OctoPack: Instruction Tuning Code Large Language Models

Niklas Muennighoff   **Qian Liu**   **Armel Zebaze**   **Qinkai Zheng**   **Binyuan Hui**
Terry Yue Zhuo   **Swayam Singh**   **Xiangru Tang**   **Leandro von Werra**   **Shayne Longpre**
`n.muennighoff@gmail.com`

Abstract

Finetuning large language models (LLMs) on instructions leads to vast performance improvements on natural language tasks. We apply instruction tuning using code, leveraging the natural structure of Git commits, which pair code changes with human instructions. We compile COMMITPACK: 4 terabytes of Git commits across 350 programming languages. We benchmark COMMITPACK against other natural and synthetic code instructions (xP3x, Self-Instruct, OASST) on the 16B parameter StarCoder model, and achieve state-of-the-art performance among models not trained on OpenAI outputs, on the HumanEval Python benchmark (46.2% pass@1). We further introduce HUMANEVALPACK, expanding the HumanEval benchmark to a total of 3 coding tasks (Code Repair, Code Explanation, Code Synthesis) across 6 languages (Python, JavaScript, Java, Go, C++, Rust). Our models, OCTOCODER and OCTOGEEX, achieve the best performance across HUMANEVALPACK among all permissive models, demonstrating COMMITPACK's benefits in generalizing to a wider set of languages and natural coding tasks. Code, models and data are freely available at `https://github.com/bigcode-project/octopack`.
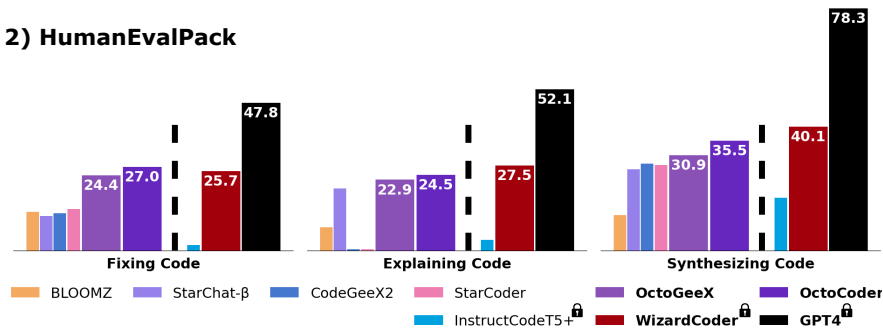
1) CommitPack

2) HumanEvalPack

Figure 1: **OCTOPACK Overview.** *1)* Sample from our 4TB dataset, COMMITPACK. *2)* Performance of OCTOCODER, OCTOGEEX and other code models including non-permissive ones (WizardCoder, GPT-4) on HUMANEVALPACK spanning 3 coding tasks and 6 programming languages.

1 Introduction

Finetuning large language models (LLMs) on a variety of language tasks explained via instructions (instruction tuning) has been shown to improve model usability and general performance (Wei et al., 2022; Sanh et al., 2022; Min et al., 2022; Ouyang et al., 2022). The instruction tuning paradigm has also proven successful for models trained on visual (Liu et al., 2023a; Li et al., 2023a), audio (Zhang et al., 2023b) and multilingual (Muennighoff et al., 2022b; Wang et al., 2022b) data.

In this work, we instruction tune LLMs on the coding modality. While Code LLMs can already be indirectly instructed to generate desired code using code comments, this procedure is brittle and does not work when the desired output is natural language, such as explaining code. Explicit instructing tuning of Code LLMs may improve their steerability and enable their application to more tasks. Concurrently to our work, three instruction tuned Code LLMs have been proposed: PanGu-Coder2 (Shen et al., 2023), WizardCoder (Luo et al., 2023) and InstructCodeT5+ (Wang et al., 2023c). These models rely on more capable and closed models from the OpenAI API[1] to create their instruction training data. This approach is problematic as **(1)** closed-source APIs keep changing and have unpredictable availability (Pozzobon et al., 2023; Chen et al., 2023a), **(2)** it relies on the assumption that a more capable model exists **(3)** it can reinforce model hallucination (Gudibande et al., 2023) and **(4)**, depending on legal interpretation, OpenAI's terms of use[2] forbid such models: "...You may not...use output from the Services to develop models that compete with OpenAI...". Thus, we consider models trained on OpenAI outputs not usable for commercial purposes in practice and classify them as non-permissive in this work.

We focus on more permissively licensed data and avoid using a closed-source model to generate synthetic data. We benchmark four popular sources of code instruction data: **(1)** xP3x (Muennighoff et al., 2022b), which contains data from common code benchmarks, **(2)** Self-Instruct (Wang et al., 2023a) data we create using a permissive Code LLM, **(3)** OASST (Köpf et al., 2023), which contains mostly natural language data and few code examples and **(4)** COMMITPACK, our new 4TB dataset of Git commits. Instruction tuning's primary purpose is to expand models' generalization abilities to a wide variety of tasks and settings. Thus, we extend the code synthesis benchmark, HumanEval (Chen et al., 2021; Zheng et al., 2023), to create HUMANEVALPACK: A code benchmark covering code synthesis, code repair, and code explanation across six programming languages.

Instruction tuning StarCoder (Li et al., 2023b) on a filtered variant of COMMITPACK and OASST leads to our best model, OCTOCODER, which surpasses all other openly licensed models (Figure 1), but falls short of the much larger GPT-4 (OpenAI, 2023). GPT-4 is close to maximum performance on the code synthesis variant, notably with a pass@1 score of 86.6% on Python HumanEval. However, it performs significantly worse on the code fixing and explanation variants of HUMANEVALPACK, which we introduce. This suggests that the original HumanEval benchmark may soon cease to be useful due to models reaching close to the maximum performance. Our more challenging evaluation variants provide room for future LLMs to improve on the performance of the current state-of-the-art.

In summary, we contribute:

- COMMITPACK and COMMITPACKFT: 4TB of permissively licensed code commits across 350 programming languages for pretraining and a filtered variant containing high-quality code instructions for finetuning

- HUMANEVALPACK: A benchmark for Code LLM generalization, spanning three scenarios (Code Repair, Code Explanation, Code Synthesis) and 6 programming languages (Python, JavaScript, Java, Go, C++, Rust)

- OCTOCODER and OCTOGEEX: The best permissive Code LLMs

2 COMMITPACK: Code Instruction Data

Prior work has shown that models can generalize to languages included in pretraining, but absent during instruction tuning (Muennighoff et al., 2022b). However, they also show that including such languages during instruction tuning boosts their performance further. We hypothesize that code data

---

[1] https://openai.com/blog/openai-api
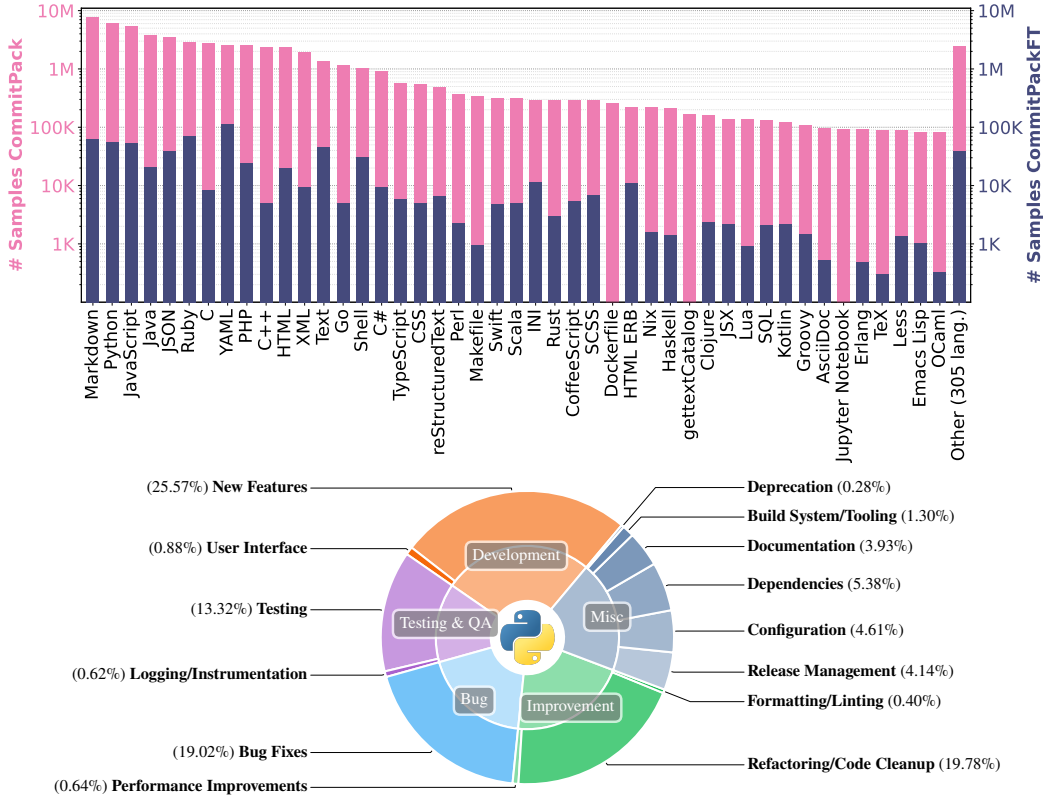[2] https://openai.com/policies/terms-of-use

Figure 2: **Overview of COMMITPACK and COMMITPACKFT.** *Top:* Language distribution of the full commit data (COMMITPACK) and the variant filtered for high-quality instructions (COMMITPACKFT). See Appendix C for the full distribution. *Bottom:* Task distribution of commits on the Python subset of COMMITPACKFT (59K samples) according to GPT-4.

Table 1: **Statistics of code instruction data we consider.** We display the number of programming languages, total samples, and fraction of samples that contain code for permissive instruction datasets. For finetuning on these datasets, we use small subsets with around 5,000 samples each.

Dataset (↓)	Base dataset			Subset		
	Lang.	Samples	Code fraction	Lang.	Samples	Code fraction
xP3x	8	532,107,156	0.67%	8	5,000	100%
StarCoder Self-Instruct	12	5,003	100%	12	5,003	100%
OASST	49	161,443	0.9%	28	8,587	2.5%
COMMITPACKFT	350	742,273	100%	6	5,000	100%

exhibits the same behavior. To improve performance on code-related tasks, we thus construct a code instruction dataset leveraging the natural structure of Git commits.

COMMITPACK To construct the dataset, we use commit metadata from the GitHub action dump on Google BigQuery.[3] We apply several quality filters, filter for commercially-friendly licenses, and discard all commits that affect more than a single file to ensure commit messages are very specific and to avoid additional complexity from dealing with multiple files. We use the filtered metadata to scrape the affected code files prior to and after the commit from GitHub. This leads to close to 4 terabytes of data covering 350 programming languages (COMMITPACK). As instruction tuning does not necessarily require so much data (Zhou et al., 2023a; Touvron et al., 2023), we apply several strict filters to reduce the dataset to 2 gigabytes (COMMITPACKFT). These strict filters include

---

[3] https://www.gharchive.org/

filtering for samples where the commit message has specific words in uppercase imperative form at the start (e.g. "Verify ..."), consists of multiple words and does not contain external references. All filters are detailed in Appendix D. Figure 2 depicts the distribution of both datasets and the tasks contained in COMMITPACKFT. For instruction tuning our models, we select 5,000 random samples from COMMITPACKFT across the 6 programming languages that we evaluate on.

Alternatives We consider three additional datasets for instruction tuning presented in Table 1. **xP3x**: xP3x is a large-scale collection of multilingual instruction data with around 532 million samples (Muennighoff et al., 2022b). We focus only on the code subset of xP3x, excluding Neural-CodeSearch (Li et al., 2019) which is not licensed permissively, and select 5,000 samples. **Self-Instruct**: Using the Self-Instruct method (Wang et al., 2022a) and the StarCoder model (Li et al., 2023b), we create 5,003 synthetic instructions and corresponding answers. **OASST**: OASST is a diverse dataset of multi-turn chat dialogues (Köpf et al., 2023). While most dialogues center around natural language, some also contain code. We reuse a filtered variant of OASST from prior work (Dettmers et al., 2023) and additionally filter out moralizing assistant answers (Appendix D) leading to 8,587 samples.

3   HUMANEVALPACK: Evaluating Instruction Tuned Code Models
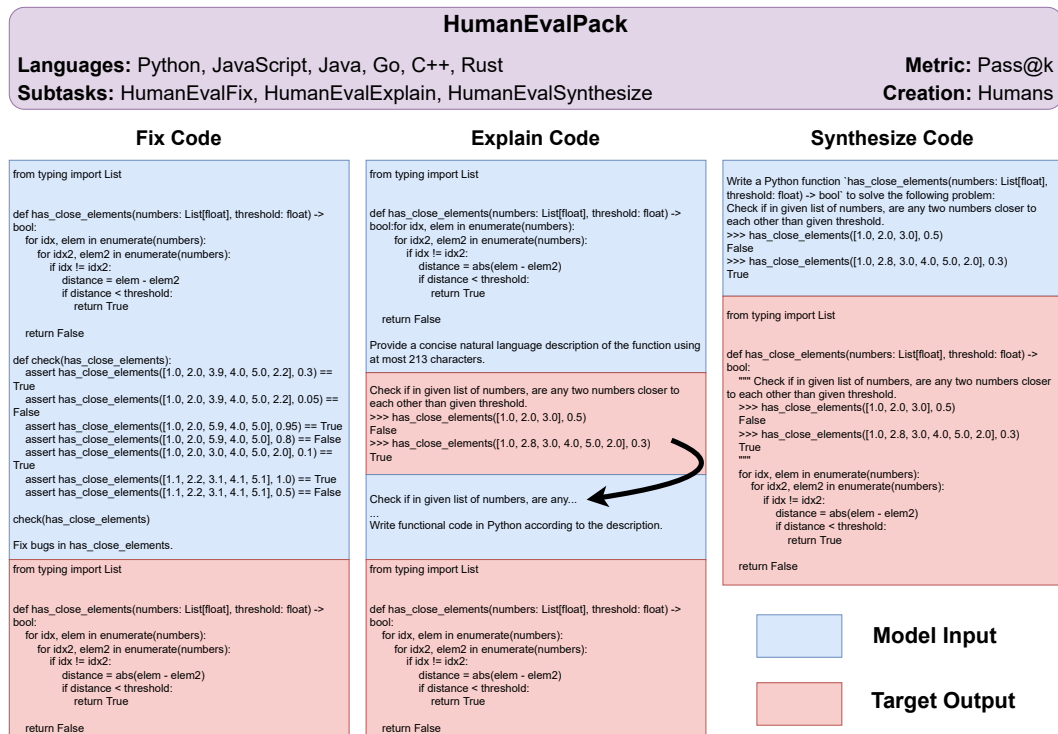
Figure 3: **HUMANEVALPACK overview.** The first HumanEval problem is depicted across the three scenarios for Python. The bug for HUMANEVALFIX consists of a missing "abs" statement.

When instruction tuning LLMs using natural language (NL) data, the input is an NL instruction with optional NL context and the target output is the NL answer to the task (Wei et al., 2022). When instruction tuning with code (C) data, code may either appear *only in the input* alongside the NL instruction (NL+C→NL, e.g. code explanation), *only in the output* (NL→C, e.g. code synthesis), or in *both input and output* (NL+C→C, e.g. code modifications like bug fixing). While prior benchmarks commonly only cover variants of code synthesis, users may want to use models in all three scenarios. Thus, we expand the code synthesis benchmark HumanEval (Chen et al., 2021; Zheng et al., 2023) to cover all three input-output combinations for six languages (Figure 3).

HUMANEVALFIX (NL+C→C) Given an incorrect code function with a subtle bug and accompanying unit tests, the model is tasked to fix the function. We manually add a bug to each of the 164 HumanEval solutions across all 6 languages (984 total bugs). For a given sample, the bugs are as similar as possible across the 6 languages enabling meaningful comparison of scores across languages. Bugs are written such that the code still runs but produces an incorrect result leading to at least one unit test failing. Bug statistics and examples are in Appendix K. We also evaluate an easier variant of this task where instead of unit tests, models are provided with the correct function docstring as the source of truth to fix bugs, see Appendix I.

HUMANEVALEXPLAIN (NL+C→NL) Given a correct code function, the model is tasked to generate an explanation of the code. Subsequently, the same model is tasked to regenerate the code given only its own explanation. The second step allows us to score this task via code execution and measure pass@k (Chen et al., 2021) instead of evaluating the explanation itself using heuristic-based metrics like BLEU (Papineni et al., 2002) or ROUGE (Lin, 2004) which have major limitations (Reiter, 2018; Schluter, 2017; Eghbali and Pradel, 2022; Zhou et al., 2023b). To prevent models from copying the solution into the description, we remove any solution overlap of at least 20 characters from the description. We further enforce a character length limit on the model-generated explanation equivalent to the length of the docstring describing the function. This limit is specified in the prompt for the model. Note that the function docstring itself is never provided to the model for this task.

HUMANEVALSYNTHESIZE (NL→C) Given a natural language docstring or comment describing the desired code, the model is tasked to synthesize the correct code. This task corresponds to the original HumanEval benchmark (Chen et al., 2021). For instruction tuned models, we add an explicit instruction to the input explaining what the model should do. For models that have only gone through language model pretraining, we follow Chen et al. (2021) and provide the model with the function header and docstring to evaluate its completion of the function.

For all tasks we execute the code generations to compute performance using the pass@k metric (Chen et al., 2021): a problem is considered solved if any of k code generations passes every test case. We focus on the simplest version of pass@k, which is pass@1: the likelihood that the model solves a problem in a single attempt. Like Chen et al. (2021), we use a sampling temperature of 0.2 and $top_p = 0.95$ to estimate pass@1. We generate $n = 20$ samples, which is enough to get reliable pass@1 estimates (Li et al., 2023b). For GPT-4, we generate $n = 1$ samples. Using $n = 1$ instead of $n = 20$ for GPT-4 only changes scores by around 2% while providing 20x cost savings.

Python HumanEval is the most commonly used code benchmark, thus many training datasets have already been decontaminated for HumanEval to enable fair evaluation. By reusing HumanEval and manually expanding it to more scenarios and languages, we ensure that existing decontamination remains valid. This enables a fair comparison across a large variety of models.

4 OCTOCODER: Best Commercially Licensed Code LLM
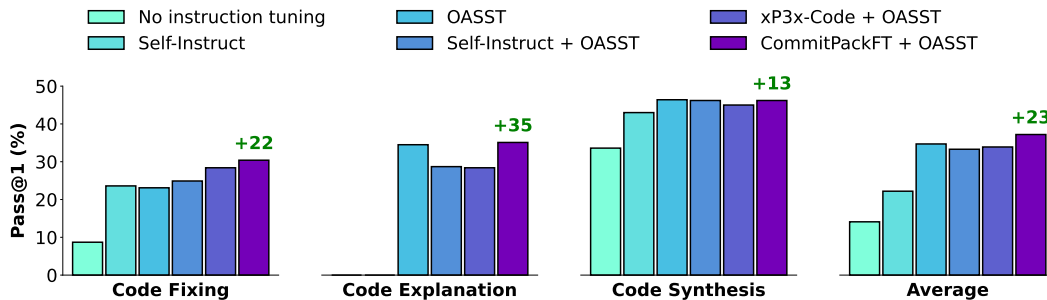
4.1 Ablating instruction data choices

Figure 4: **Comparing permissively licensed instruction datasets by instruction tuning StarCoder.** Models are evaluated on the Python subset of HUMANEVALPACK.

5

We instruction tune the pretrained StarCoder model (Li et al., 2023b) on different combinations of our instruction datasets (§2). We evaluate all models on the Python subset of HUMANEVALPACK as depicted in Figure 4. Similar to prior work (Taori et al., 2023), we format all instructions into a consistent schema to distinguish question and answer (see Figure 17).

COMMITPACKFT enables CodeLLMs to fix bugs COMMITPACKFT is critical for the performance boost on code repair (HUMANEVALFIX), where instruction tuning on only OASST or other variants results in a significantly lower score. This is likely due to COMMITPACKFT including around 20% of bug fixes among other code-related tasks (Figure 2).

Importance of samples with natural language targets The pretrained StarCoder model, as well as the Self-Instruct variant, perform poorly on code explanation (HUMANEVALEXPLAIN). This is because both models are only conditioned to write code instead of natural language. We find that to perform well at explaining code, it is necessary to include samples with natural language as the target output during instruction tuning. Only relying on data with code as the target, such as the Self-Instruct data, will lead to models always outputting code even if the question requires a natural language output. Thus, we mix all other ablations with OASST, which contains many natural language targets. While the xP3x subset also contains samples with natural language output, many of its target outputs are short, which leads to models with a bias for short answers. This is impractical for the explanation task leading to the comparatively low score of mixing xP3x with OASST.

COMMITPACKFT+OASST yields best performance All instruction datasets provide similar boosts for code synthesis (HUMANEVALSYNTHESIZE), which has been the focus of all prior work on code instruction models (Wang et al., 2023c; Luo et al., 2023; Muennighoff et al., 2022b). We achieve the best average score by instruction tuning on COMMITPACKFT mixed with our filtered OASST data yielding an absolute 23% improvement over StarCoder. Thus, we select COMMITPACKFT+OASST for our final model dubbed OCTOCODER. Using the same data, we also instruction tune the 6 billion parameter CodeGeeX2 (Zheng et al., 2023) to create OCTOGEEX.

4.2 Comparing with other Models

We benchmark OCTOCODER and OCTOGEEX with state-of-the-art Code LLMs on HUMANEVAL-PACK in Table 2. For all models, we use the prompt put forward by the model creators if applicable or else a simple intuitive prompt, see Appendix N.

OCTOCODER performs best among permissive models OCTOCODER has the highest average score across all three evaluation scenarios among all permissive models. With just 6 billion parameters, OCTOGEEX is the smallest model benchmarked, but still outperforms *all* prior permissive Code LLMs. GPT-4 (OpenAI, 2023) performs best among all models benchmarked with a significant margin. However, GPT-4 is closed-source and likely much larger than all other models evaluated.

Instruction tuning generalizes to unseen programming languages Trained primarily on natural language, not code, BLOOMZ (Muennighoff et al., 2022b) performs worse than other models despite having 176 billion parameters. Go and Rust are not contained in BLOOMZ's instruction data, yet it performs much better than the random baseline of 0.0 for these two languages across most tasks. This confirms our hypothesis that models are capable of generalizing instructions to programming languages only seen at pretraining, similar to crosslingual generalization for natural languages (Muennighoff et al., 2022b). To improve programming language generalization further, we tune OCTOCODER and OCTOGEEX on many languages from COMMITPACKFT, and this generalization improvement is reflected in the performance on HUMANEVALPACK's new languages.

Pretraining weight correlates with programming language performance after instruction tuning Prior work has shown that the performance on natural languages after instruction tuning is correlated with the weight of these languages during pretraining (Muennighoff et al., 2022b). The more weight during pretraining, the better the performance after instruction tuning. We find the same to be the case for programming languages. Python, Java, and JavaScript collectively make up around 30% of the pretraining data of StarCoder (Li et al., 2023b). After instruction tuning StarCoder to produce OCTOCODER, we see the best performance among these three languages, especially for HUMANEVALSYNTHESIZE. OCTOCODER performs weakest on Rust, which is the lowest resource language of StarCoder among the languages we benchmark (1.2% of pretraining data).

Table 2: **Zero-shot pass@1 (%) performance across HUMANEVALPACK.** InstructCodeT5+, WizardCoder, StarChat-β, StarCoder and OCTOCODER have 16B parameters. CodeGeeX2 and OCTOGEEX have 6B parameters. BLOOMZ has 176B parameters. In this work, we call models "permissive" if weights are freely accessible and usable for commercial purposes. *: Commercial license available after submitting a form. †: Trained on data that may not be used "to develop models that compete with OpenAI" thus we classify them as non-permissive in this work (see §1).

Model (↓)	Python	JavaScript	Java	Go	C++	Rust	Avg.
HUMANEVALFIX							
Non-permissive models							
InstructCodeT5+†	2.7	1.2	4.3	2.1	0.2	0.5	1.8
WizardCoder†	31.8	29.5	30.7	30.4	18.7	13.0	25.7
GPT-4	47.0	48.2	50.0	50.6	47.6	43.3	<u>47.8</u>
Permissive models							
BLOOMZ	16.6	15.5	15.2	16.4	6.7	5.7	12.5
StarChat-β	18.1	18.1	24.1	18.1	8.2	3.6	11.2
CodeGeeX2*	15.9	14.7	18.0	13.6	4.3	6.1	12.1
StarCoder	8.7	15.7	13.3	20.1	15.6	6.7	13.4
OCTOGEEX*	28.1	27.7	30.4	27.6	22.9	9.6	24.4
OCTOCODER	**30.4**	**28.4**	**30.6**	**30.2**	**26.1**	**16.5**	**27.0**
HUMANEVALEXPLAIN							
Non-permissive models							
InstructCodeT5+†	20.8	0.0	0.0	0.0	0.1	0.0	3.5
WizardCoder†	32.5	33.0	27.4	26.7	28.2	16.9	27.5
GPT-4	64.6	57.3	51.2	58.5	38.4	42.7	<u>52.1</u>
Permissive models							
BLOOMZ	14.7	8.8	12.1	8.5	0.6	0.0	7.5
StarChat-β	25.4	21.5	24.5	18.4	17.6	13.2	20.1
CodeGeeX2*	0.0	0.0	0.0	0.0	0.0	0.0	0.0
StarCoder	0.0	0.0	0.0	0.0	0.0	0.0	0.0
OCTOGEEX*	30.4	24.0	24.7	**21.7**	21.0	**15.9**	22.9
OCTOCODER	**35.1**	**24.5**	**27.3**	21.1	**24.1**	14.8	**24.5**
HUMANEVALSYNTHESIZE							
Non-permissive models							
InstructCodeT5+†	37.0	18.9	17.4	9.5	19.8	0.3	17.1
WizardCoder†	57.3	49.5	36.1	36.4	40.9	20.2	40.1
GPT-4	86.6	82.9	81.7	72.6	78.7	67.1	<u>78.3</u>
Permissive models							
BLOOMZ	15.6	14.8	18.4	8.4	6.5	5.5	11.5
StarChat-β	33.5	31.4	26.7	25.5	26.6	14.0	26.3
CodeGeeX2*	35.9	32.2	30.8	22.5	29.3	18.1	28.1
StarCoder	33.6	30.8	30.2	17.6	31.6	21.8	27.6
OCTOGEEX*	44.7	33.8	36.9	21.9	32.3	15.7	30.9
OCTOCODER	**46.2**	**39.2**	**38.2**	**30.4**	**35.6**	**23.4**	**35.5**

Models struggle with small targeted changes HUMANEVALFIX is the most challenging task for most models. They commonly regenerate the buggy function without making any change (e.g. WizardCoder in Figure 33) or they introduce new bugs (e.g. GPT-4 in Figure 32). We analyze model performance by bug type in Appendix L and find bugs that require removing excess code are the most challenging. OCTOCODER performs comparatively well across all languages. Instruction tuning on COMMITPACKFT has likely taught OCTOCODER to make small, targeted changes to fix bugs.

Models struggle switching between code and text Some models fail at HUMANEVALEXPLAIN, as they do not generate natural language explanations. We manually inspect explanations for the first ten samples of the Python split and disqualify a model if none of them are explanations. This is the case for StarCoder and CodeGeeX2, which generate code instead of natural language explanations. BLOOMZ and InstructCodeT5+ also occasionally generate code. Other models exclusively generate natural language explanations, not containing any code for inspected samples.

Models struggle adhering to a specified output length HUMANEVALEXPLAIN instructs models to fit their explanation within a given character limit (§3). Current models appear to have no understanding of how many characters they are generating. They commonly write very short and thus underspecified explanations (e.g. BLOOMZ in Figure 34) or excessively long explanations that end up being cut off (e.g. StarChat-β in Figure 37). Future work could investigate how to enable models to be aware of their generated output length to improve HUMANEVALEXPLAIN performance.

HumanEval code synthesis is close to saturation Pure code synthesis on HUMANEVALSYNTHESIZE is the easiest task for all models. With a pass rate of 86.6% for a single solution, GPT-4 is close to fully saturating the Python subset. GPT-4 was originally found to score 67% on Python HumanEval (OpenAI, 2023) and 81% in later work (Bubeck et al., 2023). Our score for GPT-4 is significantly higher, possibly due to improvements made to the API by OpenAI, contamination of HumanEval in GPT-4 training, or slightly different prompting and evaluation. An example of our prompt is depicted in Figure 3 (right). We perform very careful evaluation to ensure every generation is correctly processed. We reproduce the HumanEval score of WizardCoder (Luo et al., 2023; Xu et al., 2023a) and find it to also perform well across other languages. For BLOOMZ and InstructCodeT5+ our evaluation leads to a higher Python score than they reported, likely because of our more careful processing of generations. OCTOCODER has the highest performance for every language among permissively licensed models. With a pass@1 of 46.2% on the original Python split, OCTOCODER improves by a relative 38% over its base model, StarCoder.

5   Related Work

5.1   Code Models

There has been extensive work on code models tailored to a specific coding task, such as code summarization (Iyer et al., 2016; Ahmad et al., 2020; Zhang et al., 2022a; Shi et al., 2022) or code editing (Drain et al., 2021; Zhang et al., 2022c; He et al., 2022; Zhang et al., 2022b; Wei et al., 2023; Prenner and Robbes, 2023; Fakhoury et al., 2023; Skreta et al., 2023) (also see work on edit models more generally (Reid and Neubig, 2022; Schick et al., 2022; Dwivedi-Yu et al., 2022; Raheja et al., 2023)). These works use task-specific heuristics that limit the applicability of their methods to other tasks. In contrast, we aim to build models applicable to all kinds of tasks related to code and beyond.

Through large-scale pretraining more generally applicable code models have been developed (Nijkamp et al., 2022, 2023; Xu et al., 2022a; Christopoulou et al., 2022; Gunasekar et al., 2023; Li et al., 2023b; Bui et al., 2023; Scao et al., 2022a,b). However, these models only continue code making them hard to use for tasks such as explaining code with natural language (HUMANEVALEXPLAIN). Teaching them to follow human instructions is critical to make them applicable to diverse tasks.

5.2   Instruction Models

Training models to follow instructions has led to new capabilities in text (Ouyang et al., 2022; Wang et al., 2022b; Chung et al., 2022) and visual modalities (Xu et al., 2023b; OpenAI, 2023). Prior work has shown its benefits for traditional language tasks (Sanh et al., 2022; Wei et al., 2022; Longpre et al., 2023a; Iyer et al., 2022), multilingual tasks (Muennighoff et al., 2022b; Yong et al., 2022), and helpfulness in dialog (Köpf et al., 2023; Bai et al., 2022; Ganguli et al., 2022). For coding applications,

PanGu-Coder2 (Shen et al., 2023), WizardCoder (Luo et al., 2023) and InstructCodeT5+ (Wang et al., 2023c) are recent models trained with coding instructions. However, they all use the CodeAlpaca dataset (Chaudhary, 2023), which is synthetically generated from OpenAI models. Using data from powerful closed-source models provides a strong advantage, but limits the model use and has other limitations highlighted in §1. CoEditor (Wei et al., 2023) proposes an "auto-editing" task, trained on 1650 python commit history repositories. Our work expands this proposal to more general coding tasks (using instructions), more languages, and orders of magnitude more commit data.

5.3 Code Benchmarks

Many code synthesis benchmarks have been proposed (Wang et al., 2022d,c; Yu et al., 2023; Lai et al., 2023; Du et al., 2023). HumanEval (Chen et al., 2021; Liu et al., 2023b) has emerged as the standard for this task. Prior work has extended HumanEval to new programming languages via automatic translation mechanisms (Athiwaratkun et al., 2022; Cassano et al., 2023; Orlanski et al., 2023). These approaches are error-prone and only translate tests, not the actual solutions, which are needed for tasks like code explanation. Thus, we rely only on humans to create all parts of HUMANEVALPACK including test cases, correct solutions, buggy solutions, and other metadata across 6 languages.

Code repair is commonly evaluated on Quixbugs (Lin et al., 2017; Prenner and Robbes, 2021; Ye et al., 2021; Xia and Zhang, 2023; Jiang et al., 2023; Sobania et al., 2023) or Python bugs (He et al., 2022; Bradley et al., 2023). The latter does not support code execution, which limits its utility. While Quixbugs supports execution with unit tests, it only contains 40 samples in Python and Java. Further, the problems in Quixbugs are generic functions, such as bucket sort. This makes them easy to solve and hard to decontaminate training data for. Our benchmark, HUMANEVALFIX, contains 164 buggy functions for six languages with solutions and unit tests. Further, our coding problems, derived from HumanEval, are very specific, such as keeping track of a bank account balance (see Figure 14).

Prior work on evaluating code explanations (Lu et al., 2021; Cui et al., 2022) has relied on metrics such as METEOR (Banerjee and Lavie, 2005) or BLEU (Papineni et al., 2002). By chaining code explanation with code synthesis, we can evaluate this task using the execution-based pass@k metric overcoming the major limitations of BLEU and other heuristics-based metrics (Reiter, 2018).

Large-scale benchmarking has proven useful in many areas of natural language processing (Wang et al., 2019; Kiela et al., 2021; Srivastava et al., 2022; Muennighoff et al., 2022a). By producing 18 scores (6 languages across 3 tasks) for 9 models, we take a step towards large-scale benchmarking of code models. However, we lack many models capable of generating code (Black et al., 2021; Fried et al., 2022; Black et al., 2022; Wang and Komatsuzaki, 2021; Biderman et al., 2023b). Future work may consider more models or extending HUMANEVALPACK to new languages or tasks, such as code efficiency (Madaan et al., 2023a; Yetistiren et al., 2022) or code classification (Khan et al., 2023).

6   Conclusion

This work studies training and evaluation of Code LLMs that follow instructions. We introduce COMMITPACK, a 4TB dataset of Git commits covering 350 programming languages. We filter this large-scale dataset to create COMMITPACKFT, 2GB of high-quality code with commit messages that assimilate instructions. To enable a comprehensive evaluation of instruction code models, we construct HUMANEVALPACK, a human-written benchmark covering 3 different tasks for 6 programming languages. We ablate several instruction datasets and find that COMMITPACKFT combined with natural language data leads to the best performance. While our models, OCTOCODER and OCTOGEEX, are the best permissively licensed Code LLMs available, they are outperformed by closed-source models such as GPT-4. In addition to improving the instruction tuning paradigm, future work should consider training more capable base models.

Acknowledgments and Disclosure of Funding

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. SantaCoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Hannah McLean Babe, Sydney Nguyen, Yangtian Zi, Arjun Guha, Molly Q Feldman, and Carolyn Jane Anderson. 2023. StudentEval: A Benchmark of Student-Written Prompts for Large Language Models of Code. *arXiv preprint arXiv:2306.04556*.

Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. 2022. Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback. *arXiv preprint arXiv:2204.05862*.

Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72.

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275*.

Mohammad Bavarian, Heewoo Jun, Nikolas A. Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient Training of Language Models to Fill in the Middle. *arXiv preprint arXiv:2207.14255*.

Loubna Ben Allal, Niklas Muennighoff, Logesh Kumar Umapathi, Ben Lipkin, and Leandro von Werra. 2022. A framework for the evaluation of code generation models. `https://github.com/bigcode-project/bigcode-evaluation-harness`.

Stella Biderman, USVSN Sai Prashanth, Lintang Sutawika, Hailey Schoelkopf, Quentin Anthony, Shivanshu Purohit, and Edward Raf. 2023a. Emergent and predictable memorization in large language models. *arXiv preprint arXiv:2304.11158*.

Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. 2023b. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, pages 2397–2430. PMLR.

Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. 2022. GPT-NeoX-20B: An Open-Source Autoregressive Language Model. *arXiv preprint arXiv:2204.06745*.

Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. GPT-Neo: Large scale autoregressive language modeling with mesh-tensorflow. *If you use this software, please cite it using these metadata*, 58.

Herbie Bradley, Honglu Fan, Harry Saini, Reshinth Adithyan, Shivanshu Purohit, and Joel Lehman. 2023. Diff Models - A New Way to Edit Code. *CarperAI Blog*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Conference on Neural Information Processing Systems (NeurIPS)*.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*.

Nghi DQ Bui, Hung Le, Yue Wang, Junnan Li, Akhilesh Deepak Gotmare, and Steven CH Hoi. 2023. CodeTF: One-stop Transformer Library for State-of-the-art Code LLM. *arXiv preprint arXiv:2306.00029*.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2023. MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*.

Sahil Chaudhary. 2023. Code Alpaca: An Instruction-following LLaMA model for code generation. `https://github.com/sahil280114/codealpaca`.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

Lingjiao Chen, Matei Zaharia, and James Zou. 2023a. How is ChatGPT's behavior changing over time?

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*.

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. 2023b. Extending Context Window of Large Language Models via Positional Interpolation. *arXiv preprint arXiv:2306.15595*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023c. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, et al. 2022. Pangu-coder: Program synthesis with function-level language modeling. *arXiv preprint arXiv:2207.11280*.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. 2022. Scaling Instruction-Finetuned Language Models. *arXiv preprint arXiv:2210.11416*.

Haotian Cui, Chenglong Wang, Junjie Huang, Jeevana Priya Inala, Todd Mytkowicz, Bo Wang, Jianfeng Gao, and Nan Duan. 2022. CodeExp: Explanatory Code Document Generation. *arXiv preprint arXiv:2211.15395*.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*.

Kaustubh D Dhole, Varun Gangal, Sebastian Gehrmann, Aadesh Gupta, Zhenhao Li, Saad Mahamood, Abinaya Mahendiran, Simon Mille, Ashish Srivastava, Samson Tan, et al. 2021. Nl-augmenter: A framework for task-sensitive natural language augmentation. *arXiv preprint arXiv:2112.02721*.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2022. Cocomic: Code completion by jointly modeling in-file and cross-file context. *arXiv preprint arXiv:2212.10007*.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration Code Generation via ChatGPT. *arXiv preprint arXiv:2304.07590*.

Dawn Drain, Colin B Clement, Guillermo Serrato, and Neel Sundaresan. 2021. Deepdebug: Fixing python bugs using stack traces, backtranslation, and code skeletons. *arXiv preprint arXiv:2105.09352*.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation. *arXiv preprint arXiv:2308.01861*.

Jane Dwivedi-Yu, Timo Schick, Zhengbao Jiang, Maria Lomeli, Patrick Lewis, Gautier Izacard, Edouard Grave, Sebastian Riedel, and Fabio Petroni. 2022. EditEval: An Instruction-Based Benchmark for Text Improvements. *arXiv preprint arXiv:2209.13331*.

Aryaz Eghbali and Michael Pradel. 2022. CrystalBLEU: precisely and efficiently measuring the similarity of code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12.

Sarah Fakhoury, Saikat Chakraborty, Madan Musuvathi, and Shuvendu K Lahiri. 2023. Towards Generating Functionally Correct Code Edits from Natural Language Issue Descriptions. *arXiv preprint arXiv:2304.03816*.

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*.

Jinlan Fu, See-Kiong Ng, Zhengbao Jiang, and Pengfei Liu. 2023. Gptscore: Evaluate as you desire. *arXiv preprint arXiv:2302.04166*.

Deep Ganguli, Liane Lovitt, Jackson Kernion, Amanda Askell, Yuntao Bai, Saurav Kadavath, Ben Mann, Ethan Perez, Nicholas Schiefer, Kamal Ndousse, et al. 2022. Red teaming language models to reduce harms: Methods, scaling behaviors, and lessons learned. *arXiv preprint arXiv:2209.07858*.

Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, Jason Phang, Laria Reynolds, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. 2021. A framework for few-shot language model evaluation.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Deepanway Ghosal, Yew Ken Chia, Navonil Majumder, and Soujanya Poria. 2023. Flacuna: Unleashing the Problem Solving Power of Vicuna using FLAN Fine-Tuning. *arXiv preprint arXiv:2307.02053*.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2023. Critic: Large language models can self-correct with tool-interactive critiquing. *arXiv preprint arXiv:2305.11738*.

Arnav Gudibande, Eric Wallace, Charlie Snell, Xinyang Geng, Hao Liu, Pieter Abbeel, Sergey Levine, and Dawn Song. 2023. The false promise of imitating proprietary llms. *arXiv preprint arXiv:2305.15717*.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks Are All You Need. *arXiv preprint arXiv:2306.11644*.

Jingxuan He, Luca Beurer-Kellner, and Martin Vechev. 2022. On distribution shift in learning-based bug detectors. In *International Conference on Machine Learning*, pages 8559–8580. PMLR.

Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2019. Global relational models of source code. In *International conference on learning representations*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Yi Hu, Haotong Yang, Zhouchen Lin, and Muhan Zhang. 2023. Code Prompting: a Neural Symbolic Method for Complex Reasoning in Large Language Models. *arXiv preprint arXiv:2305.18507*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083.

Srinivasan Iyer, Xi Victoria Lin, Ramakanth Pasunuru, Todor Mihaylov, Daniel Simig, Ping Yu, Kurt Shuster, Tianlu Wang, Qing Liu, Punit Singh Koura, Xian Li, Brian O'Horo, Gabriel Pereyra, Jeff Wang, Christopher Dewan, Asli Celikyilmaz, Luke Zettlemoyer, and Ves Stoyanov. 2022. OPT-IML: Scaling Language Model Instruction Meta Learning through the Lens of Generalization. *arXiv preprint arXiv:2212.12017*.

Mingi Jeon, Seung-Yeop Baik, Joonghyuk Hahn, Yo-Sub Han, and Sang-Ki Ko. 2022. Deep Learning-based Code Complexity Prediction. *openreview*.

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. *arXiv preprint arXiv:2302.05020*.

Tae-Hwan Jung. 2021. Commitbert: Commit message generation using pre-trained programming language model. *arXiv preprint arXiv:2105.14242*.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2023. xCodeEval: A Large Scale Multilingual Multitask Benchmark for Code Understanding, Generation, Translation and Retrieval. *arXiv preprint arXiv:2303.03004*.

Douwe Kiela, Hamed Firooz, Aravind Mohan, Vedanuj Goswami, Amanpreet Singh, Casey A Fitzpatrick, Peter Bull, Greg Lipstein, Tony Nelli, Ron Zhu, et al. 2021. The hateful memes challenge: Competition report. In *NeurIPS 2020 Competition and Demonstration Track*, pages 344–360. PMLR.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The Stack: 3 TB of permissively licensed source code. *arXiv preprint arXiv:2211.15533*.

Andreas Köpf, Yannic Kilcher, Dimitri von Rütte, Sotiris Anagnostidis, Zhi-Rui Tam, Keith Stevens, Abdullah Barhoum, Nguyen Minh Duc, Oliver Stanley, Richárd Nagyfi, et al. 2023. OpenAssistant Conversations–Democratizing Large Language Model Alignment. *arXiv preprint arXiv:2304.07327*.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

Hugo Laurençon, Lucile Saulnier, Thomas Wang, Christopher Akiki, Albert Villanova del Moral, Teven Le Scao, Leandro Von Werra, Chenghao Mou, Eduardo González Ponferrada, Huu Nguyen, et al. 2022. The bigscience roots corpus: A 1.6 tb composite multilingual dataset. *Advances in Neural Information Processing Systems*, 35:31809–31826.

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O Stanley. 2022. Evolution through large models. *arXiv preprint arXiv:2206.08896*.

Bo Li, Yuanhan Zhang, Liangyu Chen, Jinghao Wang, Jingkang Yang, and Ziwei Liu. 2023a. Otter: A multi-modal model with in-context instruction tuning. *arXiv preprint arXiv:2305.03726*.

Hongyu Li, Seohyun Kim, and Satish Chandra. 2019. Neural code search evaluation dataset. *arXiv preprint arXiv:1908.09804*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023b. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Xueyang Li, Shangqing Liu, Ruitao Feng, Guozhu Meng, Xiaofei Xie, Kai Chen, and Yang Liu. 2022a. Transrepair: Context-aware program repair for compilation errors. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022b. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multilingual program repair benchmark set based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, pages 55–56.

Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023a. Visual instruction tuning. *arXiv preprint arXiv:2304.08485*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023b. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023c. Lost in the Middle: How Language Models Use Long Contexts. *arXiv preprint arXiv:2307.03172*.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2023d. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. *arXiv preprint arXiv:2306.03091*.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. 2023e. Gpteval: Nlg evaluation using gpt-4 with better human alignment. *arXiv preprint arXiv:2303.16634*.

Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V Le, Barret Zoph, Jason Wei, et al. 2023a. The Flan Collection: Designing Data and Methods for Effective Instruction Tuning. *arXiv preprint arXiv:2301.13688*.

Shayne Longpre, Gregory Yauney, Emily Reif, Katherine Lee, Adam Roberts, Barret Zoph, Denny Zhou, Jason Wei, Kevin Robinson, David Mimno, et al. 2023b. A Pretrainer's Guide to Training Data: Measuring the Effects of Data Age, Domain Coverage, Quality, & Toxicity. *arXiv preprint arXiv:2305.13169*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *arXiv preprint arXiv:2306.08568*.

Aman Madaan, Alexander Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. 2023a. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023b. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*.

Sewon Min, Mike Lewis, Luke Zettlemoyer, and Hannaneh Hajishirzi. 2022. MetaICL: Learning to learn in context. *Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

Martin Monperrus, Matias Martinez, He Ye, Fernanda Madeiral, Thomas Durieux, and Zhongxing Yu. 2021. Megadiff: A dataset of 600k java source code changes categorized by diff size. *arXiv preprint arXiv:2108.04631*.

Niklas Muennighoff. 2022. SGPT: GPT Sentence Embeddings for Semantic Search. *arXiv preprint arXiv:2202.08904*.

Niklas Muennighoff, Alexander M Rush, Boaz Barak, Teven Le Scao, Aleksandra Piktus, Nouamane Tazi, Sampo Pyysalo, Thomas Wolf, and Colin Raffel. 2023. Scaling Data-Constrained Language Models. *arXiv preprint arXiv:2305.16264*.

Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. 2022a. MTEB: Massive Text Embedding Benchmark. *arXiv preprint arXiv:2210.07316*.

Niklas Muennighoff, Thomas Wang, Lintang Sutawika, Adam Roberts, Stella Biderman, Teven Le Scao, M Saiful Bari, Sheng Shen, Zheng-Xin Yong, Hailey Schoelkopf, et al. 2022b. Crosslingual generalization through multitask finetuning. *arXiv preprint arXiv:2211.01786*.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128. PMLR.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Code-Gen2: Lessons for Training LLMs on Programming and Natural Languages. *arXiv preprint arXiv:2305.02309*.

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 arXiv:2203.13474*.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. 2021. Show Your Work: Scratchpads for Intermediate Computation with Language Models. *arXiv preprint arXiv:2112.00114*.

OpenAI. 2023. GPT-4 Technical Report.

Gabriel Orlanski, Kefan Xiao, Xavier Garcia, Jeffrey Hui, Joshua Howland, Jonathan Malmaud, Jacob Austin, Rishah Singh, and Michele Catasta. 2023. Measuring The Impact Of Programming Language Distribution. *arXiv preprint arXiv:2302.01973*.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. In *Conference on Neural Information Processing Systems (NeurIPS)*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, et al. 2023. RWKV: Reinventing RNNs for the Transformer Era. *arXiv preprint arXiv:2305.13048*.

Ethan Perez, Douwe Kiela, and Kyunghyun Cho. 2021. True few-shot learning with language models. *Advances in Neural Information Processing Systems*, 34:11054–11070.

Luiza Amador Pozzobon, Beyza Ermis, Patrick Lewis, and Sara Hooker. 2023. On the Challenges of Using Black-Box APIs for Toxicity Evaluation in Research. In *ICLR 2023 Workshop on Trustworthy and Reliable Large-Scale Machine Learning Models*.

Julian Aron Prenner and Romain Robbes. 2021. Automatic Program Repair with OpenAI's Codex: Evaluating QuixBugs. *arXiv preprint arXiv:2111.03922*.

Julian Aron Prenner and Romain Robbes. 2023. RunBugRun–An Executable Dataset for Automated Program Repair. *arXiv preprint arXiv:2304.01102*.

Ofir Press, Noah A Smith, and Mike Lewis. 2021. Train short, test long: Attention with linear biases enables input length extrapolation. *arXiv preprint arXiv:2108.12409*.

Vipul Raheja, Dhruv Kumar, Ryan Koo, and Dongyeop Kang. 2023. CoEdIT: Text Editing by Task-Specific Instruction Tuning. *arXiv preprint arXiv:2305.09857*.

Machel Reid and Graham Neubig. 2022. Learning to Model Editing Processes. *arXiv preprint arXiv:2205.12374*.

Ehud Reiter. 2018. A structured review of the validity of BLEU. *Computational Linguistics*, 44(3):393–401.

Victor Sanh, Albert Webson, Colin Raffel, Stephen Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, et al. 2022. Multitask Prompted Training Enables Zero-Shot Task Generalization. *International Conference on Learning Representations (ICLR)*.

Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. 2022a. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*.

Teven Le Scao, Thomas Wang, Daniel Hesslow, Lucile Saulnier, Stas Bekman, M Saiful Bari, Stella Bideman, Hady Elsahar, Niklas Muennighoff, Jason Phang, et al. 2022b. What Language Model to Train if You Have One Million GPU Hours? *arXiv preprint arXiv:2210.15424*.

Timo Schick, Jane Dwivedi-Yu, Zhengbao Jiang, Fabio Petroni, Patrick Lewis, Gautier Izacard, Qingfei You, Christoforos Nalmpantis, Edouard Grave, and Sebastian Riedel. 2022. PEER: A Collaborative Language Model. *arXiv preprint arXiv:2208.11663*.

Natalie Schluter. 2017. The limits of automatic summarisation according to rouge. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, pages 41–45. Association for Computational Linguistics.

Noam M. Shazeer. 2019. Fast Transformer Decoding: One Write-Head is All You Need. *arXiv preprint arXiv:1911.02150*.

Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, Yuenan Guo, and Qianxiang Wang. 2023. PanGu-Coder2: Boosting Large Language Models for Code with Ranking Feedback.

Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the evaluation of neural code summarization. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1597–1608.

Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. 2023a. RepoFusion: Training Code Models to Understand Your Repository. *arXiv preprint arXiv:2306.10998*.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023b. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*, pages 31693–31715. PMLR.

Marta Skreta, Naruki Yoshikawa, Sebastian Arellano-Rubach, Zhi Ji, Lasse Bjørn Kristensen, Kourosh Darvish, Alán Aspuru-Guzik, Florian Shkurti, and Animesh Garg. 2023. Errors are Useful Prompts: Instruction Guided Task Programming with Verifier-Assisted Iterative Prompting. *arXiv preprint arXiv:2303.14100*.

Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. *arXiv preprint arXiv:2301.08653*.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. 2022. Beyond the Imitation Game: Quantifying and extrapolating the capabilities of language models.

Simeng Sun, Kalpesh Krishna, Andrew Mattarella-Micke, and Mohit Iyyer. 2021. Do Long-Range Language Models Actually Use Long-Range Context? *ArXiv*, abs/2109.09115.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. `https://github.com/tatsu-lab/stanford_alpaca`.

Ross Taylor, Marcin Kardas, Guillem Cucurull, Thomas Scialom, Anthony Hartshorn, Elvis Saravia, Andrew Poulton, Viktor Kerkez, and Robert Stojnic. 2022. Galactica: A large language model for science. *arXiv preprint arXiv:2211.09085*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Lewis Tunstall, Nathan Lambert, Nazneen Rajani, Edward Beeching, Teven Le Scao, Leandro von Werra, Sheon Han, Philipp Schmid, and Alexander Rush. 2023. Creating a Coding Assistant with StarCoder. *Hugging Face Blog*. Https://huggingface.co/blog/starchat.

Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. 2019. SuperGLUE: A stickier benchmark for general-purpose language understanding systems. *Conference on Neural Information Processing Systems (NeurIPS)*.

Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023a. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *International Conference on Learning Representations (ICLR)*.

Yizhong Wang, Hamish Ivison, Pradeep Dasigi, Jack Hessel, Tushar Khot, Khyathi Raghavi Chandu, David Wadden, Kelsey MacMillan, Noah A Smith, Iz Beltagy, et al. 2023b. How Far Can Camels Go? Exploring the State of Instruction Tuning on Open Resources. *arXiv preprint arXiv:2306.04751*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022a. Self-instruct: Aligning language model with self generated instructions. *arXiv preprint arXiv:2212.10560*.

Yizhong Wang, Swaroop Mishra, Pegah Alipoormolabashi, Yeganeh Kordi, Amirreza Mirzaei, Anjana Arunkumar, Arjun Ashok, Arut Selvan Dhanasekaran, Atharva Naik, David Stap, et al. 2022b. Super-naturalinstructions: Generalization via declarative instructions on 1600+ nlp tasks. *arXiv preprint arXiv:2204.07705*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023c. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Zhiruo Wang, Grace Cuenca, Shuyan Zhou, Frank F Xu, and Graham Neubig. 2022c. Mconala: a benchmark for code generation from multiple natural languages. *arXiv preprint arXiv:2203.08388*.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022d. Execution-based evaluation for open-domain code generation. *arXiv preprint arXiv:2212.10481*.

Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2022. Finetuned language models are zero-shot learners. *International Conference on Learning Representations (ICLR)*.

Jiayi Wei, Greg Durrett, and Isil Dillig. 2023. Coeditor: Leveraging Contextual Changes for Multi-round Code Auto-editing. *arXiv preprint arXiv:2305.18584*.

Minghao Wu and Alham Fikri Aji. 2023. Style Over Substance: Evaluation Biases for Large Language Models. *arXiv preprint arXiv:2307.03025*.

Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair. *arXiv preprint arXiv:2301.13246*.

Mengzhou Xia, Mikel Artetxe, Chunting Zhou, Xi Victoria Lin, Ramakanth Pasunuru, Danqi Chen, Luke Zettlemoyer, and Ves Stoyanov. 2022. Training Trajectories of Language Models Across Scales. *arXiv preprint arXiv:2212.09803*.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023a. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244*.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022a. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10.

Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, and Hanghang Tong. 2022b. Combining Code Context and Fine-grained Code Difference for Commit Message Generation. In *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, pages 242–251.

Zhiyang Xu, Ying Shen, and Lifu Huang. 2023b. MultiInstruct: Improving Multi-Modal Zero-Shot Learning via Instruction Tuning.

Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. In *International Conference on Machine Learning*, pages 11941–11952. PMLR.

He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. A comprehensive study of automatic program repair on the QuixBugs benchmark. *Journal of Systems and Software*, 171:110825.

Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. Assessing the quality of GitHub copilot's code generation. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 62–71.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow. In *International Conference on Mining Software Repositories*, MSR, pages 476–486. ACM.

Zheng-Xin Yong, Hailey Schoelkopf, Niklas Muennighoff, Alham Fikri Aji, David Ifeoluwa Adelani, Khalid Almubarak, M Saiful Bari, Lintang Sutawika, Jungo Kasai, Ahmed Baruwa, et al. 2022. BLOOM+ 1: Adding Language Support to BLOOM for Zero-Shot Prompting. *arXiv preprint arXiv:2212.09535*.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Tao Xie, and Qianxiang Wang. 2023. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. *arXiv preprint arXiv:2302.00288*.

Yan Zeng, Hanbo Zhang, Jiani Zheng, Jiangnan Xia, Guoqiang Wei, Yang Wei, Yuchen Zhang, and Tao Kong. 2023. What Matters in Training a GPT4-Style Language Model with Multimodal Inputs? *arXiv preprint arXiv:2307.02469*.

Chunyan Zhang, Junchao Wang, Qinglei Zhou, Ting Xu, Ke Tang, Hairen Gui, and Fudong Liu. 2022a. A survey of automatic source code summarization. *Symmetry*, 14(3):471.

Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023a. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*.

Hang Zhang, Xin Li, and Lidong Bing. 2023b. Video-llama: An instruction-tuned audio-visual language model for video understanding. *arXiv preprint arXiv:2306.02858*.

Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2022b. Repairing Bugs in Python Assignments Using Large Language Models. *arXiv preprint arXiv:2209.14876*.

Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2022c. CoditT5: Pretraining for Source Code and Natural Language Editing. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12.

Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida Wang. 2023c. Coder reviewer reranking for code generation. In *International Conference on Machine Learning*, pages 41832–41846. PMLR.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*.

Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, et al. 2023a. Lima: Less is more for alignment. *arXiv preprint arXiv:2305.11206*.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023b. Codebertscore: Evaluating code generation with pretrained models of code. *arXiv preprint arXiv:2302.05527*.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910*.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. 2022. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474*.

Terry Yue Zhuo. 2023. Large Language Models Are State-of-the-Art Evaluators of Code Generation. *arXiv preprint arXiv:2304.14317*.

Appendix

Contents

A Contributions

Niklas Muennighoff created COMMITPACK and HUMANEVALPACK, wrote most of the paper and led the project. Qian Liu devised many quality filters, ran SantaCoder ablations, investigated early training decisions and helped edit the paper. Armel Zebaze created the Self-Instruct data and ran numerous ablations. Niklas Muennighoff, Armel Zebaze and Qinkai Zheng created and evaluated OCTOCODER and OCTOGEEX. Binyuan Hui pretrained SantaCoder, made major contributions to the presentation and helped edit the paper. Terry Yue Zhuo ran GPT-4 evaluations and helped edit the paper. Xiangru Tang provided help on several experiments for evaluation and helped edit the paper. Leandro von Werra provided early guidance, suggested many quality filters and added the commit data to StarCoder pretraining. Niklas Muennighoff, Qian Liu, Binyuan Hui, Swayam Singh and Shayne Longpre conducted the data analysis. Shayne Longpre advised the project and made large contributions to the paper.

B   Artifacts

Table 3: **Used and produced artifacts.**

Model	Public Link
Other models	
Diff Codegen 2B (Bradley et al., 2023)	`https://hf.co/CarperAI/diff-codegen-2b-v2`
InstructCodeT5+ (Wang et al., 2023c)	`https://hf.co/Salesforce/instructcodet5p-16b`
BLOOMZ (Muennighoff et al., 2022b)	`https://hf.co/bigscience/bloomz`
StarChat-β (Tunstall et al., 2023)	`https://hf.co/HuggingFaceH4/starchat-beta`
CodeGeeX2 (Zheng et al., 2023)	`https://github.com/THUDM/CodeGeeX2`
SantaCoder (Allal et al., 2023)	`https://hf.co/bigcode/santacoder`
StarCoder (Li et al., 2023b)	`https://hf.co/bigcode/starcoder`
WizardCoder (Luo et al., 2023)	`https://hf.co/WizardLM/WizardCoder-15B-V1.0`
GPT-4 (OpenAI, 2023)	`https://openai.com/gpt-4`
Data Ablations (Appendix J) - Data	
Filtered xP3x code	REDACTED FOR REVIEW
StarCoder Self-Instruct	REDACTED FOR REVIEW
Filtered OASST	REDACTED FOR REVIEW
Manual selection (Appendix J)	REDACTED FOR REVIEW
Data Ablations (Appendix J) - Data	
Filtered xP3x code	`https://hf.co/datasets/bigcode/xp3x-octopack`
StarCoder Self-Instruct	`https://hf.co/datasets/codeparrot/self-instruct-starcoder`
Filtered OASST	`https://hf.co/datasets/bigcode/oasst-octopack`
Manual selection (Appendix J)	`https://hf.co/datasets/bigcode/co-manual`
Data Ablations (Appendix J) - Models	
Self-Instruct (SI)	`https://hf.co/bigcode/starcoder-s`
OASST (O)	`https://hf.co/bigcode/starcoder-o`
SI + O	`https://hf.co/bigcode/starcoder-so`
xP3x + O	`https://hf.co/bigcode/starcoder-xo`
COMMITPACKFT + O (Formatting)	`https://hf.co/bigcode/starcoder-co-format`
COMMITPACKFT + O (Target loss)	`https://hf.co/bigcode/starcoder-co-target`
COMMITPACKFT + O (Manual)	`https://hf.co/bigcode/starcoder-co-manual`
COMMITPACKFT + xP3x + O	`https://hf.co/bigcode/starcoder-cxo`
COMMITPACKFT + xP3x + SI + O	`https://hf.co/bigcode/starcoder-cxso`
SantaCoder ablations (Appendix G, Appendix H)	
Commit format Pretraining	`https://hf.co/bigcode/santacoderpack`
Commit format Finetuning	`https://hf.co/bigcode/santacoder-cf`
Line diff format Finetuning	`https://hf.co/bigcode/santacoder-ldf`
Other datasets	
COMMITPACK Metadata	`https://hf.co/datasets/bigcode/commitpackmeta`
Main artifacts	
COMMITPACK	`https://hf.co/datasets/bigcode/commitpack`
COMMITPACKFT	`https://hf.co/datasets/bigcode/commitpackft`
HUMANEVALPACK	`https://hf.co/datasets/bigcode/humanevalpack`
OCTOGEEX	`https://hf.co/bigcode/octogeex`
OCTOCODER	`https://hf.co/bigcode/octocoder`

C COMMITPACK and COMMITPACKFT Languages

Table 4: **Programming language distribution of COMMITPACK and COMMITPACKFT.** Shortcuts: MB=Megabytes, owl=web-ontology-language, pir=parrot-internal-representation, dcl=digital-command-language, mms=module-management-system, gf=grammatical-framework

Language (↓)	COMMITPACK			COMMITPACKFT		
	MB	Samples	% (MB)	MB	Samples	% (MB)
Total	3709175.78	57700105	100.0	1545.02	702062	100.0
json	583293.82	3495038	15.73	86.74	39777	5.61
xml	279208.68	1923159	7.53	23.68	9337	1.53
text	270662.6	1389525	7.3	66.66	46588	4.31
javascript	262824.84	5401937	7.09	125.01	52989	8.09
objective-c++	239009.3	32227	6.44	0.38	86	0.02
python	234311.56	6189601	6.32	132.68	56025	8.59
c	200876.8	2779478	5.42	21.08	8506	1.36
c++	186585.26	2402294	5.03	14.14	4992	0.92
markdown	171849.95	7645354	4.63	131.15	62518	8.49
java	127103.45	3744377	3.43	56.28	20635	3.64
html	105305.28	2366841	2.84	48.42	20214	3.13
yaml	100466.64	2592787	2.71	190.88	114320	12.35
go	86444.62	1183612	2.33	12.13	5004	0.79
csv	82946.19	79268	2.24	0.53	375	0.03
php	74961.64	2555419	2.02	60.22	24791	3.9
jupyter-notebook	66854.08	94000	1.8	0.1	48	0.01
gettext-catalog	62296.88	168327	1.68	0.13	72	0.01
sql	56802.76	132772	1.53	3.74	2069	0.24
unity3d-asset	39535.01	17867	1.07	0.16	101	0.01
typescript	39254.8	572136	1.06	14.28	5868	0.92
owl	36435.46	7458	0.98	0	0	0.0
ruby	35830.74	2928702	0.97	195.29	69413	12.64
c#	33669.65	923157	0.91	26.84	9346	1.74
nix	33547.92	221281	0.9	3.84	1593	0.25
shell	25109.95	1017977	0.68	66.86	31217	4.33
perl	21148.93	374266	0.57	4.99	2288	0.32
tex	17471.11	89283	0.47	0.56	307	0.04
css	16306.63	548818	0.44	9.36	5049	0.61
restructuredtext	15613.89	494037	0.42	15.73	6560	1.02
rust	15011.3	296214	0.4	7.24	2996	0.47
groff	12020.19	32923	0.32	0.4	192	0.03
ini	8375.16	297100	0.23	21.04	11360	1.36
scala	8325.96	316064	0.22	11.18	5040	0.72
coffeescript	6795.14	292446	0.18	16.96	5513	1.1
haskell	6306.12	217325	0.17	3.31	1389	0.21
swift	5902.72	319289	0.16	16.27	4849	1.05
lua	5763.12	139091	0.16	1.85	920	0.12
svg	5645.44	27095	0.15	0.25	169	0.02
gas	5585.38	15121	0.15	0.34	193	0.02
ocaml	5355.4	81360	0.14	0.7	333	0.05
erlang	5043.32	93685	0.14	1.19	480	0.08
makefile	4238.51	343379	0.11	2.53	960	0.16
asciidoc	4138.59	96671	0.11	1.86	523	0.12
emacs-lisp	3988.65	83228	0.11	1.97	1015	0.13
scss	3944.94	288190	0.11	13.21	6829	0.86
clojure	3523.41	158674	0.09	5.07	2403	0.33
org	3126.22	30198	0.08	0.27	136	0.02
common-lisp	2954.9	74628	0.08	1.45	778	0.09

diff	2586.05	21021	0.07	1.48	680	0.1
groovy	2569.14	110057	0.07	4.17	1486	0.27
html+erb	2450.68	225379	0.07	23.1	10910	1.5
nesc	2439.56	473	0.07	0.02	7	0.0
dart	2395.8	56873	0.06	1.96	765	0.13
powershell	2289.28	55381	0.06	2.06	991	0.13
f#	2289.24	66840	0.06	0.66	254	0.04
dm	2223.14	55584	0.06	0.15	16	0.01
kotlin	2219.25	124266	0.06	5.37	2214	0.35
pascal	2194.68	42511	0.06	0.05	25	0.0
jsx	2124.74	139148	0.06	5.5	2199	0.36
viml	1948.21	74062	0.05	1.96	1063	0.13
actionscript	1844.15	28819	0.05	0.12	49	0.01
cython	1736.59	25927	0.05	0.31	123	0.02
turtle	1698.95	3882	0.05	0.05	21	0.0
less	1616.56	88634	0.04	3.72	1360	0.24
mathematica	1475.04	925	0.04	0.01	1	0.0
xslt	1441.46	27956	0.04	0.26	99	0.02
scheme	1249.24	30546	0.03	0.42	213	0.03
perl6	1223.16	12167	0.03	0.27	122	0.02
edn	1186.94	2289	0.03	0.09	48	0.01
fortran	1178.55	13463	0.03	0.14	70	0.01
java-server-pages	1173.07	53574	0.03	0.45	173	0.03
standard-ml	1133.48	20097	0.03	0.15	72	0.01
cmake	1132.07	58446	0.03	2.27	981	0.15
json5	1108.2	1827	0.03	0.08	33	0.01
vala	1104.51	14822	0.03	0.12	50	0.01
vue	1093.8	68967	0.03	1.38	587	0.09
freemarker	1032.33	36216	0.03	1.03	510	0.07
graphql	1004.84	2009	0.03	0.03	17	0.0
twig	958.96	39588	0.03	3.96	1610	0.26
tcl	869.83	16407	0.02	0.29	103	0.02
pod	859.02	14922	0.02	0.15	54	0.01
dockerfile	849.73	259379	0.02	0.1	39	0.01
yacc	845.7	8230	0.02	0.01	3	0.0
postscript	800.73	903	0.02	0.02	9	0.0
racket	796.64	16615	0.02	0.2	117	0.01
eagle	785.68	2237	0.02	0.01	4	0.0
haxe	772.9	28447	0.02	0.34	174	0.02
julia	752.07	22695	0.02	0.31	180	0.02
handlebars	740.82	49842	0.02	3.29	1429	0.21
smarty	720.94	41065	0.02	1.59	737	0.1
visual-basic	681.52	10511	0.02	0.15	48	0.01
literate-haskell	673.74	10729	0.02	0.02	7	0.0
smalltalk	665.89	11741	0.02	0.46	284	0.03
isabelle	655.82	8359	0.02	0.01	2	0.0
nimrod	652.86	12023	0.02	0.24	67	0.02
zig	621.38	4290	0.02	0.01	4	0.0
m4	603.58	12465	0.02	0.26	101	0.02
max	603.56	2259	0.02	0	0	0.0
elixir	558.12	35473	0.02	2.35	1150	0.15
mako	543.01	8943	0.01	0.76	170	0.05
arduino	534.18	32350	0.01	0.46	225	0.03
jade	531.4	46993	0.01	2.35	1119	0.15
haml	502.01	74792	0.01	10.74	4415	0.7
elm	481.97	18542	0.01	0.62	265	0.04
purebasic	474.28	36	0.01	0.02	5	0.0
coldfusion	470.78	9263	0.01	0.02	9	0.0
lean	470.03	7507	0.01	0.02	3	0.0

r	454.32	12858	0.01	0.23	121	0.01
cuda	437.67	11450	0.01	0.07	25	0.0
textile	425.12	18491	0.01	0.18	61	0.01
robotframework	421.61	9211	0.01	0.21	85	0.01
abap	409.62	1955	0.01	0.01	1	0.0
rdoc	397.03	38760	0.01	0.55	270	0.04
llvm	382.2	10727	0.01	1.6	780	0.1
ada	380.7	13258	0.01	0.73	265	0.05
batchfile	372.16	43674	0.01	2.98	1466	0.19
qml	361.45	19360	0.01	0.94	368	0.06
jasmin	359.82	4782	0.01	0.05	9	0.0
assembly	343.62	8126	0.01	0.17	105	0.01
g-code	334.96	3690	0.01	0.04	7	0.0
cucumber	331.38	26677	0.01	2.59	976	0.17
html+php	323.35	18381	0.01	0.33	150	0.02
kicad	321.94	759	0.01	0	0	0.0
api-blueprint	317.85	4765	0.01	0.06	23	0.0
eiffel	311.48	373	0.01	0.01	2	0.0
toml	292.68	63517	0.01	5.58	3424	0.36
modelica	284.62	2611	0.01	0.04	15	0.0
bitbake	277.58	43239	0.01	4.46	1308	0.29
lex	275.96	705	0.01	0	0	0.0
stylus	273.06	21967	0.01	0.95	480	0.06
protocol-buffer	254.12	9202	0.01	0.52	181	0.03
unknown	252.23	30570	0.01	3.05	1597	0.2
nit	244.54	4951	0.01	0.02	3	0.0
factor	241.19	15378	0.01	0.36	113	0.02
xs	239.04	3215	0.01	0.02	7	0.0
sass	230.65	23144	0.01	1.36	705	0.09
pir	230.2	6231	0.01	0.08	23	0.01
html+django	217.04	10535	0.01	0.85	399	0.06
mediawiki	214.32	10188	0.01	0.08	33	0.01
logos	212.3	1733	0.01	0.04	19	0.0
genshi	209.3	956	0.01	0.02	3	0.0
coldfusion-cfc	208.16	4410	0.01	0.05	20	0.0
xtend	179.54	7775	0.0	0.13	55	0.01
sqf	168.66	7778	0.0	0.09	45	0.01
vhdl	155.95	2185	0.0	0.02	5	0.0
antlr	143.55	3651	0.0	0.03	15	0.0
systemverilog	140.19	3944	0.0	0.08	35	0.01
hcl	136.75	13379	0.0	0.91	421	0.06
asp	136.1	4286	0.0	0.09	22	0.01
nsis	129.12	4048	0.0	0.06	15	0.0
inform-7	120.19	184	0.0	0.01	2	0.0
slim	119.04	18726	0.0	2.06	1052	0.13
groovy-server-pages	117.37	6695	0.0	0.07	25	0.0
ceylon	116.14	7256	0.0	0.1	49	0.01
fish	111.28	15351	0.0	1.33	813	0.09
processing	108.58	5912	0.0	0.07	35	0.0
component-pascal	105.5	43	0.0	0	0	0.0
lasso	104.17	67	0.0	0	0	0.0
glsl	99.49	9478	0.0	0.34	164	0.02
saltstack	98.2	12314	0.0	1.41	617	0.09
xbase	94.42	1670	0.0	0.01	3	0.0
autohotkey	94.22	1452	0.0	0.02	15	0.0
liquid	93.79	2651	0.0	0.09	30	0.01
purescript	92.41	5024	0.0	0.17	80	0.01
agda	92.06	4956	0.0	0.02	10	0.0
inno-setup	91.36	3014	0.0	0.06	16	0.0

oz	90.48	1551	0.0	0.03	8	0.0
chapel	89.62	26447	0.0	0.04	20	0.0
arc	87.21	758	0.0	0.01	2	0.0
opencl	86.43	2489	0.0	0.05	23	0.0
graphviz-dot	85.8	1525	0.0	0.07	35	0.0
pawn	85.42	580	0.0	0.01	3	0.0
jsoniq	75.15	1343	0.0	0.01	6	0.0
bluespec	72.38	2500	0.0	0.01	2	0.0
smali	71.38	174	0.0	0	0	0.0
krl	69.87	1879	0.0	0.02	4	0.0
maple	68.28	1311	0.0	0.01	2	0.0
unrealscript	67.67	585	0.0	0.01	1	0.0
ooc	63.19	3416	0.0	0.04	15	0.0
pure-data	62.62	603	0.0	0.01	1	0.0
xquery	61.96	2237	0.0	0.08	39	0.01
dcl	59.64	833	0.0	0.04	19	0.0
moonscript	59.21	1951	0.0	0.02	10	0.0
awk	57.18	2206	0.0	0.1	52	0.01
pike	52.87	1262	0.0	0.02	6	0.0
livescript	51.23	5194	0.0	0.13	63	0.01
solidity	50.86	3689	0.0	0.08	37	0.01
monkey	48.26	1367	0.0	0.02	4	0.0
jsonld	48.01	462	0.0	0.02	6	0.0
zephir	42.68	1265	0.0	0.02	4	0.0
crystal	41.92	4217	0.0	0.35	182	0.02
rhtml	41.02	4551	0.0	0.35	135	0.02
stata	40.68	1344	0.0	0.02	10	0.0
idris	39.9	3025	0.0	0.13	38	0.01
raml	39.39	948	0.0	0.03	9	0.0
openscad	37.73	2178	0.0	0.05	21	0.0
red	35.26	1108	0.0	0.01	1	0.0
c2hs-haskell	34.47	1021	0.0	0.01	2	0.0
cycript	33.96	197	0.0	0	0	0.0
applescript	33.51	1304	0.0	0.04	19	0.0
mupad	32.49	178	0.0	0.02	4	0.0
literate-agda	31.38	567	0.0	0.01	1	0.0
boo	31.17	26289	0.0	0.01	2	0.0
sourcepawn	29.53	717	0.0	0.01	3	0.0
qmake	29.51	3632	0.0	0.32	140	0.02
ragel-in-ruby-host	28.3	888	0.0	0.01	4	0.0
io	27.95	1247	0.0	0.01	4	0.0
desktop	27.65	5021	0.0	0.36	186	0.02
propeller-spin	26.77	625	0.0	0.01	1	0.0
thrift	26.75	1007	0.0	0.08	28	0.01
volt	25.05	1660	0.0	0.02	9	0.0
xproc	24.21	914	0.0	0.02	3	0.0
igor-pro	23.75	388	0.0	0.01	1	0.0
lolcode	23.74	24861	0.0	0	0	0.0
html+eex	21.41	2100	0.0	0.29	135	0.02
logtalk	20.43	1035	0.0	0.06	21	0.0
mirah	20.1	706	0.0	0.04	16	0.0
gnuplot	19.68	889	0.0	0.03	17	0.0
literate-coffeescript	19.02	1041	0.0	0.05	19	0.0
jflex	18.61	555	0.0	0.01	1	0.0
emberscript	18.39	1024	0.0	0.02	7	0.0
cobol	17.0	24953	0.0	0	0	0.0
yang	16.94	597	0.0	0.02	6	0.0
rebol	16.47	239	0.0	0.01	3	0.0
linker-script	16.08	1604	0.0	0.08	37	0.01

cartocss	15.92	555	0.0	0.01	3	0.0
urweb	13.07	304	0.0	0.02	6	0.0
rmarkdown	13.03	750	0.0	0	0	0.0
darcs-patch	13.01	80	0.0	0	0	0.0
csound	12.85	229	0.0	0.01	4	0.0
squirrel	12.84	531	0.0	0.01	4	0.0
apl	12.56	586	0.0	0.02	7	0.0
hlsl	12.17	1529	0.0	0.03	11	0.0
latte	11.89	1380	0.0	0.02	7	0.0
pony	11.84	624	0.0	0.05	16	0.0
ioke	10.86	373	0.0	0.04	25	0.0
hy	10.51	879	0.0	0.04	12	0.0
uno	10.36	628	0.0	0.01	2	0.0
pan	10.34	637	0.0	0.05	23	0.0
xojo	10.31	642	0.0	0	0	0.0
papyrus	10.26	130	0.0	0	0	0.0
stan	10.25	540	0.0	0	0	0.0
slash	9.9	640	0.0	0.01	4	0.0
supercollider	9.8	318	0.0	0.01	2	0.0
vcl	9.46	747	0.0	0.04	18	0.0
smt	9.03	117	0.0	0.01	3	0.0
glyph	8.95	7	0.0	0	0	0.0
wisp	8.74	262	0.0	0.01	3	0.0
renpy	8.3	421	0.0	0.02	3	0.0
clips	7.73	450	0.0	0	0	0.0
dns-zone	7.56	54	0.0	0.01	2	0.0
sas	7.54	269	0.0	0.01	1	0.0
rouge	7.2	396	0.0	0.1	41	0.01
ec	7.03	94	0.0	0	0	0.0
dylan	6.82	280	0.0	0.01	2	0.0
tcsh	6.52	748	0.0	0.02	10	0.0
aspectj	6.33	451	0.0	0.02	8	0.0
netlogo	6.3	140	0.0	0	0	0.0
gap	6.1	46	0.0	0	0	0.0
fancy	5.95	675	0.0	0.02	8	0.0
coq	5.74	80	0.0	0	0	0.0
click	5.74	9	0.0	0	0	0.0
capn-proto	5.64	330	0.0	0.04	12	0.0
flux	5.57	47	0.0	0.01	3	0.0
forth	5.51	265	0.0	0.01	2	0.0
ats	5.42	383	0.0	0.01	3	0.0
netlinx	5.17	144	0.0	0.01	1	0.0
clean	5.07	171	0.0	0.01	1	0.0
parrot-assembly	4.66	227	0.0	0.01	2	0.0
alloy	4.64	203	0.0	0	0	0.0
lfe	4.58	287	0.0	0.02	6	0.0
gdscript	4.49	460	0.0	0.03	9	0.0
augeas	4.44	395	0.0	0.04	13	0.0
sparql	4.4	1036	0.0	0.04	23	0.0
lilypond	4.31	265	0.0	0.01	6	0.0
scilab	4.09	375	0.0	0.02	10	0.0
autoit	4.06	279	0.0	0	0	0.0
myghty	3.86	105	0.0	0	0	0.0
blitzmax	3.74	220	0.0	0.01	1	0.0
creole	3.42	337	0.0	0.01	2	0.0
harbour	3.34	107	0.0	0.01	1	0.0
piglatin	3.17	513	0.0	0.02	11	0.0
opa	3.16	211	0.0	0	0	0.0
sage	3.03	414	0.0	0.01	1	0.0

ston	2.85	414	0.0	0.01	6	0.0
maxscript	2.8	47	0.0	0	0	0.0
lsl	2.68	74	0.0	0.01	3	0.0
gentoo-ebuild	2.58	601	0.0	0.06	16	0.0
nu	2.38	170	0.0	0.01	2	0.0
bro	2.34	333	0.0	0.01	3	0.0
xc	2.02	88	0.0	0	0	0.0
j	1.81	142	0.0	0	0	0.0
metal	1.72	151	0.0	0.02	4	0.0
mms	1.54	91	0.0	0.01	1	0.0
webidl	1.51	96	0.0	0.05	6	0.0
tea	1.47	29	0.0	0	0	0.0
redcode	1.27	149	0.0	0	0	0.0
shen	1.2	71	0.0	0	0	0.0
pov-ray-sdl	1.14	104	0.0	0.01	5	0.0
x10	1.01	33	0.0	0	0	0.0
brainfuck	0.96	167	0.0	0.01	2	0.0
ninja	0.95	187	0.0	0.03	14	0.0
golo	0.9	115	0.0	0	0	0.0
webassembly	0.86	83	0.0	0	0	0.0
self	0.82	15	0.0	0	0	0.0
labview	0.81	61	0.0	0	0	0.0
octave	0.8	12	0.0	0	0	0.0
pogoscript	0.8	74	0.0	0	0	0.0
d	0.8	20	0.0	0	0	0.0
http	0.74	140	0.0	0.03	19	0.0
ecl	0.66	48	0.0	0.01	4	0.0
chuck	0.58	99	0.0	0	0	0.0
gosu	0.52	60	0.0	0	0	0.0
parrot	0.52	17	0.0	0	0	0.0
opal	0.47	69	0.0	0	0	0.0
objective-j	0.46	37	0.0	0	0	0.0
kit	0.41	48	0.0	0	0	0.0
gams	0.38	18	0.0	0	0	0.0
prolog	0.28	35	0.0	0	0	0.0
clarion	0.27	13	0.0	0	0	0.0
mask	0.25	37	0.0	0.01	4	0.0
brightscript	0.24	28	0.0	0	0	0.0
scaml	0.18	31	0.0	0.01	1	0.0
matlab	0.16	29	0.0	0	0	0.0
idl	0.15	1	0.0	0	0	0.0
ags-script	0.12	31	0.0	0	0	0.0
lookml	0.12	10	0.0	0	0	0.0
apacheconf	0.11	59	0.0	0.01	2	0.0
oxygene	0.1	9	0.0	0	0	0.0
txl	0.1	3	0.0	0	0	0.0
gf	0.09	39	0.0	0	0	0.0
renderscript	0.06	54	0.0	0	0	0.0
mtml	0.05	13	0.0	0.01	2	0.0
unified-parallel-c	0.05	6	0.0	0	0	0.0
dogescript	0.04	10	0.0	0	0	0.0
gentoo-eclass	0.04	6	0.0	0	0	0.0
zimpl	0.04	7	0.0	0	0	0.0
irc-log	0.04	9	0.0	0	0	0.0
fantom	0.03	11	0.0	0	0	0.0
numpy	0.03	1	0.0	0	0	0.0
cirru	0.02	4	0.0	0	0	0.0
xpages	0.02	7	0.0	0.01	1	0.0
nginx	0.02	6	0.0	0.01	2	0.0

objdump	0.02	1	0.0	0	0	0.0
python-traceback	0.02	10	0.0	0	0	0.0
realbasic	0.01	1	0.0	0	0	0.0
befunge	0.01	2	0.0	0	0	0.0
bison	0.01	1	0.0	0	0	0.0
m	0.01	1	0.0	0	0	0.0
omgrofl	0.01	1	0.0	0	0	0.0

D   Dataset Creation

COMMITPACK   We use the GitHub archive available on GCP which contains metadata from GitHub commits up to 2016.[4] It contains around 3TB of GitHub activity data for more than 2.8 million GitHub repositories including more than 145 million unique commits, over 2 billion different file paths and the contents of the latest revision for 163 million files.[5] We apply the filters in Table 5 to this dataset. The resulting dataset containing only metadata is uploaded at `https://hf.co/datasets/bigcode/commitpackmeta`. As the activity dataset only contains commit ids without the actual code changes, we scrape the code from GitHub. We use the metadata and the GitHub API to scrape the changed file prior and after the respective commit. Some repositories referenced in the activity data are no longer accessible, thus we discard them. This results in COMMITPACK with approximately 4 terabytes uploaded at `https://hf.co/datasets/bigcode/commitpack`.

Table 5: **COMMITPACK filters.**

Description	Details
License	Only keep samples licensed as MIT, Artistic-2.0, ISC, CC0-1.0, EPL-1.0, MPL-2.0, Apache-2.0, BSD-3-Clause, AGPL-3.0, LGPL-2.1, BSD-2-Clause or without license.
Length	Only keep code where the commit message has at least 5 and at most 10,000 characters
Noise	Remove code where the lowercased commit message is any of 'add files via upload', "can't you see i'm updating the time?", 'commit', 'create readme.md', 'dummy', 'first commit', 'heartbeat update', 'initial commit', 'mirroring from micro.blog.', 'no message', 'pi push', 'readme', 'update', 'updates', 'update _config.yaml', 'update index.html', 'update readme.md', 'update readme', 'updated readme', 'update log', 'update data.js', 'update data.json', 'update data.js', 'pi push' or starts with 'merge'
Single file	Remove samples that contain changes across multiple files
Opt-out	Remove samples from repositories owned by users that opted out of The Stack (Kocetkov et al., 2022)

COMMITPACKFT   Prior work has shown the importance of careful data filtering to maintain quality (Yin et al., 2018; Dhole et al., 2021; Laurençon et al., 2022; Longpre et al., 2023b). To create a smaller version focused on commits that resemble high-quality instructions, we further filter COMMITPACK to create COMMITPACKFT using the steps outlined in Table 6. We also checked for any contamination with HumanEval (Chen et al., 2021) but did not find any solution or docstring present in COMMIT-PACKFT. This is likely because our commit data only goes up to 2016, which is several years prior to the release of HumanEval. Our filters reduce the dataset by a factor of around 1000 resulting in close to 2 gigabytes. To gain a deeper understanding of the rich content within COMMITPACKFT, we analyze commits on its Python subset (56K samples). We first collect the most prevalent commit domain by prompting GPT-4 with: `"I'd like to know the main types of commits on Github and aim to cover as comprehensively as possible."`. Subsequently, we use GPT-4 to classify each sample using the prompt in Figure 5. The task distribution is visualized in Figure 2.

---

[4]https://www.gharchive.org/

[5]https://github.blog/2016-06-29-making-open-source-data-more-available/

Table 6: **COMMITPACKFT filters applied to COMMITPACK.** With the commit message we refer to the commit message subject only, not the body.

Description	Details
Length	Remove samples where the before code has more than 50,000 characters
Length	Remove samples where the after code has 0 characters
Difference	Remove samples where the before and after code are the same (e.g. file name changes)
Difference	Remove samples that contain a hashtag (to avoid references to issues)
Extension	Remove samples where the filename of the code after has an atypical extension for the programming language (e.g. only keep '.py' for Python)
Filename	Remove samples where the filename is contained in the commit message (as we do not use the filename in finetuning)
Length	Only keep samples where the commit message has more than 10 and less than 1000 characters
Words	Only keep samples where the commit message can be split into more than 4 and less than 1000 space-separated words
Clean	Remove any appearances of '[skip ci]', '[ci skip]', sequences at the beginning or end that are in brackets, sequences at the beginning that end with ':' and strip whitespace at the beginning or end
Capitalized	Only keep samples where the message starts with an uppercase letter
Tokens	Only keep samples where the concatenation of the code before, a special token and the code after has at least 50 tokens and at most 768 tokens according to the StarCoder tokenizer
Instructions	Only keep samples where the lowercased commit message starts with any of the words in Table 7
Noise	Remove samples where the lowercased commit message contains any of 'auto commit', 'update contributing', '<?xml', 'merge branch', 'merge pull request', 'signed-off-by', "fix that bug where things didn't work but now they should", "put the thingie in the thingie", "add a beter commit message", "code review", "//codereview", "work in progress", "wip", "https://", "http://", "| leetcode", "cdpcp", " i ", "i've" , "i'm" or both "thanks to" and "for"
Regex	Remove samples where the lowercased commit message has a match for any of the regular expressions `(?:v)?\d+\.\d+\.\d+(?=$|\S)`, `^[a-f0-9]+(?:-[a-f0-9]+)*$`, `([a-f0-9]{40})`, `issue\s*\d+`, `bug\s*\d+` or `feature\s*\d+`
Downsample	With 90% probability remove samples where the commit message starts with "Bump", "Set version" or "Update version"

xP3x We use a subset of xP3x (Muennighoff et al., 2022b) focusing on code datasets consisting of APPS (Hendrycks et al., 2021), CodeContests (Li et al., 2022b), Jupyter Code Pairs,[6] MBPP (Austin et al., 2021), XLCoST (Zhu et al., 2022), Code Complex (Jeon et al., 2022), Docstring Corpus (Barone and Sennrich, 2017), Great Code (Hellendoorn et al., 2019) and State Changes.[7]

OASST We reuse a filtered variant of OASST (Köpf et al., 2023) from prior work (Dettmers et al., 2023) and apply additional filters to remove responses that refuse to comply with the user request. To compute the programming languages and code fraction for OASST depicted in Table 1, we count all responses containing e.g. ```python or ```py for the Python programming language. There are code samples that are not enclosed in backticks or do not specify the language, thus we are likely underestimating the actual fraction of code data for OASST in Table 1.

---

[6]https://hf.co/datasets/codeparrot/github-jupyter-text-code-pairs
[7]https://hf.co/datasets/Fraser/python-state-changes

Table 7: **Commit message starting words allowed in COMMITPACKFT.**

"abort', 'accelerate', 'access', 'accumulate', 'add', 'address', 'adjust', 'advance', 'align', 'al-lot', 'allow', 'amplify', 'annotate', 'append', 'apply', 'archive', 'arrange', 'attach', 'augment', 'automate', 'backup', 'boost', 'break', 'bring', 'brush up', 'build', 'bump', 'call', 'change', 'check', 'choose', 'clarify', 'clean', 'clear', 'clone', 'comment', 'complete', 'compress', 'con-catenate', 'configure', 'connect', 'consolidate', 'convert', 'copy', 'correct', 'cover', 'create', 'customize', 'cut', 'deal with', 'debug', 'decipher', 'declare', 'decommission', 'decomplexify', 'decompress', 'decrease', 'decrypt', 'define', 'delete', 'deploy', 'designate', 'destroy', 'detach', 'determine', 'develop', 'diminish', 'disable', 'discard', 'disentangle', 'dismantle', 'divide', 'document', 'downgrade', 'drop', 'duplicate', 'edit', 'embed', 'emphasize', 'enable', 'encrypt', 'enforce', 'enhance', 'enlarge', 'enumerate', 'eradicate', 'escalate', 'establish', 'exclude', 'exit', 'expand', 'expedite', 'expire', 'extend', 'facilitate', 'fix', 'format', 'gather', 'generalize', 'halt', 'handle', 'hasten', 'hide', 'implement', 'improve', 'include', 'increase', 'increment', 'indent', 'index', 'inflate', 'initialize', 'insert', 'install', 'integrate', 'interpolate', 'interrupt', 'introduce', 'isolate', 'join', 'kill', 'leverage', 'load', 'magnify', 'maintain', 'make', 'man-age', 'mark', 'mask', 'mend', 'merge', 'migrate', 'modify', 'monitor', 'move', 'multiply', 'normalize', 'optimize', 'orchestrate', 'order', 'package', 'paraphrase', 'paste', 'patch', 'plug ', 'prepare', 'prepend', 'print', 'provision', 'purge', 'put', 'quit', 'raise', 'read', 'reannotate', 'rearrange', 'rebase', 'reboot', 'rebuild', 'recomment', 'recompile', 'reconfigure', 'reconnect', 'rectify', 'redact', 'redefine', 'reduce', 'refactor', 'reformat', 'refresh', 'reimplement', 'rein-force', 'relocate', 'remove', 'rename', 'reorder', 'reorganize', 'repackage', 'repair', 'rephrase', 'replace', 'reposition', 'reschedule', 'reset', 'reshape', 'resolve', 'restructure', 'return', 'revert', 'revise', 'revoke', 'reword', 'rework', 'rewrite', 'rollback', 'save', 'scale', 'scrub', 'secure', 'select', 'send', 'set', 'settle', 'simplify', 'solve', 'sort', 'speed up', 'split', 'stabilize', 'standard-ize', 'stipulate', 'stop', 'store', 'streamline', 'strengthen', 'structure', 'substitute', 'subtract', 'support', 'swap', 'switch', 'synchronize', 'tackle', 'tag', 'terminate', 'test', 'throw', 'tidy', 'transform', 'transpose', 'trim', 'troubleshoot', 'truncate', 'tweak', 'unblock', 'uncover', 'undo', 'unify', 'uninstall', 'unplug', 'unpublish', 'unravel', 'unstage', 'unsync', 'untangle', 'unwind', 'update', 'upgrade', 'use', 'validate', 'verify', 'watch', 'watermark', 'whitelist', 'withdraw', 'work', 'write"

Please categorize the following commit message, which may fall into more than one category.

Category
Bug fixes, New features, Refactoring/code cleanup, Documentation, Testing, User interface, Dependencies, Configuration, Build system/tooling, Performance improvements, Formatting/Linting, Security, Technical debt repayment, Release management, Accessibility, Deprecation, Logging/In-strumentation, Internationalization

Commit Message
Add the blacklist checking to the bulk

Classification
Bug fixes, New features

Commit Message
{COMMIT_MESSAGE}
Classification

Figure 5: **GPT-4 1-shot prompt for classifying commits in COMMITPACKFT.**

E Comparing Data Before and After Filtering

In Table 8 we compare word statistics prior to and after filtering COMMITPACK to create COMMIT-PACKFT. The mean commit subject and message length increases suggesting that messages are more informative in COMMITPACKFT. The code lengths decrease significantly as we limit the number of allowed tokens in the filters in Table 6. Notably, the percentage of code changed between pre- and post-commit is $77.6/59.1 = 1.31$ (a 31% increase) as opposed to $3269.8/3269.9 = 1.007$ (a 0.7% increase). Thus, the filtered data carries significantly more signal per token with fewer repetitions of the code prior to the commit.

Table 8: **The effect of data filters on subject, message, and code lengths**. We compare differences in word statistics of COMMITPACK and COMMITPACKFT.

Metric	Before Filter	After Filter	Difference
Subject Length (words)	5.7±0.02	6.9±0.01	+1.28
Message Length (words)	8.7±0.06	9.9±0.05	+1.34
Pre-Commit Code Length (words)	3269.9±298.8	59.1±0.19	-3210.9
Post-Commit Code Length (words)	3269.8±299.5	77.6±0.23	-3214.2

F Comparing COMMITPACK and The Stack

In Table 9 we provide statistics on repositories and usernames of COMMITPACK and The Stack (Kocetkov et al., 2022). COMMITPACK contains a total of 1,934,255 repositories. Around half (49.3%) of them are also in The Stack. However, The Stack only provides the raw code files of these repositories from some fixed point in time. COMMITPACK contains the changes made to the code files in the form of commits. Thus, the same code file may appear multiple times in COMMITPACK for each change that was made to it. Therefore, The Stack only contains 3 terabytes of data, while COMMITPACK contains close to 4.

Table 9: **Overlap in repositories and usernames of COMMITPACK and The Stack.**

Statistic (↓)	COMMITPACK	The Stack 1.2	Shared	Shared (%)
Repositories	1,934,255	18,712,378	954,135	49.3%
Usernames	825,885	6,434,196	663,050	80.3%

G Pretraining on COMMITPACK

Due to the scale of COMMITPACK, it is also adequate as a large-scale pretraining dataset. Parts of COMMITPACK were included during the pretraining of StarCoder (Li et al., 2023b) in the format of `<commit_before>code_before<commit_msg>message<commit_after> code_after`. We also pretrain a new model, named SANTACODERPACK, with the same architecture as SantaCoder (Allal et al., 2023) on COMMITPACK using this format. We filter COMMITPACK for our six evaluation languages and samples that fit within 8192 tokens leaving us a total of 35B tokens. Following prior work (Muennighoff et al., 2023), we train on this data repeated close to 4 times for a total of 131B tokens taking 14 days. Detailed hyperparameters are in Appendix M.

In Table 10, we benchmark StarCoder and SANTACODERPACK on HUMANEVALFIX using the above-detailed commit format. We find that the commit format leads to very strong performance for StarCoder often surpassing the instruction tuned OCTOCODER from Table 2. However, this pretraining format is not suitable for HUMANEVALEXPLAIN limiting its universality. For SANTACODERPACK, we find performance comparable to SantaCoder, including checkpoints at 131B and 236B tokens. SANTACODERPACK performs slightly worse on Python than SantaCoder. We hypothesize that this discrepancy is due to a *multilingual tax*, as SANTACODERPACK needs to accommodate three additional coding languages (Go, C++ and Rust). SantaCoder has thus more capacity allocated to Python, JavaScript, and Java.

SANTACODERPACK may also be bottlenecked by its small model size of 1.1B parameters. More research into what exactly happens during pretraining (Xia et al., 2022; Biderman et al., 2023a) and how to unify pretraining and instruction tuning are needed. Prior work has also found that including raw code data during pretraining benefits some natural language tasks (Muennighoff et al., 2023). Future work may consider the effects of including code commit data on natural language tasks.

Table 10: **Zero-shot pass@1 (%) performance on HUMANEVALFIX of pretraining experiments.**

Model (↓)	Python	JavaScript	Java	Go	C++	Rust	Avg.
SantaCoder (131B tokens) Instruct Format	6.5	4.2	2.9	-	-	-	-
SantaCoder (236B tokens) Instruct Format	7.1	4.2	1.8	-	-	-	-
SANTACODERPACK (131B tokens) Commit Format	3.2	4.9	1.8	3.6	4.2	1.7	3.3
StarCoder Commit Format	32.7	33.6	33.0	31.9	31.6	20.2	30.5

H   Line Diff Format for Fixing Code

We finetune SantaCoder to experiment with different formatting strategies for fixing bugs comparing *full code generation* and *code diff generation*. When fixing a code bug, usually only a small part of the code needs to change. Only generating the code diff corresponding to the necessary change can make inference significantly more efficient by avoiding repeated characters in the output generation. We finetune SantaCoder on the Python, Java and JavaScript subset of COMMITPACKFT. We exclude other languages as SantaCoder has only been pretrained on these three languages (Allal et al., 2023).

Commit Format   For *full code generation*, we reuse the format that we employed for commits in StarCoder pretraining from Appendix G: `<commit_before>code_before<commit_msg>message<commit_after>code_after`. However, SantaCoder has not seen this format during pretraining and does not have special tokens like StarCoder for the delimiters. Thus, for SantaCoder e.g. `<commit_before>` is tokenized as `['<', 'commit', '_', 'before', '>']`.

Unified diff format   For *code diff generation*, a simple solution is using the unified diff format,[8] which is a standard way to display changes between code files in a compact and readable format (Lehman et al., 2022; Jung, 2021; Xu et al., 2022b; Monperrus et al., 2021). We depict an example of this format in Figure 6. However, the unified diff format still requires the model to output several unchanged lines below and after the actual modification. Thus, its efficiency gains are limited and there is still unnecessary duplication of the input.

Line diff format   To address the inefficiencies of the unified diff format, we propose the line diff format for representing code differences. There are two requirements for our format: **(1)** The diff can be unambiguously applied to the code before the commit to generate the code after the commit, and **(2)** the code diff should be as short as possible to maximize efficiency by avoiding the inclusion of unchanged code. In Figure 7, we show how our format addresses these. The line diff format keeps track of each change sequentially line-by-line to ensure the code can be correctly modified. By focusing only on the lines that change, we reduce the number of characters in the diff by 70% compared to the unified diff representation in Figure 6.

Both the unified diff format and our line diff format require the model to predict line numbers. This is very challenging when training on raw code as models need to count and keep track of line numbers. To simplify line number prediction, we automatically add line numbers to the raw code in the finetuning dataset for the line diff format. This allows the model to simply copy the line number into the output simplifying the diff generation. However, it diminishes efficiency slightly by adding additional input tokens that the model needs to process.

As summarized in Table 11, finetuning SantaCoder using the line diff format significantly improves performance compared to prior finetuning on HUMANEVALFIX across all languages. It also outperforms finetuning using the commit format, which only provides gains on JavaScript and Java compared to no finetuning. However, finetuning on the diff format may converge slower than the commit format as the diff format significantly differs from the raw code seen during pretraining.

---
[8]`https://en.wikipedia.org/wiki/Diff#Unified_format`

```
from typing import List                          from typing import List

def has_close_elements(numbers: List[float       def has_close_elements(numbers: List[float
    ], threshold: float) -> bool:                    ], threshold: float) -> bool:
    for idx, elem in enumerate(numbers):             for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(                    for idx2, elem2 in enumerate(
            numbers):                                        numbers):
            if idx != idx2:                                  if idx != idx2:
                distance = elem - elem2                          distance = abs(elem -
                if distance < threshold:                             elem2)
                    return True                              if distance < threshold:
                                                                 return True
    return False
                                                     return False
```

```
                @@ -4,7 +4,7 @@
                    for idx, elem in enumerate(numbers):
                        for idx2, elem2 in enumerate(numbers):
                            if idx != idx2:
-                               distance = elem - elem2
+                               distance = abs(elem - elem2)
                            if distance < threshold:
                                return True
```

Figure 6: **The first problem from the HUMANEVALFIX Python split and the necessary change to fix the bug in unified diff format**. *Top:* Code with and without the bug from Figure 11. *Bottom:* Necessary change to fix the bug in unified diff format.

```
-  7                  distance = elem - elem2
+  7                  distance = abs(elem - elem2)
```

Figure 7: **The line diff format for the problem from Figure 6.**

Figures 8, 9, 10 show line diff generations of our model. A limitation of our current line diff implementation is that it does not handle code insertion well. The inserted lines may change the line numbers of all following lines, which can result in problems when applying the diff. Further, the diff format is not useful for HUMANEVALEXPLAIN and HUMANEVALSYNTHESIZE. Future work could consider training models that can both be instructed to use the line diff format, such as for HUMANEVALFIX, but also explain or synthesize code without producing a diff.

Table 11: **Zero-shot pass@1 (%) performance on HUMANEVALFIX of SantaCoder formatting experiments.**

Model	Python	JavaScript	Java
SantaCoder	7.1	4.2	1.8
SantaCoder + Commit format finetuning	3.8	5.3	9.2
SantaCoder + Line diff format finetuning	**9.9**	**9.7**	**10.0**

```
-  3      let depth = 0, max_depth = 0;
+  3      let depth = 0, max_depth = 1;
- 12      return max_depth;
+ 12      return max_depth - 1;
- 14   return paren_string.split('␣')
- 15          .filter(x => x != '')
- 16          .map(x => parseParenGroup(x));
- 17 }
+ 14   let paren_list = paren_string.split('␣');
+ 15   let nested_parens = paren_list.map(x => parseParenGroup(x));
+ 16   return nested_parens.reduce((prev, curr) => {
+ 17     if (prev == 0) {
+ 18       return curr;
+ 19     } else {
+ 20       return curr - 1;
+ 21     }
+ 22   });
+ 23 }
```

Figure 8: **A line diff generation of our model on a JavaScript HUMANEVALFIX problem.**

```
- 18                if (current_depth < 0) {
+ 18                if (current_depth < 0 && current_string.length() > 0) {
```

Figure 9: **A line diff generation of our model on a Java HUMANEVALFIX problem.**

```
-  2      for i, l1 in enumerate(l):
-  3          for j in range(i, len(l)):
+  2      for i in range(0, len(l)):
+  3          for j in range(i+1, len(l)):
```

Figure 10: **A line diff generation of our model on a Python HUMANEVALFIX problem.**

I   Results on HUMANEVALFIXDOCS

The default version of HUMANEVALFIX does not include docstrings, but only provides the unit tests to the model alongside the buggy function. An alternative is providing docstrings as the source of ground truth for the model to fix the buggy function. Solving from docstrings is generally easier for models than from tests, as models can also solve it via pure code synthesis without looking at the buggy function at all. We provide results of some models on this variant in Table 12. For StarCoder, we distinguish two prompting formats: An instruction to fix bugs like in Figure 3 or the commit format it has seen during pretraining (Appendix G). OCTOCODER performs very strongly on this variant. Diff Codegen 2B (Bradley et al., 2023) performs poorly as its predicted code diffs are often irrelevant to the actual bug, see Figure 38.

Table 12: **Zero-shot pass@1 (%) performance on HUMANEVALFIXDOCS.**

Model	Python	JavaScript	Java	Go	C++	Rust	Avg.
Non-permissive models							
GPT-4	88.4	80.5	82.9	81.1	82.3	68.9	80.7
Permissive Models							
Diff Codegen 2B	0.0	0.1	0.0	0.3	0.0	0.2	0.1
StarCoder Commit Format	43.5	29.3	45.7	31.9	28.1	19.4	27.1
StarCoder Instruct Format	41.7	30.7	44.3	34.5	28.7	14.0	26.5
OCTOCODER	**53.8**	**48.1**	**54.3**	**54.9**	**49.2**	**32.1**	**48.7**

J   Full Instruction Data Ablations

We provide tabular results of the ablations from Figure 4 in Table 13. We try some additional mixtures, however, none of them perform better than COM-MITPACKFT + OASST. We experiment with changing the formatting to be `<commit_before>old code<commit_msg>message<commit_after>new code` for COM-MITPACKFT and `<commit_before><commit_msg>input<commit_after>output` for OASST referred to as the "Formatting" ablation. We hypothesized that aligning the formatting during instruction tuning with the commit format that we used during pretraining (Appendix G) would improve performance. While it seems to improve performance for HUMANEVALFIX compared to our default formatting (see Figure 17), it reduces performance on the other tasks leading to a worse average score of 35.3 in Table 13. "Target Loss" refers to an ablation where we mask loss for inputs as is commonly done during instruction tuning (Muennighoff et al., 2022b). While this leads to the best performance on HUMANEVALSYNTHESIZE, its average performance is worse compared to COMMITPACKFT + OASST, where the loss is computed over the full sequence. We also perform an ablation where we manually select 1178 high-quality samples (725 from OASST and 89, 61, 86, 72, 70 and 75 from COMMITPACKFT for Python, JavaScript, Java, Go, C++ and Rust, respectively). However, this manual selection did not outperform random selection for OCTOCODER. It performed better for OCTOGEEX, however, hence we used it for OCTOGEEX. We hypothesize that our models could achieve significantly better performance by further improving the quality of the instruction data beyond. This may necessitate very careful human selection of samples and manual editing of the data to ensure a uniform style in the outputs. We leave such explorations to future work.

Table 13: **Zero-shot pass@1 (%) performance across the Python split of HUMANEVALPACK for StarCoder instruction tuning data ablations.**

Instruction Tuning Dataset (↓)	HUMANEVALPACK Python			Average
	Fix	Explain	Synthesize	
Without instruction tuning	8.7	0.0	33.6	14.1
Self-Instruct (SI)	23.6	0.6	43.0	22.2
OASST	23.1	34.5	46.4	34.7
SI + OASST	24.9	28.7	46.2	33.3
xP3x + OASST	28.4	28.4	45.0	33.9
COMMITPACKFT + OASST	30.4	**35.1**	46.2	**37.2**
COMMITPACKFT + OASST (Formatting)	31.1	28.9	45.8	35.3
COMMITPACKFT + OASST (Target loss)	29.8	31.2	**47.8**	36.3
COMMITPACKFT + OASST (Manual)	27.2	29.6	45.8	34.2
COMMITPACKFT + xP3x + OASST	30.9	29.5	45.9	35.4
COMMITPACKFT + SI + xP3x + OASST	**31.4**	33.8	46.0	37.1

K   HUMANEVALFIX Bug Types

Table 14 contains an overview of bugs that were manually added by one of the authors to HumanEval solutions for the construction of HUMANEVALFIX. Figures 11-16 contain an example of each type from the Python split. The bug type for each problem is the same across all programming languages in HUMANEVALFIX, but for a few samples it affects a different part of the solution due to the code solutions not being perfectly parallel across languages.

Table 14: **HUMANEVALFIX bug types.**

Bug type	Subtype	Explanation	Example	Count
Missing logic		Misses code needed to solve the problem	Figure 11	33
Excess logic		Contains excess code leading to mistakes	Figure 12	31
Wrong logic	Value misuse	An incorrect value is used	Figure 13	44
	Operator misuse	An incorrect operator is used	Figure 14	25
	Variable misuse	An incorrect variable is used	Figure 15	23
	Function misuse	An incorrect function is used	Figure 16	8
Total				164

```
from typing import List


def has_close_elements(numbers: List[
    float], threshold: float) -> bool:
    """ Check if in given list of
        numbers, are any two numbers
        closer to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0,
        3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8,
        3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(
            numbers):
            if idx != idx2:
                distance = abs(elem -
                    elem2)
                if distance < threshold:
                    return True

    return False
```

```
from typing import List


def has_close_elements(numbers: List[
    float], threshold: float) -> bool:
    """ Check if in given list of
        numbers, are any two numbers
        closer to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0,
        3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8,
        3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(
            numbers):
            if idx != idx2:
                distance = elem - elem2
                if distance < threshold:
                    return True

    return False
```

Figure 11: **Missing logic bug example.** The buggy code (*right*) misses the 'abs' statement.

```
def truncate_number(number: float) ->
    float:
    """ Given a positive floating point
        number, it can be decomposed
        into
    and integer part (largest integer
        smaller than given number) and
        decimals
    (leftover part always smaller than
        1).

    Return the decimal part of the
        number.
    >>> truncate_number(3.5)
    0.5
    """
    return number % 1.0
```

```
def truncate_number(number: float) ->
    float:
    """ Given a positive floating point
        number, it can be decomposed
        into
    and integer part (largest integer
        smaller than given number) and
        decimals
    (leftover part always smaller than
        1).

    Return the decimal part of the
        number.
    >>> truncate_number(3.5)
    0.5
    """
    return number % 1.0 + 1.0
```

Figure 12: **Excess logic bug example.** The buggy code (*right*) incorrectly adds 1 to the result.

```
from typing import List, Tuple                    from typing import List, Tuple


def sum_product(numbers: List[int]) ->            def sum_product(numbers: List[int]) ->
    Tuple[int, int]:                                  Tuple[int, int]:
    """ For a given list of integers,                """ For a given list of integers,
        return a tuple consisting of a                   return a tuple consisting of a
        sum and a product of all the                     sum and a product of all the
        integers in a list.                              integers in a list.
    Empty sum should be equal to 0 and               Empty sum should be equal to 0 and
        empty product should be equal                    empty product should be equal
        to 1.                                            to 1.
    >>> sum_product([])                              >>> sum_product([])
    (0, 1)                                           (0, 1)
    >>> sum_product([1, 2, 3, 4])                    >>> sum_product([1, 2, 3, 4])
    (10, 24)                                         (10, 24)
    """                                              """
    sum_value = 0                                    sum_value = 0
    prod_value = 1                                   prod_value = 0

    for n in numbers:                                for n in numbers:
        sum_value += n                                   sum_value += n
        prod_value *= n                                  prod_value *= n
    return sum_value, prod_value                     return sum_value, prod_value
```

Figure 13: **Value misuse bug example.** The buggy code (*right*) incorrectly initializes the product to 0.

```
from typing import List                           from typing import List


def below_zero(operations: List[int]) ->          def below_zero(operations: List[int]) ->
    bool:                                              bool:
    """ You're given a list of deposit               """ You're given a list of deposit
        and withdrawal operations on a                   and withdrawal operations on a
        bank account that starts with                    bank account that starts with
    zero balance. Your task is to detect         zero balance. Your task is to detect
        if at any point the balance of                   if at any point the balance of
        account fallls below zero, and                   account fallls below zero, and
    at that point function should return         at that point function should return
        True. Otherwise it should                        True. Otherwise it should
        return False.                                    return False.
    >>> below_zero([1, 2, 3])                        >>> below_zero([1, 2, 3])
    False                                            False
    >>> below_zero([1, 2, -4, 5])                    >>> below_zero([1, 2, -4, 5])
    True                                            True
    """                                              """
    balance = 0                                      balance = 0

    for op in operations:                            for op in operations:
        balance += op                                    balance += op
        if balance < 0:                                  if balance == 0:
            return True                                      return True

    return False                                     return False
```

Figure 14: **Operator misuse bug example.** The buggy code (*right*) incorrectly checks for equality with 0.

```
from typing import List


def mean_absolute_deviation(numbers:
    List[float]) -> float:
    """ For a given list of input
        numbers, calculate Mean
        Absolute Deviation
    around the mean of this dataset.
    Mean Absolute Deviation is the
        average absolute difference
        between each
    element and a centerpoint (mean in
        this case):
    MAD = average | x - x_mean |
    >>> mean_absolute_deviation([1.0,
        2.0, 3.0, 4.0])
    1.0
    """
    mean = sum(numbers) / len(numbers)
    return sum(abs(x - mean) for x in
        numbers) / len(numbers)
```

```
from typing import List


def mean_absolute_deviation(numbers:
    List[float]) -> float:
    """ For a given list of input
        numbers, calculate Mean
        Absolute Deviation
    around the mean of this dataset.
    Mean Absolute Deviation is the
        average absolute difference
        between each
    element and a centerpoint (mean in
        this case):
    MAD = average | x - x_mean |
    >>> mean_absolute_deviation([1.0,
        2.0, 3.0, 4.0])
    1.0
    """
    mean = sum(numbers) / len(numbers)
    return sum(abs(x - mean) for x in
        numbers) / mean
```

Figure 15: **Variable misuse bug example.** The buggy code (*right*) incorrectly divides by the mean.

```
def flip_case(string: str) -> str:
    """ For a given string, flip
        lowercase characters to
        uppercase and uppercase to
        lowercase.
    >>> flip_case('Hello')
    'hELLO'
    """
    return string.swapcase()
```

```
def flip_case(string: str) -> str:
    """ For a given string, flip
        lowercase characters to
        uppercase and uppercase to
        lowercase.
    >>> flip_case('Hello')
    'hELLO'
    """
    return string.lower()
```

Figure 16: **Function misuse bug example.** The buggy code (*right*) incorrectly uses the 'lower()' function.

L Performance Breakdown by HUMANEVALFIX Bug Type

All bugs in HUMANEVALFIX are categorized into bug types as described in Appendix K. In Table 15, we break down the HUMANEVALFIX performance of select models from Table 2 by bug type. We find that models struggle most with bugs that require removing excess logic (e.g. Figure 12). WizardCoder is only able to solve 11% of excess logic bugs while solving about four times more bugs that relate to value misuse. The performance of OCTOGEEX and OCTOCODER is more stable than WizardCoder across the different bug types, possibly due to the diversity of COMMITPACKFT as displayed in Figure 2. GPT-4 performs best across all bug types.

Table 15: **Breakdown of HUMANEVALFIX Python pass@1 (%) performance by bug type for select models.** Statistics for each bug type are in Table 14.

Bug type	Subtype	OCTOGEEX	OCTOCODER	WizardCoder	GPT-4
Missing logic		24.2	24.4	31.2	45.5
Excess logic		16.3	16.9	11.0	38.7
	Value misuse	33.2	34.7	45.1	50.0
	Operator misuse	32.8	42.0	34.4	56.0
Wrong logic	Variable misuse	35.7	33.7	30.4	43.5
	Function misuse	25.0	37.5	37.5	50.0
Overall		28.1	30.4	31.8	47.0

M Hyperparameters

StarCoder finetuning (OCTOCODER)   For all experiments finetuning StarCoder, we use a learning rate of 5e-4 with a cosine schedule and linear warmup. We use a batch size of 32 and train for up to one epoch, as we did not observe benefits from more steps. We use LoRA (Hu et al., 2021) as we did not observe a significant difference from full finetuning. OCTOCODER was trained for 35 steps with a sequence length of 2048 and packing corresponding to 2.2 million total finetuning tokens.

CodeGeeX finetuning (OCTOGEEX)   To create OCTOGEEX, we finetune CodeGeeX2 for 35 steps with a batch size of 48 and a learning rate of 5e-5 largely following the OCTOCODER setup.

SantaCoder finetuning   For all experiments finetuning SantaCoder, we use a learning rate of 5e-5 with a cosine schedule and linear warmup. We finetune SantaCoder using a batch size of 64 for up to 200,000 steps.

SantaCoder pretraining (SANTACODERPACK)   We follow the setup from Allal et al. (2023) to pretrain on COMMITPACK except for using a sequence length of 8192 and the StarCoder tokenizer, which has special tokens for the commit format delimiters (see Appendix G). SANTACODERPACK utilizes Multi Query Attention (MQA) (Shazeer, 2019) but removes Fill-in-the-Middle (FIM) (Bavarian et al., 2022). We conducted pretraining on 32 A100 GPUs, totaling 250k training steps, with a global batch size of 64. Other hyperparameter settings follow SantaCoder, including using Adam with $\beta_1 = 0.9, \beta_2 = 0.95, \epsilon = 10^{-8}$, and a weight decay of 0.1. The learning rate is set to 2×10^{-4} and follows a cosine decay after warming up for 2% of the training steps.

N Prompts

The prompting format can significantly impact performance. In the spirit of true few-shot learning (Perez et al., 2021) we do not optimize prompts and go with the format provided by the respective model authors or the most intuitive format if none is provided. For each task, we define an instruction, an optional context and an optional function start (16). The function start is provided to make sure the model directly completes the function without having to search for the function in the model output. These three parts are then combined in slightly different ways for each model (Figures 17-23). We implement our evaluation using open-source frameworks (Ben Allal et al., 2022; Gao et al., 2021).

Table 16: **Instructions and function examples used.** If no function start or no context is present, that part is not added to the prompt (and the preceding newline is also removed).

HUMANEVALFIX	
Instruction	Fix bugs in has_close_elements.
| Context | ```python
from typing import List


def has_close_elements(numbers: List[float], threshold: float) ->
    bool:
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = elem - elem2
                if distance < threshold:
                    return True

    return False
``` |
| Function start | ```python
from typing import List


def has_close_elements(numbers: List[float], threshold: float) ->
    bool:
``` |
| **HUMANEVALEXPLAIN** | |
| Instruction (Describe) | Provide a concise natural language description of the code using at most 213 characters. |
| Context (Describe) | ```python
from typing import List


def has_close_elements(numbers: List[float], threshold: float) ->
    bool:
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True

    return False
``` |
| Instruction (Synthesize) | Write functional code in Python according to the description. |
| Context (Synthesize) | {Description generated by the model} |
| Function start (Synthesize) | ```python
from typing import List


def has_close_elements(numbers: List[float], threshold: float) ->
    bool:
``` |
| **HUMANEVALSYNTHESIZE** | |
| Instruction | Write a Python function 'has_close_elements(numbers: List[float], threshold: float) -> bool' to solve the following problem:
Check if in given list of numbers, are any two numbers closer to each other than given threshold.
»> has_close_elements([1.0, 2.0, 3.0], 0.5)
False
»> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
True |
| Function start | ```python
from typing import List


def has_close_elements(numbers: List[float], threshold: float) ->
    bool:
    """ Check if in given list of numbers, are any two numbers
        closer to each other than given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
``` |

Question: {instruction}
{context}

Answer:
{function_start}

Figure 17: **OCTOCODER and OCTOGEEX prompting format**

Below is an instruction that describes a task. Write a response that appropriately completes the request.

Instruction:
{instruction}
{context}

Response:
{function_start}

Figure 18: **WizardCoder prompting format from their codebase.**[9]

Below is an instruction that describes a task. Write a response that appropriately completes the request.

Instruction:
{instruction}
{context}

Response:{function_start}

Figure 19: **InstructCodeT5+ decoder prompting format from their codebase.**[10] The encoder receives the same prompt without the function start.

[9]https://github.com/nlpxucan/WizardLM/blob/9c6fb34d012d60dc4f31348ee0a8e35335c04564/WizardCoder/src/humaneval_gen.py#L38
[10]https://github.com/salesforce/CodeT5/blob/main/CodeT5%2B/humaneval/generate_codet5p.py#L89
[11]https://huggingface.co/HuggingFaceH4/starchat-beta

```
<|system|>
<|end|>
<|user|>
{instruction}
{context}<|end|>
<|assistant|>
{function_start}
```

Figure 20: **StarChat-β prompting format from their documentation.**[11]

```
{context}
{instruction}
{function_start}
```

Figure 21: **Default prompting format (StarCoder, BLOOMZ).** Used for Figure 3.

```
{context}
{instruction}
```

Figure 22: **GPT-4 prompting format.** Same as Figure 21 but excluding the function start, as we found the model would not continue from the provided function start, likely due to the chat mode.

```
{context}
{instruction}
 Start your code with:
{func_start}
```

Figure 23: **GPT-4 prompting format for HUMANEVALEXPLAIN (Synthesize).** We include the function start for the synthesize part of HUMANEVALEXPLAIN in case the function name is not included in the code description generated by GPT-4.

```
<NME> {filename}
<BEF> {context}
<MSG> {instruction}
<DFF>
```

Figure 24: **Prompting format for Diff Codegen 2B evaluated in Appendix I.**

O Examples

O.1 OCTOCODER

Input:

Question: Fix bugs in sum_product.

```
#include<stdio.h>
#include<vector>
using namespace std;
#include<algorithm>
#include<math.h>
#include<stdlib.h>
vector<int> sum_product(vector<int> numbers){
    int sum=0,product=0;
    for (int i=0;i<numbers.size();i++)
    {
        sum+=numbers[i];
        product*=numbers[i];
    }
    return {sum,product};
}
```

Answer:

```
#include<stdio.h>
#include<vector>
using namespace std;
#include<algorithm>
#include<math.h>
#include<stdlib.h>
vector<int> sum_product(vector<int> numbers){
```

OCTOCODER:

```
    int sum=0,product=1;
    for (int i=0;i<numbers.size();i++)
    {
        sum+=numbers[i];
        product*=numbers[i];
    }
    return {sum,product};
}
```

Figure 25: **Correct solution from OCTOCODER to a HUMANEVALFIX C++ problem.** The corresponding Python problem is depicted in Figure 13.

Question: Write a Java function 'public String decodeShift(String s)' to solve the following problem:
takes as input string encoded with encodeShift function. Returns decoded string.
Answer:

```
import java.util.*;
import java.lang.*;

class Solution {
    /**
    returns encoded string by shifting every character by 5 in the alphabet.
     */
    public String encodeShift(String s) {
        StringBuilder sb = new StringBuilder();
        for (char ch : s.toCharArray()) {
            sb.append((char) ('a' + ((ch + 5 - 'a') % 26)));
        }
        return sb.toString();
    }

    /**
    takes as input string encoded with encodeShift function. Returns decoded string.
     */
    public String decodeShift(String s) {
```

OCTOCODER*:*

```
        StringBuilder sb = new StringBuilder();
        for (char ch : s.toCharArray()) {
            sb.append((char) ('a' + ((ch - 'a' - 5) % 26)));
        }
        return sb.toString();
    }
}
```

Figure 26: **Incorrect solution from OCTOCODER to a HUMANEVALSYNTHESIZE Java problem.**
The correct solution would be to add 21 to ch instead of subtracting 5 from it.

Input:

Question: Provide a concise natural language description of the code using at most 671 characters.

```go
func FruitDistribution(s string,n int) int {
    split := strings.Split(s, " ")
        for _, i := range split {
                atoi, err := strconv.Atoi(i)
                if err != nil {
                        continue
                }
                n = n - atoi
        }
        return n
}
```

Answer:

OCTOCODER*:*

The function FruitDistribution takes two arguments, a string s and an integer n. The function splits the string s into a slice of strings using the space character as the delimiter. The function iterates over the slice and converts each string to an integer using the strconv.Atoi function. If the conversion is successful, the integer is subtracted from n. The function returns the value of n after all conversions have been performed.

Input (without previous input):

Question: Write functional code in Go according to the description.
The function FruitDistribution takes two arguments, a string s and an integer n. The function splits the string s into a slice of strings using the space character as the delimiter. The function iterates over the slice and converts each string to an integer using the strconv.Atoi function. If the conversion is successful, the integer is subtracted from n. The function returns the value of n after all conversions have been performed.
Answer:

```go
func FruitDistribution(s string,n int) int {
```

OCTOCODER*:*

```go
        fruits := strings.Split(s, " ")
        for _, fruit := range fruits {
                if num, err := strconv.Atoi(fruit); err == nil {
                        n -= num
                }
        }
        return n
}
```

Figure 27: **Correct solution from OCTOCODER to a HUMANEVALEXPLAIN Go problem.** Given its own description OCTOCODER writes a correct solution to the problem. Its solution is more compact and efficient than the original solution.

O.2 GPT-4

User:

Write a Go function 'func RescaleToUnit(numbers []float64) []float64' to solve the following problem:
Given a 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
>>> RescaleToUnit([1.0, 2.0, 3.0, 4.0, 5.0])
[0.0, 0.25, 0.5, 0.75, 1.0]

GPT-4:

Here is a Go function that solves the problem:

```go
package main

import (
    "fmt"
    "math"
)

func RescaleToUnit(numbers []float64) []float64 {
    min := numbers[0]
    max := numbers[0]

    // Find the min and max values in the list
    for _, number := range numbers {
        if number < min {
            min = number
        }
        if number > max {
            max = number
        }
    }

    // Apply the linear transform to each number in the list
    for i, number := range numbers {
        numbers[i] = (number - min) / (max - min)
    }

    return numbers
}

func main() {
    fmt.Println(RescaleToUnit([]float64{1.0, 2.0, 3.0, 4.0, 5.0}))
}
```

This function first finds the minimum and maximum values in the list of numbers. Then it applies the linear transformation to each number in the list. The transformation is '(number - min) / (max - min)', which scales the number such that 'min' becomes 0 and 'max' becomes 1. The transformed numbers are returned in a new list.

Figure 28: **Incorrect solution from GPT-4 to a HUMANEVALSYNTHESIZE Go problem.** GPT-4 includes an import that it does not use (math) which leads to a compilation error in Golang and thus the function does not pass the test cases.

Provide a concise natural language description of the code using at most 246 characters.

```
def truncate_number(number: float) -> float:
    return number % 1.0
```

GPT-4:

The code defines a function that takes a float as input and returns its fractional part by using the modulus operator with 1.0.

User (without previous conversation history):

The code defines a function that takes a float as input and returns its fractional part by using the modulus operator with 1.0.
Write functional code in Python according to the description.
Start your code with:

```
def truncate_number(number: float) -> float:
```

GPT-4:

```
def truncate_number(number: float) -> float:
    return number % 1.0
```

Figure 29: **Correct solution from GPT-4 to a HUMANEVALEXPLAIN Python problem.** GPT-4 writes a description that is significantly shorter than the docstring and thus within the character limit. It then generates the perfect solution from its own description. The docstring for this problem can be seen in Figure 12.

```
fn has_close_elements(numbers:Vec<f32>, threshold: f32) -> bool{
    for i in 0..numbers.len(){
        for j in 1..numbers.len(){

            if i != j {
                let distance:f32 = numbers[i] - numbers[j];

            if distance < threshold{
                return true;
            }

            }

        }
    }

    return false;

}
```

```
def check(has_close_elements):
    assert has_close_elements([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3) == True
    assert has_close_elements([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05) == False
    assert has_close_elements([1.0, 2.0, 5.9, 4.0, 5.0], 0.95) == True
    assert has_close_elements([1.0, 2.0, 5.9, 4.0, 5.0], 0.8) == False
    assert has_close_elements([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1) == True
    assert has_close_elements([1.1, 2.2, 3.1, 4.1, 5.1], 1.0) == True
    assert has_close_elements([1.1, 2.2, 3.1, 4.1, 5.1], 0.5) == False

check(has_close_elements)
```

Fix bugs in has_close_elements.

GPT-4:

The function has_close_elements is supposed to check if there are any two elements in the vector that are closer than a given threshold. However, there are a few issues with the current implementation:
1. The distance calculation is not correct. It should be the absolute difference between the two numbers, not just the difference.
2. The second loop should start from i+1, not 1. This is because we only need to compare each number with the numbers after it, not the numbers before it.
Here is the corrected function:

```
fn has_close_elements(numbers:Vec<f32>, threshold: f32) -> bool{
    for i in 0..numbers.len(){
        for j in i+1..numbers.len(){
            let distance:f32 = (numbers[i] - numbers[j]).abs();
            if distance < threshold{
                return true;
            }
        }
    }
    return false;
}
```

This function will now correctly check if there are any two elements in the vector that are closer than the given threshold.

Figure 30: **Correct solution from GPT-4 to a HUMANEVALFIX Rust problem.** GPT-4 correctly fixes the bug by adding the necessary 'abs' statement to compute the absolute value. It additionally improves the efficiency of the function by starting the inner loop from 'i+1' instead of 1 like in the original solution. As some problems in HUMANEVALPACK are not perfectly efficient, evaluating the ability of models to improve the efficiency of the solutions is another interesting benchmarking direction (Yetistiren et al., 2022).

```
def car_race_collision(n: int):
    return n**3



def check(car_race_collision):
    assert car_race_collision(2) == 4
    assert car_race_collision(3) == 9
    assert car_race_collision(4) == 16
    assert car_race_collision(8) == 64
    assert car_race_collision(10) == 100

check(car_race_collision)
```

Fix bugs in car_race_collision.

GPT-4:

The function car_race_collision seems to be working fine as it is. It takes an integer 'n' as input and returns the cube of 'n'. There don't seem to be any bugs in this function.

However, if the function is supposed to calculate something specific related to a car race collision and it's not doing that correctly, we would need more information about the expected behavior to fix it.

Figure 31: **Incorrect solution from GPT-4 to a HUMANEVALFIX Python problem.** GPT-4 does not catch that the test cases expect the function to square, not cube n. OCTOCODER correctly modifies the 3 to 2 in 6 out of $n = 20$ generations.

```
def rounded_avg(n, m):
    if m < n:
        return -1
    summation = 0
    for i in range(n, m+1):
        summation += i
    return bin(round(summation/(m - n)))

def check(rounded_avg):

    # Check some simple cases
    assert rounded_avg(1, 5) == "0b11"
    assert rounded_avg(7, 13) == "0b1010"
    assert rounded_avg(964,977) == "0b1111001010"
    assert rounded_avg(996,997) == "0b1111100100"
    assert rounded_avg(560,851) == "0b1011000010"
    assert rounded_avg(185,546) == "0b101101110"
    assert rounded_avg(362,496) == "0b110101101"
    assert rounded_avg(350,902) == "0b1001110010"
    assert rounded_avg(197,233) == "0b11010111"


    # Check some edge cases that are easy to work out by hand.
    assert rounded_avg(7, 5) == -1
    assert rounded_avg(5, 1) == -1
    assert rounded_avg(5, 5) == "0b101"

check(rounded_avg)
```

Fix bugs in rounded_avg.

GPT-4:

The function rounded_avg is supposed to calculate the average of all numbers between n and m (inclusive), round it, and then return the binary representation of the result.

There are several bugs in the function:

1. The function should return -1 if m < n, but it should also return -1 if m == n because in this case, the denominator of the average calculation would be 0, which would cause a division by zero error.

2. The function calculates the average by summing all numbers between n and m and then dividing by (m - n). However, this is incorrect because the number of integers between n and m (inclusive) is (m - n + 1), not (m - n).

3. The function uses the bin function to convert the result to binary. However, the bin function returns a string that starts with '0b' to indicate that it is a binary number. If the function is supposed to return the binary representation as an integer, it should remove this prefix.

Here is the corrected function:

```
def rounded_avg(n, m):
    if m <= n:
        return -1
    summation = 0
    for i in range(n, m+1):
        summation += i
    average = round(summation / (m - n + 1))
    return int(bin(average)[2:])
```

This function now correctly calculates the average, rounds it, and returns the binary representation as an integer. If n is greater than or equal to m, it returns -1.

Figure 32: **Incorrect solution from GPT-4 to a HUMANEVALFIX Python problem.** The bug consists of a "+1" missing after "m-n". GPT-4 correctly fixes that, however, it introduces two new bugs: Retuning "-1" when m=n and removing the "0b", both of which evidently lead to the provided unit tests failing.

O.3 WizardCoder

Input:

Below is an instruction that describes a task. Write a response that appropriately completes the request.
Instruction: Fix bugs in bf.

```
def bf(planet1, planet2):
    planet_names = ("Mercury", "Venus", "Earth", "Mars", "Jupyter", "Saturn", "Uranus", "
        Neptune")
    if planet1 not in planet_names or planet2 not in planet_names or planet1 == planet2:
        return ()
    planet1_index = planet_names.index(planet1)
    planet2_index = planet_names.index(planet2)
    if planet1_index < planet2_index:
        return (planet_names[planet1_index + 1: planet2_index])
    else:
        return (planet_names[planet2_index + 1 : planet1_index])

def check(bf):

    # Check some simple cases
    assert bf("Jupiter", "Neptune") == ("Saturn", "Uranus"), "First test error: " + str(
        len(bf("Jupiter", "Neptune")))
    assert bf("Earth", "Mercury") == ("Venus",), "Second test error: " + str(bf("Earth",
        "Mercury"))
    assert bf("Mercury", "Uranus") == ("Venus", "Earth", "Mars", "Jupiter", "Saturn"), "
        Third test error: " + str(bf("Mercury", "Uranus"))
    assert bf("Neptune", "Venus") == ("Earth", "Mars", "Jupiter", "Saturn", "Uranus"), "
        Fourth test error: " + str(bf("Neptune", "Venus"))


    # Check some edge cases that are easy to work out by hand.
    assert bf("Earth", "Earth") == ()
    assert bf("Mars", "Earth") == ()
    assert bf("Jupiter", "Makemake") == ()

check(bf)
```

Response:

```
def bf(planet1, planet2):
```

WizardCoder:

```
    planet_names = ("Mercury", "Venus", "Earth", "Mars", "Jupyter", "Saturn", "Uranus", "
        Neptune")
    if planet1 not in planet_names or planet2 not in planet_names or planet1 == planet2:
        return ()
    planet1_index = planet_names.index(planet1)
    planet2_index = planet_names.index(planet2)
    if planet1_index < planet2_index:
        return (planet_names[planet1_index + 1: planet2_index])
    else:
        return (planet_names[planet2_index + 1 : planet1_index])
```

Figure 33: **Incorrect solution from WizardCoder to a HUMANEVALFIX Python problem.** The bug consists of "Jupiter" being misspelled as "Jupyter" in the function. WizardCoder returns the same function without making a single change leaving the bug. This is a particularly difficult bug that only GPT-4 and OCTOCODER (only for 1 out of the n=20 generations) are able to solve.

O.4 BLOOMZ

```
const generateIntegers = (a, b) => {
  if (a > b) {
    let tmp = a;
    a = b;
    b = tmp;
  }
  let y = []
  for (let i = a; i <= b; i++) {
    if (i == 2 || i == 4 || i == 6 || i == 8) { y.push(i) }
  }
  return y
}
```

Provide a concise natural language description of the code using at most 218 characters.

BLOOMZ:

Generate a list of integers from a to b.

Input (without previous input):

Generate a list of integers from a to b.
Write functional code in JavaScript according to the description.

```
const generateIntegers = (a, b) => {
```

BLOOMZ:

```
  let integers = [];
  for (let i = a; i <= b; i++) {
    integers.push(i);
  }
  return integers;
}
```

Figure 34: **Incorrect solution from BLOOMZ to a** HUMANEVALEXPLAIN **JavaScript problem.**
BLOOMZ misses in its description that the function generates only the even digits between a and
b, in ascending order. Thus, its synthesized function based on its own description simply collects
all integers between a and b. BLOOMZ likely left out this part due to its bias for short and concise
answers. However, this bias backfires for this task.

O.5 StarCoder

```java
import java.util.*;
import java.lang.*;

class Solution {
    public Map<String, Integer> histogram(String test) {
        Map<String, Integer> dict1 = new HashMap<>();
        List<String> list1 = Arrays.asList(test.split(" " ));
        int t = 0;
        for (String i : list1) {
            if (Collections.frequency(list1, i) > t && !i.isEmpty()) {
                t = Collections.frequency(list1, i);
            }
        }
        if (t > 0) {
            for (String i : list1) {
                if (Collections.frequency(list1, i) == t) {
                    dict1.put(i, t);
                }
            }
        }
        return dict1;
    }
}
```

Provide a concise natural language description of the code using at most 423 characters.

StarCoder:

```java
public class Solution {
    public Map<String, Integer> histogram(String test) {


            int t = 0;

        }
    }
        if (t > 0) {
```

Write functional code in Java according to the description.

```java
import java.util.*;
import java.lang.*;

class Solution {
    public Map<String, Integer> histogram(String test) {
```

StarCoder:

```java
        Map<String, Integer> map = new HashMap<String, Integer>();
        int t = 0;
        for (int i = 0; i < test.length(); i++) {
            if (test.charAt(i) == 't') {
                t++;
            }
        }
        map.put("t", t);
        return map;
    }
}
```

Figure 35: **Incorrect solution from StarCoder to a HUMANEVALEXPLAIN Java problem.** Star-Coder is not capable of generating natural language descriptions and instead copies the solution. However, our post-processing removes substrings from the solution (§3) leading to a useless description. Given this description it generates an irrelevant function. As StarCoder cannot generate natural language explanations, we disqualify it from HUMANEVALEXPLAIN setting its score to 0 in Table 2.

O.6 InstructCodeT5+

Below is an instruction that describes a task. Write a response that appropriately completes the request.
Instruction: Fix bugs in has_close_elements.

```
from typing import List


def has_close_elements(numbers: List[float], threshold: float) -> bool:
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = elem - elem2
                if distance < threshold:
                    return True

    return False



def check(has_close_elements):
    assert has_close_elements([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3) == True
    assert has_close_elements([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05) == False
    assert has_close_elements([1.0, 2.0, 5.9, 4.0, 5.0], 0.95) == True
    assert has_close_elements([1.0, 2.0, 5.9, 4.0, 5.0], 0.8) == False
    assert has_close_elements([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1) == True
    assert has_close_elements([1.1, 2.2, 3.1, 4.1, 5.1], 1.0) == True
    assert has_close_elements([1.1, 2.2, 3.1, 4.1, 5.1], 0.5) == False

check(has_close_elements)
```

Response:from typing import List

```
def has_close_elements(numbers: List[float], threshold: float) -> bool:
```

```
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = elem - elem2
                if distance < threshold:
                    return True

    return False
```

Figure 36: **Incorrect solution from InstructCodeT5+ to a HUMANEVALFIX Python problem.**
InstructCodeT5+ simply repeated the function without adding the missing "abs" statement to turn the distance into an absolute value. This is one of the simplest bugs that most other models are able to fix. For InstructCodeT5+ the prompt starts directly after "Response:" without a newline (Figure 19).

O.7 StarChat-β

```
#include<stdio.h>
#include<math.h>
#include<vector>
#include<string>
#include<algorithm>
using namespace std;
#include<stdlib.h>
int specialFilter(vector<int> nums){
    int num=0;
    for (int i=0;i<nums.size();i++)
    if (nums[i]>10)
    {
        string w=to_string(nums[i]);
        if (w[0]%2==1 and w[w.length()-1]%2==1) num+=1;
    }
    return num;
}
```

Provide a concise natural language description of the code using at most 296 characters.

StarChat-β:

This code is a solution to the "Special Array" problem on LeetCode. It uses C++ and includes several standard libraries, including vector, string, and algorithm.
The code starts by declaring some variables and importing the necessary libraries. It then defines a specialFilter function that takes

Input (without previous input):

This code is a solution to the "Special Array" problem on LeetCode. It uses C++ and includes several standard libraries, including vector, string, and algorithm.
The code starts by declaring some variables and importing the necessary libraries. It then defines a specialFilter function that takes
Write functional code in C++ according to the description.

```
#include<stdio.h>
#include<math.h>
#include<vector>
#include<string>
#include<algorithm>
using namespace std;
#include<stdlib.h>
int specialFilter(vector<int> nums){
```

StarChat-β:

```
    int n = nums.size();
    int sum = 0;
    for(int i = 0; i < n; i++){
        sum += pow(nums[i], 2);
    }
    int target = (int)sqrt(sum);
    vector<int> res;
    for(int i = 0; i < n; i++){
        if(nums[i] > target){
            res.push_back(nums[i]);
        }
    }
    return res;
}
```

Figure 37: **Incorrect solution from StarChat-β to a HUMANEVALEXPLAIN C++ problem.** StarChat-β has a bias for excessively long answers. StarChat-β's description includes irrelevant parts like "..problem on LeetCode..includes several standard libraries.." (despite the problem not existing on LeetCode). As the description exceeds the 296-character limit for the code, it is cut off at the end.

O.8 Diff Codegen

Input:

```
<NME> truncate_number.py
<BEF> def truncate_number(number: float) -> float:
    """ Given a positive floating point number, it can be decomposed into
    and integer part (largest integer smaller than given number) and decimals
    (leftover part always smaller than 1).

    Return the decimal part of the number.
    >>> truncate_number(3.5)
    0.5
    """
    return number % 1.0 + 1.0
<MSG> Fix bugs in truncate_number.
<DFF>
```

Diff Codegen 2B

```
@@ -4,7 +4,7 @@ def truncate_number(number: float) -> float:
    and integer part (largest integer smaller than given number) and decimals
    (leftover part always smaller than 1).

-    Return the decimal part of the number.
+    Return
```

Figure 38: **Incorrect solution from Diff Codegen 2B to a HUMANEVALFIXDOCS Python problem.** Diff Codegen 2B suggests an irrelevant diff modifying parts of the docstring. The model commonly outputs diffs that modify the docstring or an import statement and rarely addresses the actual bug.

P   Limitations and Future Work

Model Execution   A promising avenue for improving performance on HUMANEVALFIX is letting the model execute the given code or its own generated code and inspect its output (Chen et al., 2022, 2023c; Yasunaga and Liang, 2021; Li et al., 2022a; Gao et al., 2023; Dong et al., 2023; Zhang et al., 2023c; Madaan et al., 2023b; Ni et al., 2023; Gou et al., 2023; Hu et al., 2023; Taylor et al., 2022; Nye et al., 2021). This could allow the model to discover which unit tests are failing and for what reason. The model could then simply iterate on the function until all unit tests are passing. We leave explorations of this strategy to improve performance on HUMANEVALPACK to future work.

Multi-file changes   For the creation of COMMITPACK, we have filtered out any commits that affect multiple files to ensure commits are very specific and account for the fact that most current models are only capable of operating on a single file. Allowing models to take multiple files as input and modify multiple files given a single instruction is a promising direction for future work. There is active research on using repository-level context (Ding et al., 2022; Shrivastava et al., 2023a,b; Zhang et al., 2023a; Liu et al., 2023d) and the necessary long context windows (Dai et al., 2019; Press et al., 2021; Sun et al., 2021; Dao et al., 2022; Peng et al., 2023; Liu et al., 2023c; Chen et al., 2023b).

Length-awareness   Current Code LLMs including OCTOCODER struggle with awareness about the length of their generated output. For HUMANEVALEXPLAIN, we instruct the models to limit their output to a given number of characters. While it is trivial for humans to count characters and adhere to the limit, all models tested frequently generate far too many characters. Prior work has shown that human raters are biased towards preferring longer texts (Wu and Aji, 2023) regardless of content. All models evaluated are instruction tuned on text that was at least indirectly assessed by human raters, hence they may be biased towards generating longer texts even if it means including literary bloat.

Better evaluation   Evaluating code instruction models is challenging for several reasons: **(1) Prompting:** The prompt can significantly impact the performance of large language models (Brown et al., 2020; Zhou et al., 2022; Muennighoff, 2022; Babe et al., 2023). To ensure fair

evaluation we use the prompting format put forth by the respective authors of the models and a simple intuitive prompt for models without a canonical prompt (see Appendix N). However, this may put models without a canonical prompt recommendation (e.g. BLOOMZ, GPT-4) at a slight disadvantage. OCTOCODER and OCTOGEEX perform best when prompted using the same format we use during training (Figure 17) and we recommend always using this format at inference.

(2) Processing: Models may accidentally impair otherwise correct code by e.g. including a natural language explanation in their output. We largely circumvent this issue through the use of strict stopping criteria and careful postprocessing (e.g. for GPT-4 we check if it has enclosed the code in backticks, and if so, extract only the inner part of the backticks discarding its explanations).

(3) Execution: When executing code to compute pass@k, it is important that the generated code matches the installed programming language version. Models may inadvertently use expressions from a different version (e.g. they may use the Python 2 syntax of `print "hi"`, which would fail in a Python 3 environment). In our evaluation, we did not find this to be a problem, however, as models become more capable, it may make sense to specify the version. Future prompts may include the version (e.g. "use JDK 1.18.0") or provide models with an execution environment that has the exact version installed that will be used for evaluation.

(4) Comprehensiveness: Executing code can only reflect functional correctness lacking a comprehensive understanding of quality. Compared to execution-based evaluation, the human judgment of code quality can be considered more comprehensive as humans can consider factors beyond correctness. Directly hiring human annotators can be inefficient and expensive, and therefore researchers have explored approaches to automate human-aligned evaluation via LLMs (Fu et al., 2023; Liu et al., 2023e; Zhuo, 2023). However, recent work (Wang et al., 2023b) suggests LLM-based evaluation can be biased towards certain contexts. Future work on automating the human-aligned evaluation of instruction tuned Code LLMs while avoiding such bias is needed.

Reward Models Our commit datasets, COMMITPACK and COMMITPACKFT, also lend themselves well for learning human preferences. The changed code after a commit generally represents a human-preferred version of the code (else the code would not have been modified). Thus, one could train a reward model that given the code before and after a commit, learns that the code afterward is better. Similar to prior work (Ouyang et al., 2022), this reward model could then be used to guide a language model to generate code that is preferred by humans.