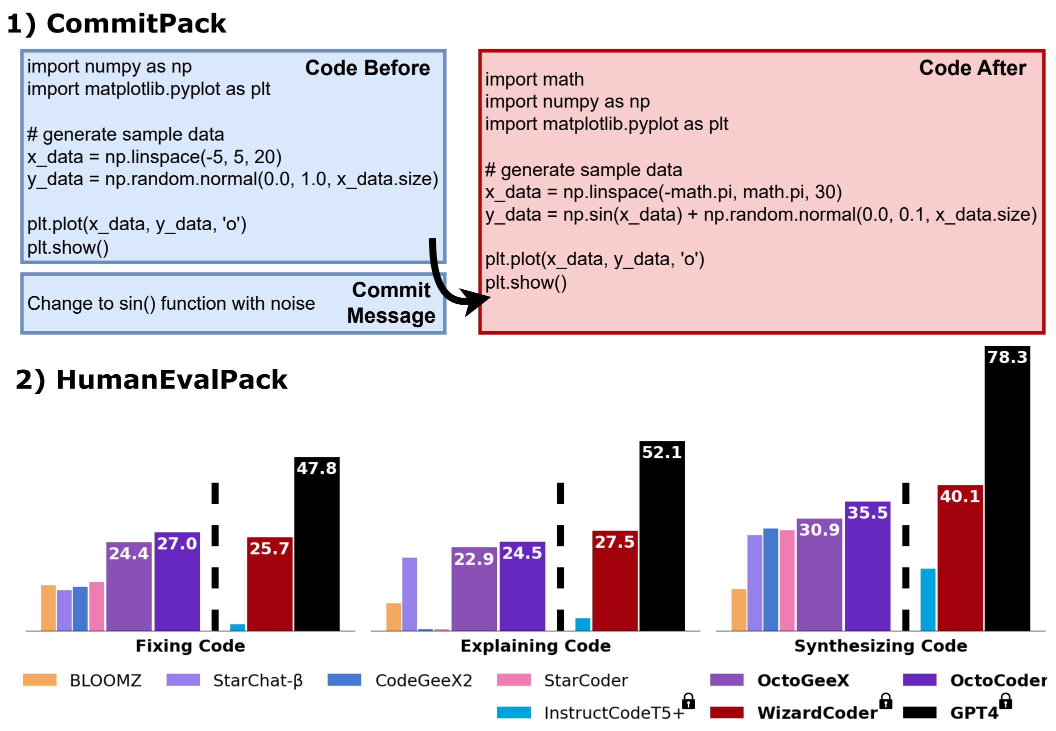
OctoPack: Instruction Tuning Code Large Language Models

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`.



```
import numpy as np
import matplotlib.pyplot as plt

# generate sample data
x_data = np.linspace(-5, 5, 20)
y_data = np.random.normal(0.0, 1.0, x_data.size)

plt.plot(x_data, y_data, 'o')
plt.show()
```

Change to sin() function with noise

```
import math
import numpy as np
import matplotlib.pyplot as plt

# generate sample data
x_data = np.linspace(-math.pi, math.pi, 30)
y_data = np.sin(x_data) + np.random.normal(0.0, 0.1, x_data.size)

plt.plot(x_data, y_data, 'o')
plt.show()
```

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 2GB variant containing high-quality code instructions used 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

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

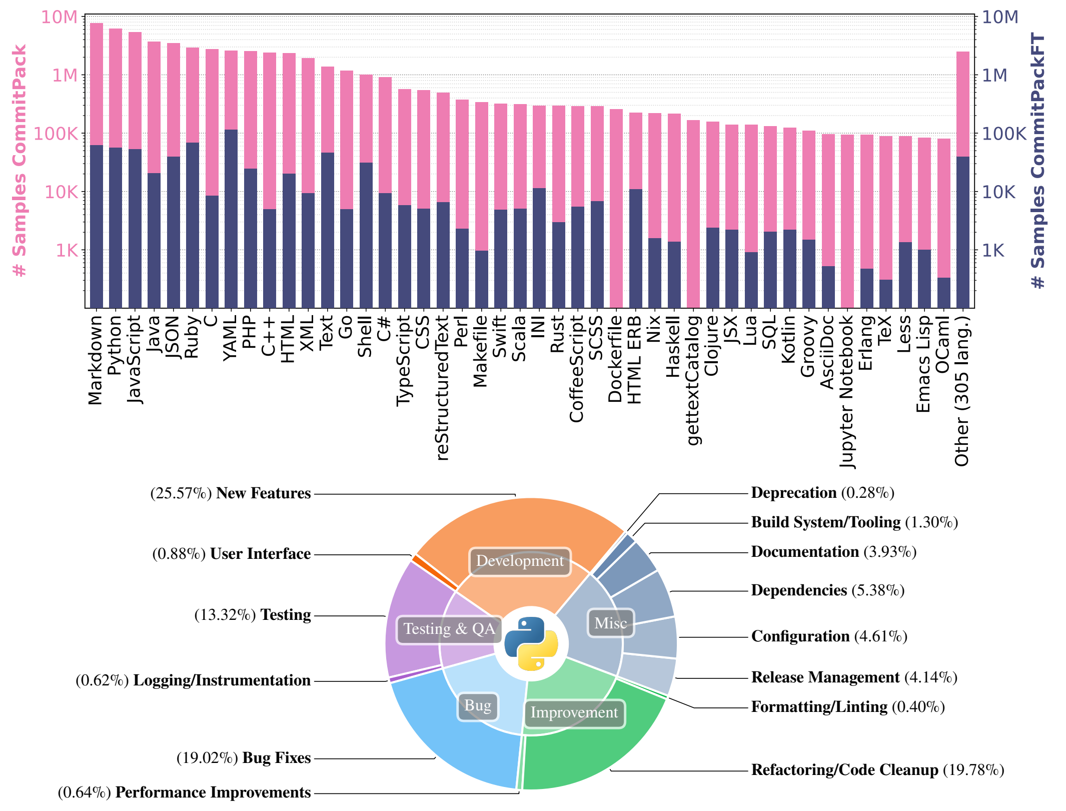


Figure 2: **Overview of COMMITPACK and COMMITPACKFT.** *Top:* Language distribution of the full commit data (COMMITPACK) and the variant filtered for high-quality instructions (COMMITPACKFT). See Appendix C for the full distribution. *Bottom:* Task distribution of commits on the Python subset of COMMITPACKFT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

	Base dataset			Subset		
Dataset (↓)	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	277	742,273	100%	6	5,000	100%

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.

languages during instruction tuning boosts their performance further. We hypothesize that code data 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.

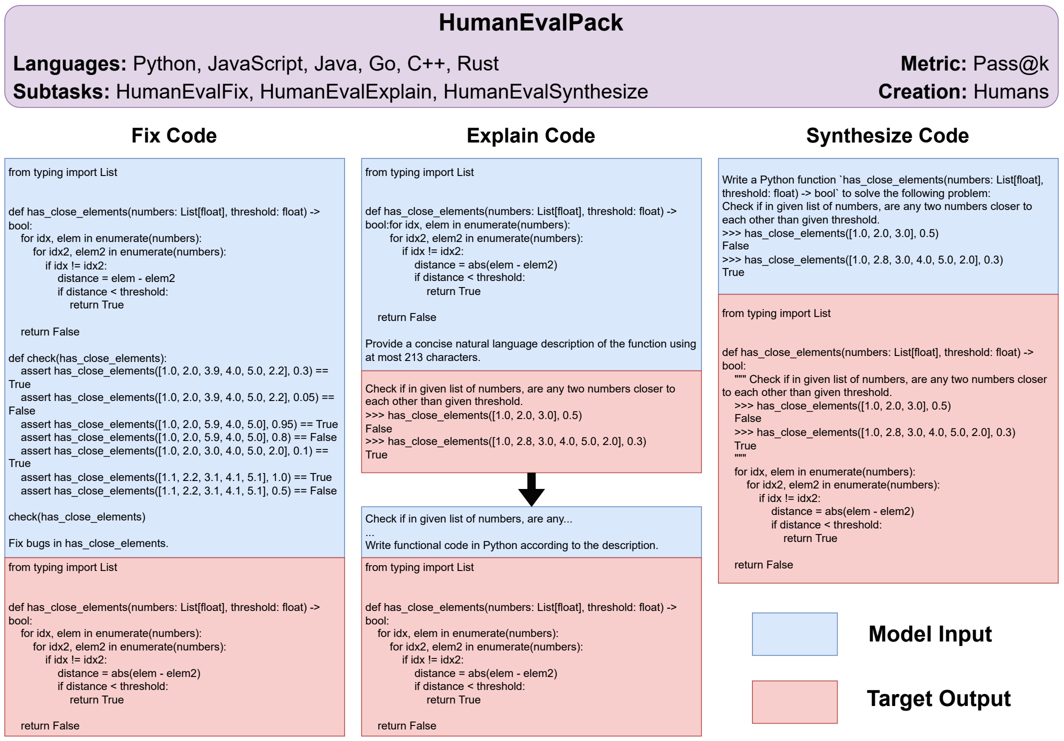
COMMITPACK To create the dataset, we use commit metadata from the GitHub action dump on Google BigQuery.[3] We apply quality filters, filter for commercially friendly licenses, and discard commits that affect more than a single file to ensure commit messages are very specific and to avoid additional 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 almost 4 terabytes of data covering 350 programming languages (COMMITPACK). As instruction tuning does not require so much data (Zhou et al., 2023a; Touvron et al., 2023), we apply several strict filters to

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

reduce the dataset to 2 gigabytes and 277 languages (CommitPackFT). These include 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. In Appendix G, we also experiment with pretraining on the entirety of CommitPack.

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). Only a few of the dialogues contain code. We reuse a filtered variant 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


HumanEvalPack
Languages: Python, JavaScript, Java, Go, C++, Rust
Metric: Pass@k
Subtasks: HumanEvalFix, HumanEvalExplain, HumanEvalSynthesize
Creation: Humans
Fix Code
Explain Code
Synthesize Code
Model Input
Target Output

Fix Code — Model Input:

```
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)

Fix bugs in has_close_elements.
```

Fix Code — Target Output:

```
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
```

Explain Code — Model Input:

```
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

Provide a concise natural language description of the function using
at most 213 characters.
```

Explain Code — Target Output:

```
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
```

↓

Explain Code — Model Input:

```
Check if in given list of numbers, are any...
...
Write functional code in Python according to the description.
```

Explain Code — Target Output:

```
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
```

Synthesize Code — Model Input:

```
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
```

Synthesize Code — Target Output:

```
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
```

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 L. 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 K.

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 & 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 changed scores from 75.0% to 75.2% pass@1 on HUMANEVALSYNTHESIZE Python while providing 20x cost savings.

Python HumanEval is the most widely used code benchmark and many training datasets have already been decontaminated for it (Kocetkov et al., 2022). By manually extending HumanEval, we ensure existing decontamination remains valid to enable fair evaluation. However, this may not hold for all models (e.g. GPT-4), thus results should be interpreted carefully.

4 OCTOCODER: BEST COMMERCIALLY LICENSED CODE LLM

4.1 ABLATING INSTRUCTION DATA CHOICES

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 18).

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

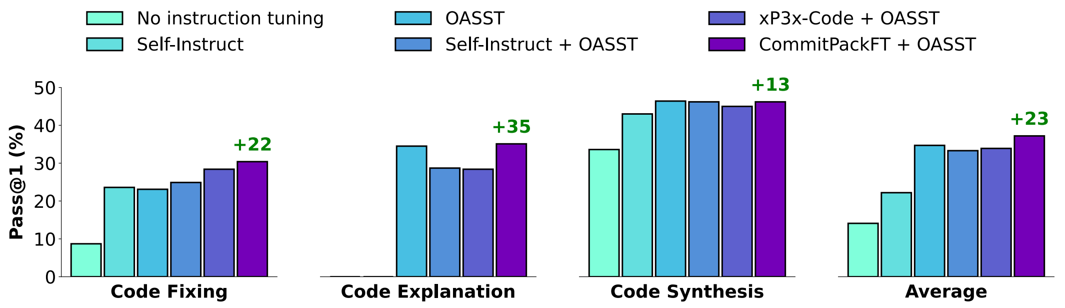


Figure 4: **Comparing permissively licensed instruction datasets by instruction tuning StarCoder.** Models are evaluated on the Python subset of HUMANEVALPACK.

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. Training hyperparameters for both models are in Appendix P.

4.2 COMPARING WITH OTHER MODELS

We benchmark OCTOCODER and OCTOGEEX with state-of-the-art Code LLMs on HUMANEVALPACK 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 Q.

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

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**

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).

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).

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 34) or they introduce new bugs (e.g. GPT-4 in Figure 33). We analyze model performance by bug type in Appendix M 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 35) or excessively long explanations that end up being cut off (e.g. StarChat-β in Figure 38). 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 & Robbes, 2023; Fakhoury et al., 2023; Skreta et al., 2023) (also see work on edit models more generally (Reid & 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 (Wei et al., 2022; Longpre et al., 2023a; Iyer et al., 2022), multilingual tasks (Muennighoff et al., 2022b; 2024; Yong et al., 2022; Üstün et al., 2024), and 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 to more general coding tasks via 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 & Robbes, 2021; Ye et al., 2021; Xia & 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 & 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 & 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.

ACKNOWLEDGEMENTS

We thank Hugging Face for providing compute instances. We are extremely grateful to Rodrigo Garcia for the Rust translations, Dimitry Ageev and Calum Bird for help with GPT-4 evaluation, Loubna Ben Allal for help on evaluation, Arjun Guha for insightful discussions on chaining evaluation tasks to avoid evaluating with BLEU, Lewis Tunstall for help on the OASST data, Victor Sanh and Nadav Timor for discussions, Jiaxi Yang for logo editing and domain classification prompting design, Ghosal et al. (2023); Zeng et al. (2023) for design inspiration, Harm de Vries for feedback and all members of BigCode for general support. Finally, we thank every programmer who takes the time to write informative commit messages.

REFERENCES

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

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

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

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

Hannah McLean Babe, Sydney Nguyen, Yangtian Zi, Arjun Guha, Molly Q Feldman, and Carolyn Jane Anderson. Studenteval: A benchmark of student-written prompts for large language models of code. *arXiv preprint arXiv:2306.04556*, 2023.

Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*, 2022. URL `https://arxiv.org/abs/2204.05862`.

Satanjeev Banerjee and Alon Lavie. 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*, pp. 65–72, 2005.

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

Mohammad Bavarian, Heewoo Jun, Nikolas A. Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*, 2022.

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

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

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. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, pp. 2397–2430. PMLR, 2023b.

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

Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022.

Herbie Bradley, Honglu Fan, Harry Saini, Reshinth Adithyan, Shivanshu Purohit, and Joel Lehman. Diff models - a new way to edit code. *CarperAI Blog*, Jan 2023. URL https://carper.ai/diff-model/.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Conference on Neural Information Processing Systems (NeurIPS)*, 2020. URL https://papers.nips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

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

Nghi DQ Bui, Hung Le, Yue Wang, Junnan Li, Akhilesh Deepak Gotmare, and Steven CH Hoi. Codetf: One-stop transformer library for state-of-the-art code llm. *arXiv preprint arXiv:2306.00029*, 2023.

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. Multipl-e: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 2023.

Sahil Chaudhary. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca, 2023.

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

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

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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023b.

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

Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

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

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*, 2022. URL https://arxiv.org/abs/2210.11416.

Haotian Cui, Chenglong Wang, Junjie Huang, Jeevana Priya Inala, Todd Mytkowicz, Bo Wang, Jianfeng Gao, and Nan Duan. Codeexp: Explanatory code document generation. *arXiv preprint arXiv:2211.15395*, 2022.

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

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

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

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

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

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *arXiv preprint arXiv:2304.07590*, 2023.

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

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*, 2023.

Jane Dwivedi-Yu, Timo Schick, Zhengbao Jiang, Maria Lomeli, Patrick Lewis, Gautier Izacard, Edouard Grave, Sebastian Riedel, and Fabio Petroni. Editeval: An instruction-based benchmark for text improvements. *arXiv preprint arXiv:2209.13331*, 2022.

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

Kawin Ethayarajh, Winnie Xu, Niklas Muennighoff, Dan Jurafsky, and Douwe Kiela. Kto: Model alignment as prospect theoretic optimization, 2024.

Sarah Fakhoury, Saikat Chakraborty, Madan Musuvathi, and Shuvendu K Lahiri. Towards generating functionally correct code edits from natural language issue descriptions. *arXiv preprint arXiv:2304.03816*, 2023.

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

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

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

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. A framework for few-shot language model evaluation, 2021. URL `https://doi.org/10.5281/zenodo.5371628`.

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

Deepanway Ghosal, Yew Ken Chia, Navonil Majumder, and Soujanya Poria. Flacuna: Unleashing the problem solving power of vicuna using flan fine-tuning. *arXiv preprint arXiv:2307.02053*, 2023.

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

Dirk Groeneveld, Iz Beltagy, Pete Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, A. Jha, Hamish Ivison, Ian Magnusson, Yizhong Wang, Shane Arora, David Atkinson, Russell Authur, Khyathi Raghavi Chandu, Arman Cohan, Jennifer Dumas, Yanai Elazar, Yuling Gu, Jack Hessel, Tushar Khot, William Merrill, Jacob Daniel Morrison, Niklas Muennighoff, Aakanksha Naik, Crystal Nam, Matthew E. Peters, Valentina Pyatkin, Abhilasha Ravichander, Dustin Schwenk, Saurabh Shah, Will Smith, Emma Strubell, Nishant Subramani, Mitchell Wortsman, Pradeep Dasigi, Nathan Lambert, Kyle Richardson, Luke Zettlemoyer, Jesse Dodge, Kyle Lo, Luca Soldaini, Noah A. Smith, and Hanna Hajishirzi. Olmo: Accelerating the science of language models. 2024. URL https://api.semanticscholar.org/CorpusID:267365485.

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

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. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*, 2023.

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

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

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

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

Yi Hu, Haotong Yang, Zhouchen Lin, and Muhan Zhang. Code prompting: a neural symbolic method for complex reasoning in large language models. *arXiv preprint arXiv:2305.18507*, 2023.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 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)*, pp. 2073–2083, 2016.

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. Opt-iml: Scaling language model instruction meta learning through the lens of generalization. *arXiv preprint arXiv:2212.12017*, 2022. URL https://arxiv.org/abs/2212.12017.

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

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. Impact of code language models on automated program repair. *arXiv preprint arXiv:2302.05020*, 2023.

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

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *arXiv preprint arXiv:2303.03004*, 2023.

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

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

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. Openassistant conversations–democratizing large language model alignment. *arXiv preprint arXiv:2304.07327*, 2023.

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

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. The bigscience roots corpus: A 1.6 tb composite multilingual dataset. *Advances in Neural Information Processing Systems*, 35:31809–31826, 2022.

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

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

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

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

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

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

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

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual 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*, pp. 55–56, 2017.

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

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

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*, 2023c.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*, 2023d.

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

Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V Le, Barret Zoph, Jason Wei, et al. The flan collection: Designing data and methods for effective instruction tuning. *arXiv preprint arXiv:2301.13688*, 2023a.

Shayne Longpre, Gregory Yauney, Emily Reif, Katherine Lee, Adam Roberts, Barret Zoph, Denny Zhou, Jason Wei, Kevin Robinson, David Mimno, et al. A pretrainer's guide to training data: Measuring the effects of data age, domain coverage, quality, & toxicity. *arXiv preprint arXiv:2305.13169*, 2023b.

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

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

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

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

Sewon Min, Mike Lewis, Luke Zettlemoyer, and Hannaneh Hajishirzi. MetaICL: Learning to learn in context. *Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2022. URL `https://arxiv.org/abs/2110.15943`.

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

Niklas Muennighoff. Sgpt: Gpt sentence embeddings for semantic search. *arXiv preprint arXiv:2202.08904*, 2022.

Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. Mteb: Massive text embedding benchmark. *arXiv preprint arXiv:2210.07316*, 2022a. doi: 10.48550/ARXIV.2210.07316. URL `https://arxiv.org/abs/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. Crosslingual generalization through multitask finetuning. *arXiv preprint arXiv:2211.01786*, 2022b.

Niklas Muennighoff, Alexander M Rush, Boaz Barak, Teven Le Scao, Aleksandra Piktus, Nouamane Tazi, Sampo Pyysalo, Thomas Wolf, and Colin Raffel. Scaling data-constrained language models. *arXiv preprint arXiv:2305.16264*, 2023.

Niklas Muennighoff, Hongjin Su, Liang Wang, Nan Yang, Furu Wei, Tao Yu, Amanpreet Singh, and Douwe Kiela. Generative representational instruction tuning, 2024.

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

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*, 2023.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021. URL https://openreview.net/forum?id=iedYJm92o0a.

OpenAI. Gpt-4 technical report, 2023.

Gabriel Orlanski, Kefan Xiao, Xavier Garcia, Jeffrey Hui, Joshua Howland, Jonathan Malmaud, Jacob Austin, Rishah Singh, and Michele Catasta. Measuring the impact of programming language distribution. *arXiv preprint arXiv:2302.01973*, 2023.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022. URL https://arxiv.org/abs/2203.02155.

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

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

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

Luiza Amador Pozzobon, Beyza Ermis, Patrick Lewis, and Sara Hooker. 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*, 2023.

Julian Aron Prenner and Romain Robbes. Automatic program repair with openai's codex: Evaluating quixbugs. *arXiv preprint arXiv:2111.03922*, 2021.

Julian Aron Prenner and Romain Robbes. Runbugrun–an executable dataset for automated program repair. *arXiv preprint arXiv:2304.01102*, 2023.

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

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36, 2024.

Vipul Raheja, Dhruv Kumar, Ryan Koo, and Dongyeop Kang. Coedit: Text editing by task-specific instruction tuning. *arXiv preprint arXiv:2305.09857*, 2023.

Machel Reid and Graham Neubig. Learning to model editing processes. *arXiv preprint arXiv:2205.12374*, 2022.

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

Victor Sanh, Albert Webson, Colin Raffel, Stephen Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, et al. Multitask prompted training enables zero-shot task generalization. *International Conference on Learning Representations (ICLR)*, 2022. URL https://openreview.net/forum?id=9Vrb9D0WI4.

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

Teven Le Scao, Thomas Wang, Daniel Hesslow, Lucile Saulnier, Stas Bekman, M Saiful Bari, Stella Bideman, Hady Elsahar, Niklas Muennighoff, Jason Phang, et al. What language model to train if you have one million gpu hours? *arXiv preprint arXiv:2210.15424*, 2022b.

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

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

Noam M. Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

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. Pangu-coder2: Boosting large language models for code with ranking feedback, 2023.

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

Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. Repofusion: Training code models to understand your repository. *arXiv preprint arXiv:2306.10998*, 2023a.

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

Shivalika Singh, Freddie Vargus, Daniel Dsouza, Börje F Karlsson, Abinaya Mahendiran, Wei-Yin Ko, Herumb Shandilya, Jay Patel, Deividas Mataciunas, Laura OMahony, et al. Aya dataset: An open-access collection for multilingual instruction tuning. *arXiv preprint arXiv:2402.06619*, 2024.

Marta Skreta, Naruki Yoshikawa, Sebastian Arellano-Rubach, Zhi Ji, Lasse Bjørn Kristensen, Kourosh Darvish, Alán Aspuru-Guzik, Florian Shkurti, and Animesh Garg. Errors are useful prompts: Instruction guided task programming with verifier-assisted iterative prompting. *arXiv preprint arXiv:2303.14100*, 2023.

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

Luca Soldaini, Rodney Kinney, Akshita Bhagia, Dustin Schwenk, David Atkinson, Russell Authur, Ben Bogin, Khyathi Raghavi Chandu, Jennifer Dumas, Yanai Elazar, Valentin Hofmann, A. Jha, Sachin Kumar, Li Lucy, Xinxi Lyu, Nathan Lambert, Ian Magnusson, Jacob Daniel Morrison, Niklas Muennighoff, Aakanksha Naik, Crystal Nam, Matthew E. Peters, Abhilasha Ravichander, Kyle Richardson, Zejiang Shen, Emma Strubell, Nishant Subramani, Oyvind Tafjord, Pete Walsh, Luke Zettlemoyer, Noah A. Smith, Hanna Hajishirzi, Iz Beltagy, Dirk Groeneveld, Jesse Dodge, and Kyle Lo. Dolma: an open corpus of three trillion tokens for language model pretraining research. 2024. URL https://api.semanticscholar.org/CorpusID:267364861.

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. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models, 2022. URL https://arxiv.org/abs/2206.04615.

Simeng Sun, Kalpesh Krishna, Andrew Mattarella-Micke, and Mohit Iyyer. Do long-range language models actually use long-range context? *ArXiv*, abs/2109.09115, 2021. URL https://api.semanticscholar.org/CorpusID:237572264.

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

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

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

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

Ahmet Üstün, Viraat Aryabumi, Zheng-Xin Yong, Wei-Yin Ko, Daniel D'souza, Gbemileke Onilude, Neel Bhandari, Shivalika Singh, Hui-Lee Ooi, Amr Kayid, et al. Aya model: An instruction finetuned open-access multilingual language model. *arXiv preprint arXiv:2402.07827*, 2024.

Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. SuperGLUE: A stickier benchmark for general-purpose language understanding systems. *Conference on Neural Information Processing Systems (NeurIPS)*, 2019. URL https://arxiv.org/abs/1905.00537.

Ben Wang and Aran Komatsuzaki. Gpt-j-6b: A 6 billion parameter autoregressive language model, 2021.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations (ICLR)*, 2023a. URL https://openreview.net/forum?id=1PL1NIMMrw.

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

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

Yizhong Wang, Hamish Ivison, Pradeep Dasigi, Jack Hessel, Tushar Khot, Khyathi Raghavi Chandu, David Wadden, Kelsey MacMillan, Noah A Smith, Iz Beltagy, et al. How far can camels go? exploring the state of instruction tuning on open resources. *arXiv preprint arXiv:2306.04751*, 2023b.

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

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

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

Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. Finetuned language models are zero-shot learners. *International Conference on Learning Representations (ICLR)*, 2022. URL https://openreview.net/forum?id=gEZrGCozdqR.

Jiayi Wei, Greg Durrett, and Isil Dillig. Coeditor: Leveraging contextual changes for multi-round code auto-editing. *arXiv preprint arXiv:2305.18584*, 2023.

Minghao Wu and Alham Fikri Aji. Style over substance: Evaluation biases for large language models. *arXiv preprint arXiv:2307.03025*, 2023.

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

Mengzhou Xia, Mikel Artetxe, Chunting Zhou, Xi Victoria Lin, Ramakanth Pasunuru, Danqi Chen, Luke Zettlemoyer, and Ves Stoyanov. Training trajectories of language models across scales. *arXiv preprint arXiv:2212.09803*, 2022.

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

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

Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, and Hanghang Tong. Combining code context and fine-grained code difference for commit message generation. In *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, pp. 242–251, 2022b.

Zhiyang Xu, Ying Shen, and Lifu Huang. Multiinstruct: Improving multi-modal zero-shot learning via instruction tuning, 2023b.

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

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

Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 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*, pp. 62–71, 2022.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pp. 476–486. ACM, 2018. doi: https://doi.org/10.1145/3196398.3196408.

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. Bloom+1: Adding language support to bloom for zero-shot prompting. *arXiv preprint arXiv:2212.09535*, 2022.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Tao Xie, and Qianxiang Wang. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. *arXiv preprint arXiv:2302.00288*, 2023.

Yan Zeng, Hanbo Zhang, Jiani Zheng, Jiangnan Xia, Guoqiang Wei, Yang Wei, Yuchen Zhang, and Tao Kong. What matters in training a gpt4-style language model with multimodal inputs? *arXiv preprint arXiv:2307.02469*, 2023.

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

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

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

Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. Repairing bugs in python assignments using large language models. *arXiv preprint arXiv:2209.14876*, 2022b.

Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. Coditt5: Pretraining for source code and natural language editing. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–12, 2022c.

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

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

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

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

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

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

Terry Yue Zhuo. Large language models are state-of-the-art evaluators of code generation. *arXiv preprint arXiv:2304.14317*, 2023.

Terry Yue Zhuo, Armel Zebaze, Nitchakarn Suppattarachai, Leandro von Werra, Harm de Vries, Qian Liu, and Niklas Muennighoff. Astraios: Parameter-efficient instruction tuning code large language models. *arXiv preprint arXiv:2401.00788*, 2024.