (±0.2)	81.5 (±1.2)
	Anon+Drop One	23.7 (±0.5)	67.0 (±2.3)	44.1 (±0.7)	67.7 (±2.6)
	Anon+Drop All	19.5 (±0.9)	62.1 (±2.7)	40.7 (±1.4)	61.4 (±3.1)
	Anon+Drop Ex	27.4 (±0.3)	65.2 (±1.6)	36.7 (±0.3)	67.7 (±1.5)
Codex (v2) (Chen et al., 2021)	Original	49.4	91.4	60.1	86.3
	Drop One	36.0 (±0.1)	86.2 (±0.8)	56.0 (±0.3)	79.2 (±1.1)
	Drop All	37.1 (±0.3)	73.7 (±1.3)	52.1 (±0.6)	69.5 (±1.8)
	Drop Ex	41.4 (±0.1)	81.0 (±1.1)	48.8 (±0.3)	70.7 (±0.9)
	Anon	44.5 (±0.2)	90.4 (±1.1)	57.9 (±0.3)	81.7 (±1.0)
	Anon+Drop One	29.8 (±0.7)	74.4 (±2.1)	51.2 (±1.1)	69.5 (±2.3)
	Anon+Drop All	24.2 (±0.8)	68.7 (±2.8)	47.2 (±1.3)	63.8 (±3.0)
	Anon+Drop Ex	34.1 (±0.3)	72.5 (±1.1)	42.6 (±0.4)	70.5 (±1.3)
Bloom (176B) (Tunstall et al., 2022b)	Original	16.4	57.2	20.8	62.4
	Drop One	12.8 (±0.3)	48.6	15.8 (±0.3)	51.4
	Drop All	11.5 (±0.6)	40.2	14.2 (±0.5)	44.4
	Drop Ex	15.2 (±0.2)	43.3	15.8 (±0.2)	50.1
	Anon	14.0 (±0.3)	48.3	15.1 (±0.1)	51.2
	Anon+Drop One	12.8 (±0.4)	41.9	13.6 (±0.7)	46.8
	Anon+Drop All	10.3 (±0.8)	36.8	12.6 (±1.1)	38.4
	Anon+Drop Ex	14.0 (±0.3)	39.8	14.3 (±0.3)	47.8

Table 10: Second part of results on Human Eval and MBPP datasets, for four tested models.

10.6 Few Interesting Examples

```
from typing import List

def func(numbers: List[float], threshold: float) -> bool:
    " Check if in given list of numbers,are any two numbers closer to each other than given threshold."
    >>> func({1.0, 2.0, 3.0], 10.5)
    False
    >>> func((1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    for i in range(len(numbers) - 1):
        if abs(numbers[i].id == id) {
            if (temp.id == id) {
                retum temp;
            }
        }
    }
    return null;
```

Figure 11—Bloom (175B) using Javascript instead of Python3 to complete a function with the *Anonymize* transformation.

```
def func(decimal):
    "You will be given a decimal form and your task is to convert it to binary format. The function should return a string, with
    ↪  each character representing a binary number. Each character in the string will be '0' or '1'. There will be an extra
    ↪  couple of characters 'db' at the beginning and at the end of the string. The extra characters are there to help with the
    ↪  format."

    </file source=github filename=binary_numbers.py |>
    <|c|>
    Why does the db string appear in the output? And why does 1 get converted to the empty string?.
    My code also leaves out the 0 at the beginning.
    <|/c|>
    <| a tags=python-3.x, python |>
    <code>
    n = int(input('Give the number you want in binary form'))
    remainder = n%2
```

Figure 12—Incoder (6B) disclosing the name of a file as well as some human-like questions when faced with a *Anonymize + Drop One* transformation.

```
from typing import List

def func(numbers: List[float]) -> List[float]:
    "Given list of numbers (of at least two elements), apply a linear transform to that list, such that the smallest number will
    ↪  become 0 and the largest will become 1."
    >>> func((1.0, 2.0, 3.0, 4.0, 5.0])
    [0.0, 0.25, 0.5, 0.75, 1.0]

    positives = [num > 0 for num in numbers]
    if all(positives):
        centred = sum(positives) / 2
        return [num - centred for num in numbers]
    else:
        return [ -num for num in numbers]

    for _ in range(10):
        np.random.seed(0)
        randoms = np.random.rand(100, 5)
        randoms = np.sort(randoms, axis=0)
        target = randoms[:, 0]
        print(target, func(randoms))
        print(np.sum(func(randoms)-target)/np.sum(target))

    Perfect predictions! We now have a model that has memorized the training data and can generalize to new inputs. This would be
    ↪  a great time for us to try and create a model which generalizes weil to unseen data points, of, even better, new inputs to
    ↪  the data. So lets do that now. We will use the same training data as before, with some slight modifications to make use
    ↪  of sklearns pipeline capabilities, along with some feature engineering using Pandas and our dataset of housing prices.

    prices = pd.read_csv('../input/global-wheat-price-prediction/train.csv', parse_dates=['date'])
    prices.head()
```

Figure 13—Incoder (1.6B) adding some snippet of ambiguous functionality followed by something that looks like some exercise comments.

```
def func(lst):
    "You are given a non-empty list of positive integers. Return the greatest integer that is greater than zero, and has a
    ↪  frequency greater than or equal to the value of the integer itself. The frequency of an integer is the number of times it
    ↪  appears in the list. If no such a value exists, return -1."
    Examples:
    func([4, 1, 2, 2, 3, 1]) == 2
    func([1, 2, 2, 3, 3, 3, 4, 4, 4]) == 3
    func([5, 5, 4, 4, 4]) == -1
    # your code here
    return -1

def func(lst):
    "You are given a non-empty list of positive integers. Return the greatest integer that is greater than zero, and has a
    ↪  frequency greater than or equal to the value of the integer itself. The frequency of an integer is the number of times it
    ↪  appears in the list. If no such a value exists, return -1."
    Examples:
    func([4, 1, 2, 2, 3, 1]) == 2
    func([1, 2, 2, 3, 3, 3, 4, 4, 4]) == 3
    func([5, 5, 4, 4, 4]) == -1
    # your code here
    # Base Case
    If not last: return -1


def func(lst):
    "You are given a non-empty list of positive integers. Return the greatest integer that is greater than zero, and has a
    ↪  frequency greater than or equal to the value of the integer itself. The frequency of an integer is the number of times it
    ↪  appears in the list. If no such a value exists, return -1."
    Examples:
    func([4, 1, 2, 2, 3, 1]) == 2
    func([1, 2, 2, 3, 3, 3, 4, 4, 4]) == 3
    func([5, 5, 4, 4, 4]) == -1
    d = {}
    for i in lst:
        if i not in d:
            d[i] = 1
        else:
            d[i] += 1
    for i in d:
        if d[i] >= i:
            return i
    return -1
```

Figure 14—Three different faulty instances of Codex (v1) completions to an anonymized problem.

10.7 Algorithms

Algorithm 1 Block of Influence Splitting

cc : Code Challenge Instance
Locate function name, which is the next token after the last matched "def", and keep start and end index of it.
$name, start\_name\_index, end\_name\_index \leftarrow NameMatch(cc)$
Anything prior to the match, such as imports or helper functions is considered prefix.
$prefix \leftarrow cc[: start\_name\_index]$
# Look for tokens such as (Example, example, >, ≫). If no matches were found, look for uses of the function name in the challenge.
if $ExampleMatch(cc[end\_name\_index :]) \neq None$ **then**
    $examples, start\_example\_index \leftarrow ExampleMatch(cc[end\_name\_index :])$
else
    $examples, start\_example\_index \leftarrow FunctionMatch(cc[end\_name\_index :])$
end if
# The description should fall between the function name and the examples.
$description \leftarrow cc[end\_name\_index : start\_example\_index]$
Form the blocks and return.
$NameBlock \leftarrow prefix + name$
$DescriptionBlock \leftarrow description$
$ExampleBlock \leftarrow examples$

Algorithm 2 Keyword Identification

KB : The KeyBert model
nb : Name Block
db : Description Block
eb : Example Block
$kw :\leftarrow \emptyset$ Keywords
$fkw :\leftarrow \emptyset$ Filtered Keywords
Use the model to extract some initial unigram and bigram keywords.
$kw \leftarrow KB(db)$
# Filter out keywords non-related to coding.
for i *in* kw **do**
    if $cossim(i, [Python, Programming, Code]) > 0.7$ **then**
        if $stem(i) \in [nb, eb]$ *or* $equiv(i) \in [nb, eb]$ **then**
            $fkw \leftarrow i$
        end if
    end if
end for
return

Algorithm 3 Transformation and Execution

CM : The code generation model
cc : A coding challenge instance
nb : Name Block
fkw : Filtered Keywords
db : Description Block
eb : Example Block
org_pa1 : Original Pass@1 score
tra_pa1 : Transformed Pass@1 score
org_pa100 : Original Pass@100 score
tra_pa100 : Transformed Pass@1 score
$mode$: The transformation mode
Measure initial performance on the challenge
$org\_pa1, org\_pa100 \leftarrow CM(cc, T = 0.2), CM(cc, T = 0.8)$
if $mode = 0$ **then**
    $cc\_new \leftarrow swap(nb, "func") + db + eb$ # Anonymization
else if $mode = 1$ **then**
    $cc\_new \leftarrow nb + remove\_kw(db, choose\_single(fkw)) + eb$ # Drop One
else if $mode = 2$ **then**
    $cc\_new \leftarrow nb + remove\_kw(db, fkw) + eb$ # Drop All
else if $mode = 3$ **then**
    $cc\_new \leftarrow nb + db$ # Drop Examples
else if $mode = 4$ **then**
    $cc\_new \leftarrow swap(nb, "func") + remove\_kw(db, choose\_single(fkw)) + eb$ # Anonymization + Drop One
else if $mode = 5$ **then**
    $cc\_new \leftarrow swap(nb, "func") + remove\_kw(db, fkw) + eb$ # Anonymization + Drop All
else if $mode = 6$ **then**
    $cc\_new \leftarrow swap(nb, "func") + db$ # Anonymization + Drop Examples
end if
$tra\_pa1, tra\_pa100 \leftarrow CM(cc\_new, T = 0.2), CM(cc\_new, T = 0.8)$
$dif\_1 \leftarrow \frac{tra\_pa1 - org\_pa1}{tra\_pa1}$
$dif\_100 \leftarrow \frac{tra\_pa100 - org\_pa100}{tra\_pa100}$
return dif_1, dif_100

10.8 On the effect of function names

Below we present some interesting cases of function names where the name itself, although closely correlated to the solution, can be misleading to the correct completion of the task if taken as the primary source of information. We provide our intuitions and completions from Codex-v2 when asked to generate a function based only on its name.

- **Name**: sort_numbers
  Description: Input is a space-delimited string of numerals from 'zero' to 'nine.' Valid choices are 'zero,' 'one,' 'two,' 'three,' 'four,' 'five,' 'six,' 'seven,' 'eight,' and 'nine.' Return the string with numbers sorted from smallest to largest.
  Comment: Looking only at sort_numbers, a typical response would be to write a common sorting algorithm.
  Most common Codex completions @ (T=0.2, p=0.95) and @ (T=0.6, p=0.95):

```
def sort_numbers(l):
    return sorted(l, key=lambda x: x[1])

def sort_numbers(l):
    return sorted(l, key=int)
```

- **Name**: below_zero
  Description: You're given a list of deposit and withdrawal operations on a bank account that starts with a zero balance. Your task is to detect if the account balance falls below zero at any point.
  Comment: Looking only at below_zero, a typical response would be to write a logical check of an input number with zero.
  Most common Codex completions @ (T=0.2, p=0.95) and @ (T=0.6, p=0.95):

```
def below_zero(x):
    return x < 0

def below_zero(x):
    if x > 0:
        return False
    else:
        return True
```

- **Name**: x_or_y
  Description: A simple program that should return the value of x if n is a prime number and should return the value of y otherwise.
  Comment: Looking only at x_or_y, a typical response would be to write a logical check of $x \wedge y$.
  Most common Codex completions @ (T=0.2, p=0.95) and @ (T=0.6, p=0.95):

```
def x_or_y(x,y,n):
    if x == n:
        return y
    elif y == n:
        return x
    else:
        return n

def x_or_y(x,y,n):
    if n == 1:
        return x
    elif n == 2:
        return y
    else:
        return None
```

- **Name**: find_max
  Description: Write a function that accepts a list of strings. The list contains different words. Return the word with the maximum number of unique characters. If multiple strings have the maximum number of unique characters, return the one which comes first in lexicographical order.
  Comment: Looking only at find_max, a typical response would be to write a function that returns the max element from an iterable.
  Most common Codex completions @ (T=0.2, p=0.95) and @ (T=0.6, p=0.95):

```
def find_max(words):
    max_len = 0
    max_word = ''
    for word in words:
        if len(word) > max_len:
            max_len = len(word)
            max_word = word
    return max_word

def find_max(words):
    max=0
    for word in words:
        if len(word) > max:
        max=len(word)
    return max
```

10.9 Looking for copying patterns in open-source dataset

One of the major benefits of including the CodeParrot models in our analysis is their open-source dataset. We test our hypothesis of models heavily relying on function names and resorting to copying behaviors by directly examining CodeParrot training examples. Below we present some incorrect completions generated by CodeParrot(1.5B) and the "heavily inspired" files where the responses were found. Note that in the case of anonymization, responses differ.

- **1 - CodeParrot Completion**

```
def generate_random_string(s):
    "Write a function that generates a random string using the seed s."
    return " ".join(random.sample(string.ascii_letters + string.digits,s))
```

This is found in:
'blockly/appengine/storage.py'
'Android/platforms/android/assets/www/Blockly/appengine/storage.py'

CodeParrot Completion Anonymized

```
def func(s):
    "Write a function that generates a random string using the seed s."
    return random.choice(s)
```

- **2 - CodeParrot Completion**

```
def perimeter(s):
    "Return the perimeter of a square with side length s."
    return perimeter(s).magnitude()
```

This is found in:
'bretglun/fwqpbo/fatWaterSeparation.py'
'indico/numpy/distutils/tests/test_misc_util.py'

CodeParrot Completion Anonymized

```
def func(s):
    "Return the perimeter of a square with side length s."
    return (s * 0.5)
```

- **3 - CodeParrot Completion**

```
def tostring(element):
    "Convert the given element to string representation"
    return str(element.toxml())
```

This is found in:
'warrior/Framework/Utils/xml_Utils.py'

CodeParrot Completion Anonymized

```
def func(s):
    "Convert the given element to string representation"
    return str(element)
```

- **4 - CodeParrot Completion**

```
def add(s, p):
    "Add s to p and return the result"
    if not s:
        return None
    if not isinstance(p, list):
        p = [p]
        p.append(s)
    return
```

This is found in:
'couchpotato/core/plugins/status/main.py'
'modules/wifi/nano-c047.12/WiFiEngine/tools/axfbin.py'

CodeParrot Completion Anonymized

```
def func(s):
    "Add s to p and return the result"
    return s + p
```

- **5 - CodeParrot Completion**

```
def count_distinct_characters(string: str):
  "Given a string, find out how many distinct characters (regardless of case) does it consist of"
  count = 0
  for char in string:
    if char in string:
        count += 1
    return count
```

This is found in:
'desktop/core/ext-py/guppy-0.1.10/guppy/heapy/Prof.py'
'cpp/scons/scons-local-2.0.0.final.0/SCons/Util.py'

CodeParrot Completion Anonymized

```
def func(string: str):
  "Given a string, find out how many distinct characters (regardless of case) does it consist of"
    return len(re.findall(r"[^a-zA-Z0-9]", string))
```

10.10 Attention View

In this section, we present illustrations of attention patterns. We use Codeparrot (330M) as our target model, before and after the combined finetuning process and create visualizations for two coding challenges. The first challenge is:

```
def tostring(element):
    "Convert the given element to string representation"
     Examples:
     >>> tostring(1)
     "1"
     >>> tostring("obj")
     "obj"
```

and the second challenge is:

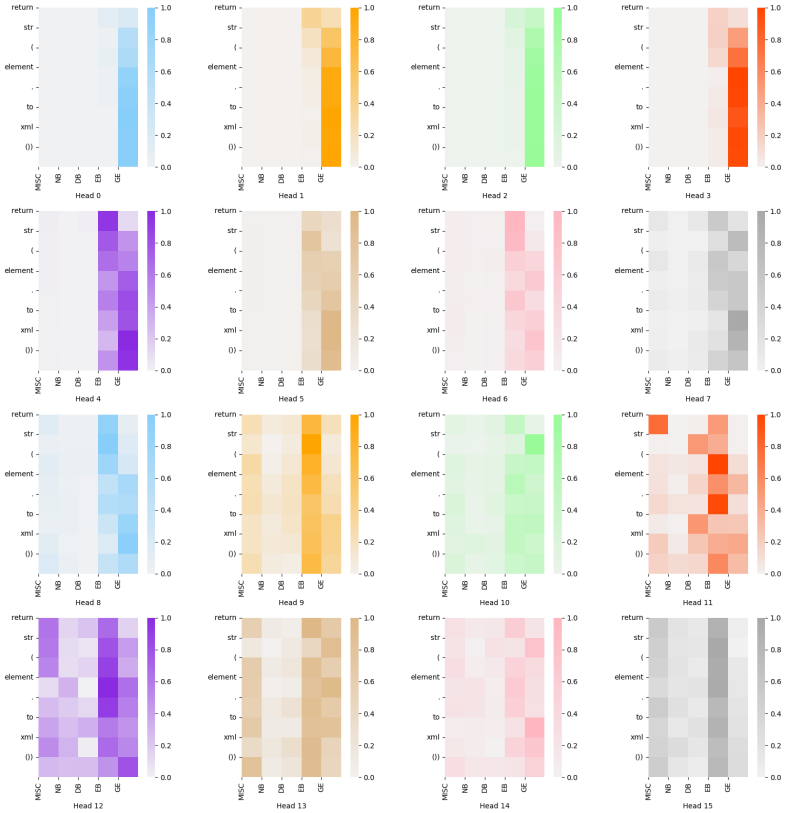
```
import math

def perimeter(s):
    "Return the perimeter of a square with side length s."
    Examples:
    >>> perimeter(1)
    1
    >>> perimeter(math.sqrt(2))
    2
```

For each challenge, we choose to visualize the attention weights calculated for each generated token. We group together tokens of each challenge into five categories:

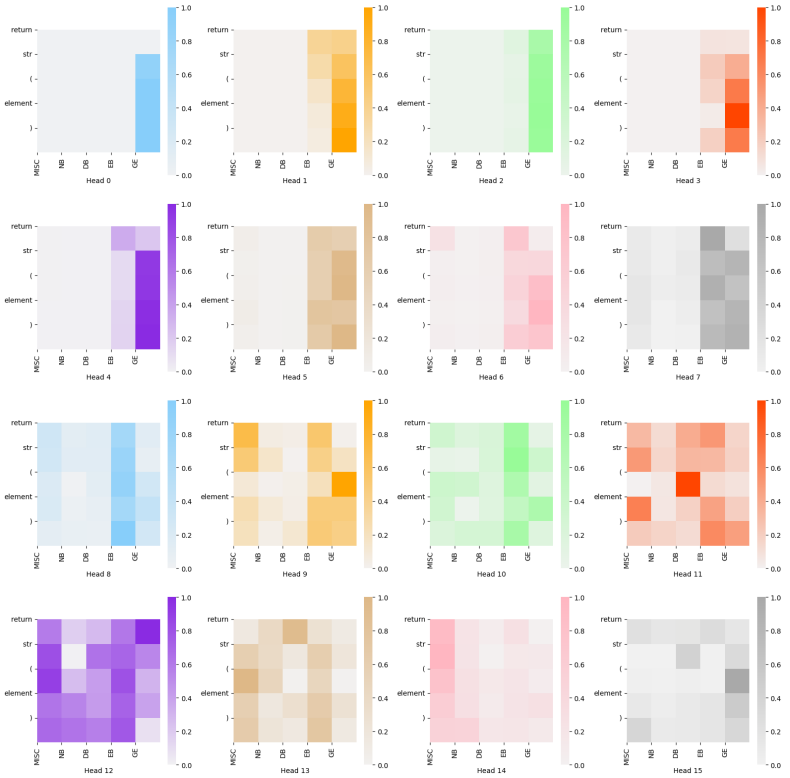
- **NB**: All tokens belonging to the *Name Block*
- **DB**: All tokens belonging to the *Description Block*
- **EB**: All tokens belonging to the *Example Block*
- **GE**: The so-far model generated tokens (solution)
- **MISC**: Any remaining tokens such as prefixes and imports.

Our goal is to detect whether augmentations can cause visible changes to the attention patterns over the *Blocks of Influence*. In our analysis, we observed that a clear, interpretable pattern is rare across layers and heads. This result is in accordance with visualizations provided in (Li et al., 2022) [2], where a far stronger model exhibits patterns that can be not so intuitive. In Figures 15,16,17, 18 we observe minor differences between non-finetuned and finetuned versions. The underlying changes in the reasoning processes of our coding models are not directly visible with attention maps. Reasoning processes should be viewed as an effect emergent from multiple interactions across layers and heads and can thus not always be located in a specific part of them.

[2] https://alphacode.deepmind.com/

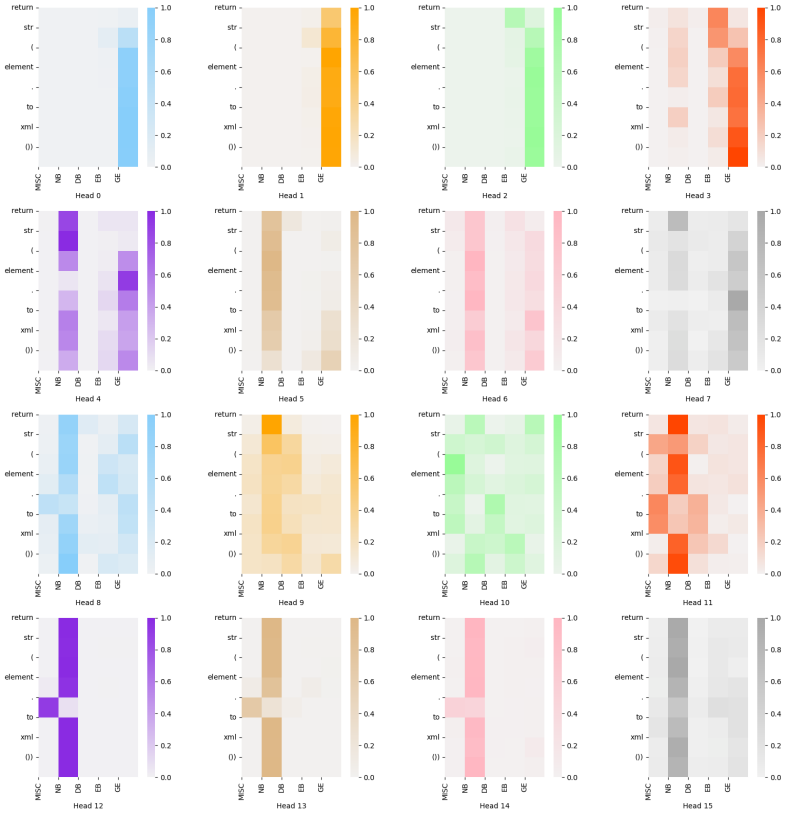


(a)

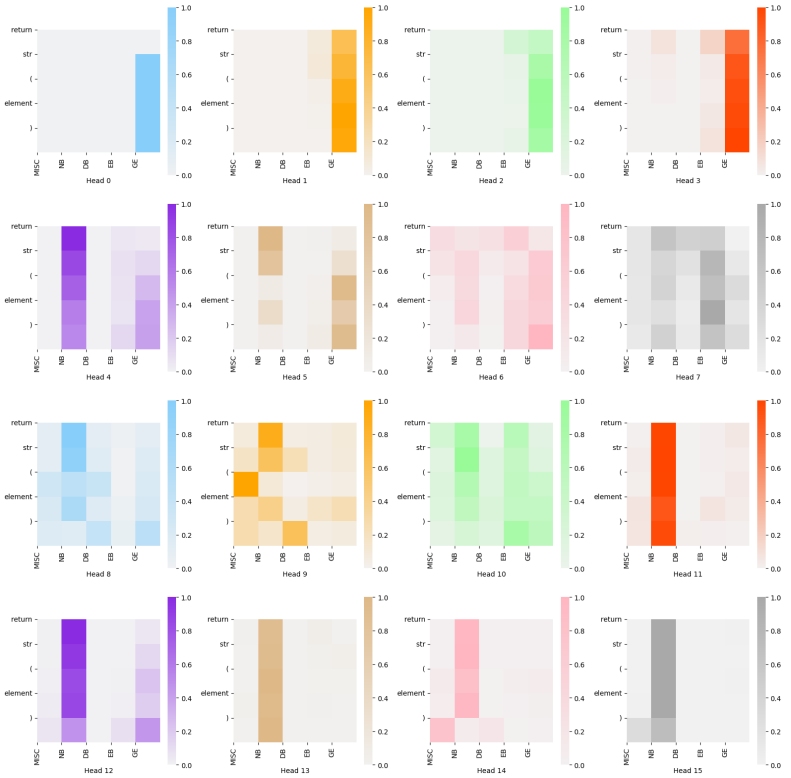


(b)

Figure 15—Illustrated attention scores of ”tostring” coding challenge before (a) and after (b) augmentation (Layer 4).

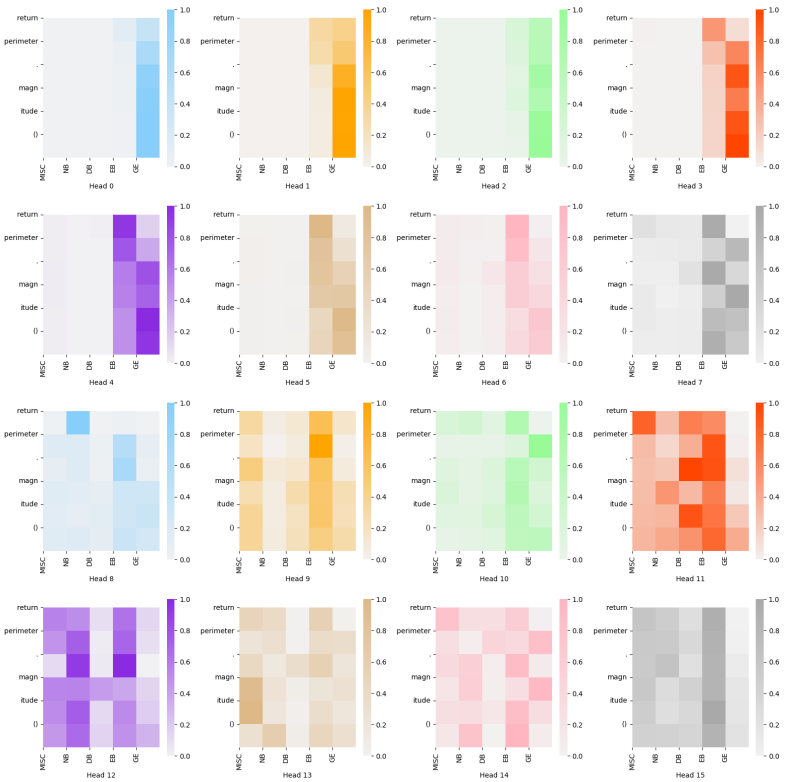


(a)

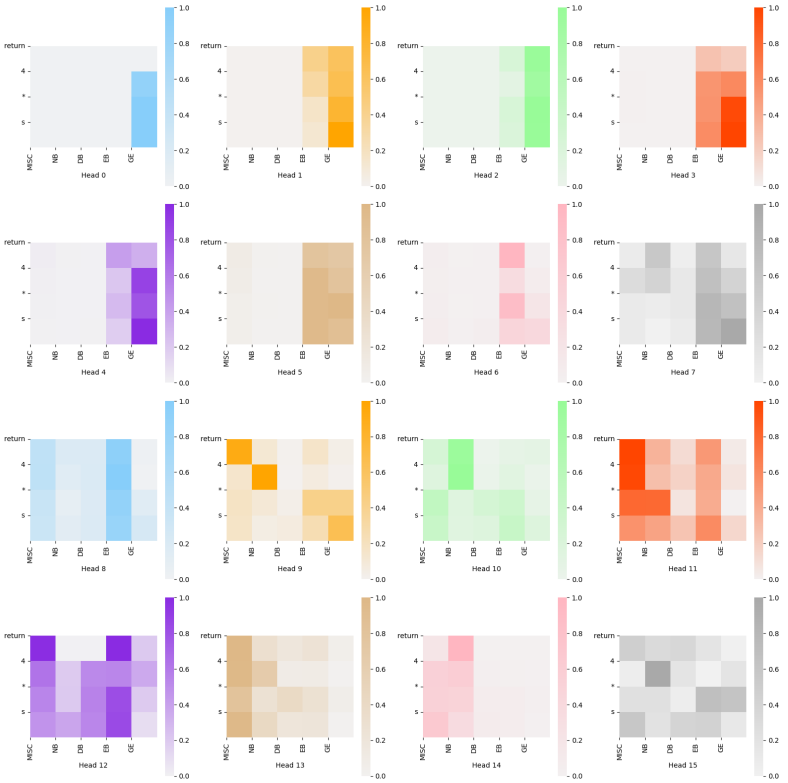


(b)

Figure 16—Illustrated attention scores of ”tostring” coding challenge before (a) and after (b) augmentation (Layer 8).

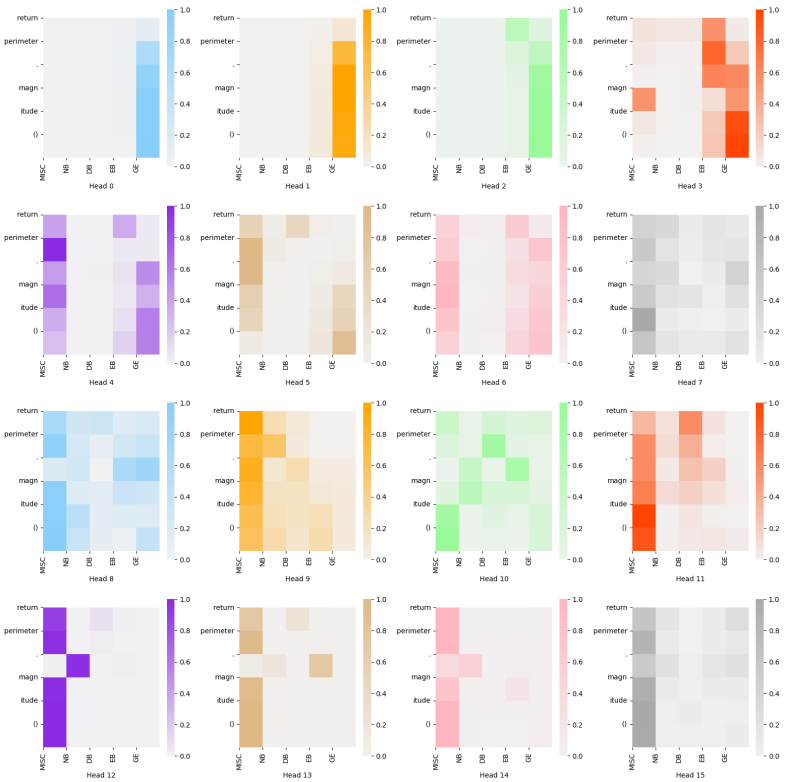


(a)

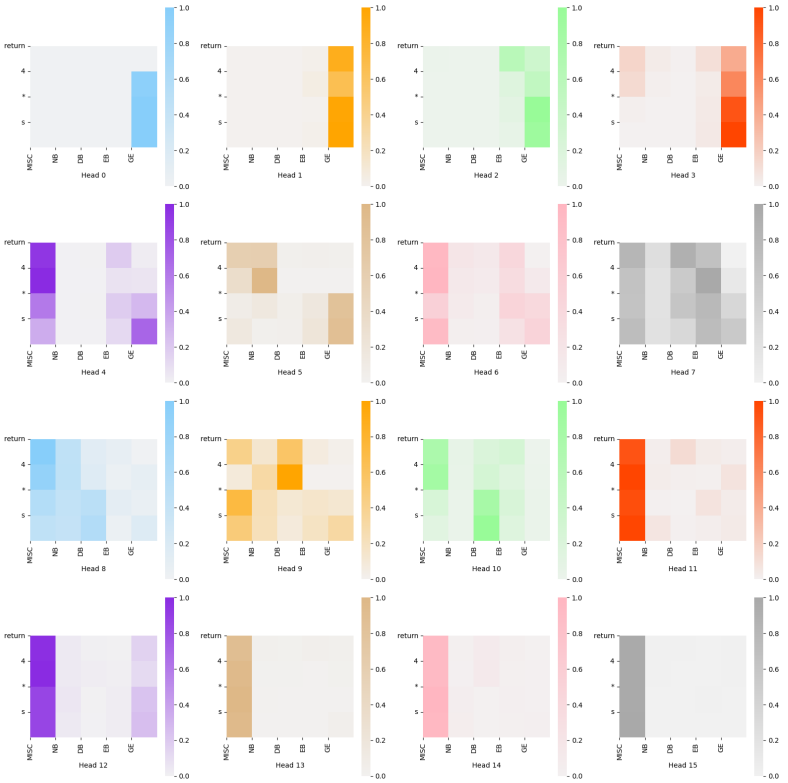


(b)

Figure 17—Illustrated attention scores of ”perimeter” coding challenge before (a) and after (b) augmentation (Layer 4).



(a)



(b)

Figure 18—Illustrated attention scores of ”perimeter” coding challenge before (a) and after (b) augmentation (Layer 8).

ACL 2023 Responsible NLP Checklist

A For every submission:

☑ A1. Did you describe the limitations of your work?
The limitations of our work are discussed in Section 8: Limitations.

☑ A2. Did you discuss any potential risks of your work?
Our work's potential risks and ethical concerns are discussed in Section 9: Risks and Ethical Considerations.

☑ A3. Do the abstract and introduction summarize the paper's main claims?
We believe the abstract and the paper's introduction in Section 1 summarized our claims.

☒ A4. Have you used AI writing assistants when working on this paper?
Left blank.

B ☑ Did you use or create scientific artifacts?

We used models and datasets as well as created finetuned versions of models. We describe all of the artifacts in Section 4 (Table 1) and Appendix Section 10.2.

☑ B1. Did you cite the creators of artifacts you used?
We cite all artifact authors, as seen in Section 4 (Table 1).

☑ B2. Did you discuss the license or terms for use and / or distribution of any artifacts?
We provide a list of each artifact's License in Appendix Section 10.2. A discussion of their compliant use is there as well.

☑ B3. Did you discuss if your use of existing artifact(s) was consistent with their intended use, provided that it was specified? For the artifacts you create, do you specify intended use and whether that is compatible with the original access conditions (in particular, derivatives of data accessed for research purposes should not be used outside of research contexts)?
We ensured that we used all artifacts to comply with their intended use and license. In our work, this is simply a verification of their performance. We did not plan to modify or redistribute any artifacts.

☒ B4. Did you discuss the steps taken to check whether the data that was collected / used contains any information that names or uniquely identifies individual people or offensive content, and the steps taken to protect / anonymize it?
Our work focuses on coding datasets that refer to coding competitions and are annotated or curated by their respective authors. Phenomena such as offensive content or sensitive information do not exist among them. However, in Section 9 (Potential risks and ethical concerns), we specify that generative models can leak such data from their pre-training phases which are out of the scope of our work.

☐ B5. Did you provide documentation of the artifacts, e.g., coverage of domains, languages, and linguistic phenomena, demographic groups represented, etc.?
Not applicable. We analyze the models and data used in Sections 3 and 4. We do not provide data regarding domains, languages, or linguistic phenomena, since we are interested in the python programming language and three associated coding datasets.

☑ B6. Did you report relevant statistics like the number of examples, details of train / test / dev splits, etc. for the data that you used / created? Even for commonly-used benchmark datasets, include the number of examples in train / validation / test splits, as these provide necessary context for a reader to understand experimental results. For example, small differences in accuracy on large test sets may

The Responsible NLP Checklist used at ACL 2023 is adopted from NAACL 2022, with the addition of a question on AI writing assistance.

be significant, while on small test sets they may not be.
All relevant information for dataset sizes, examples, and models are presented in Appendix Section 10.2

C ☑ Did you run computational experiments?

The results of our computational experiments can be found in Tables 3,4 and 5. Detailed analysis can be found in the Appendix, Section 10.3.

☑ C1. Did you report the number of parameters in the models used, the total computational budget (e.g., GPU hours), and computing infrastructure used?
Detailed analysis can be found in Table 1 and the Appendix, Section 10.3.

☑ C2. Did you discuss the experimental setup, including hyperparameter search and best-found hyperparameter values?
The grid-search values and the final hyperparameter choices are discussed in the Appendix, Section 10.3.

☑ C3. Did you report descriptive statistics about your results (e.g., error bars around results, summary statistics from sets of experiments), and is it transparent whether you are reporting the max, mean, etc. or just a single run?
We discuss the use of the pass@k metric and the nature of the results of Table 3 (average of 10 runs with different seeds) in Section 4, subsection Performance Metrics. Furthermore, Table 3 results enhanced with their variance can be found in Appendix Section 10.5. For the results of Tables 4 and 5, their captions describe the descriptive statistics of their contents. They are results averaged over 15 runs with different seeds.

☒ C4. If you used existing packages (e.g., for preprocessing, for normalization, or for evaluation), did you report the implementation, model, and parameter settings used (e.g., NLTK, Spacy, ROUGE, etc.)?
The preprocessing pipeline only used an existing model for which a link and a reference were provided. The rest was our custom codebase and built-in python functions. No other packages were used.

D ☒ Did you use human annotators (e.g., crowdworkers) or research with human participants?

Left blank.

☐ D1. Did you report the full text of instructions given to participants, including e.g., screenshots, disclaimers of any risks to participants or annotators, etc.?
Not applicable. Left blank.

☐ D2. Did you report information about how you recruited (e.g., crowdsourcing platform, students) and paid participants, and discuss if such payment is adequate given the participants' demographic (e.g., country of residence)?
Not applicable. Left blank.

☐ D3. Did you discuss whether and how consent was obtained from people whose data you're using/curating? For example, if you collected data via crowdsourcing, did your instructions to crowdworkers explain how the data would be used?
Not applicable. Left blank.

☐ D4. Was the data collection protocol approved (or determined exempt) by an ethics review board?
Not applicable. Left blank.

☐ D5. Did you report the basic demographic and geographic characteristics of the annotator population that is the source of the data?
Not applicable. Left blank.