Magicoder: Empowering Code Generation with OSS-INSTRUCT

Yuxiang Wei [1] **Zhe Wang** [2 †] **Jiawei Liu** [1] **Yifeng Ding** [1] **Lingming Zhang** [1]

Abstract

We introduce Magicoder, a series of *fully open-source* (code, weights, and data) Large Language Models (LLMs) for code that significantly closes the gap with top code models while having no more than 7B parameters. Magicoder models are trained on 75K synthetic instruction data using OSS-INSTRUCT, a novel approach to enlightening LLMs with open-source code snippets to generate diverse instruction data for code. Our main motivation is to mitigate the inherent bias of the synthetic data generated by LLMs through the wealth of open-source references for the production of more realistic and controllable data. The orthogonality of OSS-INSTRUCT and other data generation methods like Evol-Instruct further enables us to build an enhanced Magicoder$\mathcal{S}$. Both Magicoder and Magicoder$\mathcal{S}$ substantially outperform state-of-the-art code models with similar or even larger sizes on a wide range of coding benchmarks. Notably, Magicoder$\mathcal{S}$-CL-7B based on CODELLAMA even surpasses the prominent ChatGPT on HumanEval+ (66.5 vs. 65.9 in pass@1). Overall, OSS-INSTRUCT opens a new direction for crafting diverse synthetic instruction data for code using abundant open-source references.

1. Introduction

Code generation, also known as program synthesis (Gulwani et al., 2017), is a long-standing challenge in computer science. In the past few decades, a large body of research has been studying symbolic approaches, such as abstraction-based synthesis (Wang et al., 2017; Feng et al., 2018) for general-purpose synthesis problems and programming by examples (Cambronero et al., 2023; Liu et al., 2023a) for domain-specific tasks. Until recently, Large Language Models (LLMs) trained on code (Austin et al., 2021; Chen et al., 2021) has shown outstanding breakthroughs in generating code that accurately satisfies user intents, and they are widely deployed to assist real-world software development (Microsoft, 2023b; Services, 2023).

Initially, closed-source models such as GPT-3.5 Turbo (OpenAI, 2022) (*i.e.,* ChatGPT) and GPT-4 (OpenAI, 2023) massively dominated various coding benchmarks and leaderboards (Chen et al., 2021; Austin et al., 2021; Liu et al., 2023b; Lai et al., 2022; Xia & Zhang, 2023). To further push the boundaries of code generation with open source LLMs, SELF-INSTRUCT (Wang et al., 2023a) is adopted to bootstrap the instruction-following ability of LLMs. In the realm of code, practitioners commonly devise synthetic coding instructions using a stronger teacher model (*e.g.,* ChatGPT and GPT-4) and then finetune a weaker student model (*e.g.,* CODELLAMA (Rozière et al., 2023)) with the generated data to distill the knowledge from the teacher (Taori et al., 2023; Chaudhary, 2023). For example, Code Alpaca (Chaudhary, 2023) consists of 20K automatically generated code instructions by applying SELF-INSTRUCT on ChatGPT using 21 seed tasks. To further enhance the coding abilities of LLMs, Luo et al. (2023b) proposes *Code Evol-Instruct* that employs various heuristics to increase the complexity of seed code instructions (Code Alpaca in this case), achieving state-of-the-art (SOTA) results among open-source models.

While these data generation methods can effectively improve the instruction-following capability of an LLM, they rely on a narrow range of predefined tasks or heuristics under the hood. For example, on the one hand, Code Alpaca that adopts SELF-INSTRUCT only relies on *21 seed tasks* to generate new code instructions using an identical prompt template. On the other hand, Code Evol-Instruct takes Code Alpaca as seeds and merely depends on *5 heuristics* to evolve the dataset. As partly suggested by Yu et al. (2023) and Wang et al. (2023a), such approaches may significantly inherit the system bias inherent in the LLMs as well as the predefined tasks.

Therefore, in this paper, we propose OSS-INSTRUCT to mitigate the inherent bias of LLMs and to unleash their potential to craft diverse and creative code instructions via direct learning from the open source. As shown in Figure 1,

[†]The work was done during a remote summer internship at the University of Illinois. [1]University of Illinois at Urbana-Champaign, USA [2]Tsinghua University, China. Correspondence to: Yuxiang Wei <ywei40@illinois.edu>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

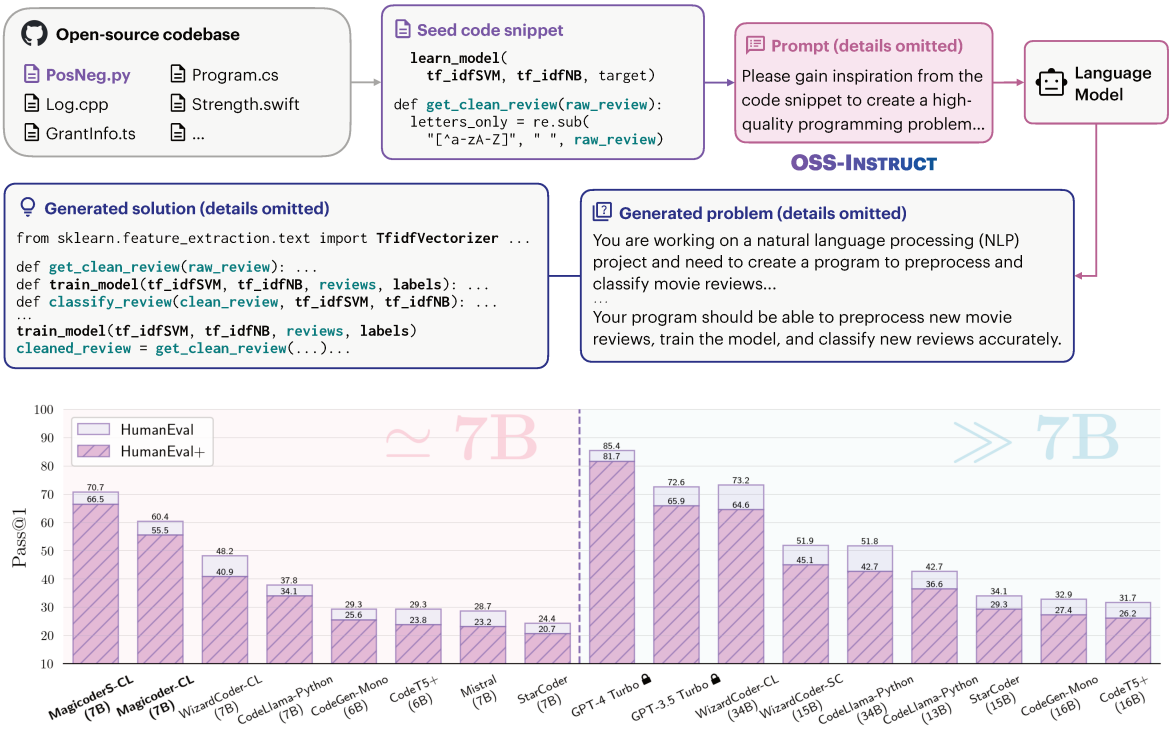

Figure 1: Overview of OSS-INSTRUCT and the pass@1 results of different LLMs on HumanEval (+)

OSS-INSTRUCT leverages a powerful LLM to automatically generate new coding problems by *drawing inspiration* from any random code snippets collected from the open source. In this example, the LLM gets inspired by two incomplete code fragments from different functions and manages to relate them and craft a realistic machine learning problem. Thanks to the "infinite" real-world open-source code, OSS-INSTRUCT can directly produce *diverse*, *realistic*, and *controllable* code instructions by providing distinct seed code snippets. In the end, we generate 75K synthetic data to finetune CODELLAMA-PYTHON-7B, resulting in Magicoder-CL. While being simple and effective, OSS-INSTRUCT is orthogonal to existing data generation methods, and they can be combined to further boost the models' coding capabilities. Therefore, we continually finetune Magicoder-CL on an open-source Evol-Instruct dataset with 110K entries, producing Magicoder$\mathcal{S}$-CL.

We evaluate Magicoder and Magicoder$\mathcal{S}$ on a wide range of coding tasks, including HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) for Python text-to-code generation, MultiPL-E (Cassano et al., 2022) for multilingual code completion, and DS-1000 (Lai et al., 2022) for solving data science problems. We further adopt EvalPlus (Liu et al., 2023b), which includes the augmented HumanEval+ and MBPP+ datasets for more rigorous model evaluation. Both Magicoder-CL and Magicoder$\mathcal{S}$-CL substantially boost the base CODELLAMA-PYTHON-7B. Additionally, Magicoder-CL even outperforms WizardCoder-CL-7B, WizardCoder-SC-15B, and all studied SOTA LLMs with less than or equal to 16B parameters on all the benchmarks we tested. Also, the pass@1 result of the enhanced Magicoder$\mathcal{S}$-CL is on par with ChatGPT on HumanEval (70.7 vs. 72.6) and surpasses it on the more rigorous HumanEval+ (66.5 vs. 65.9), indicating that Magicoder$\mathcal{S}$-CL can generate more robust code. It also achieves SOTA results among all code models at the same scale.

Additionally, we notice a very recent advancement in the development of the DeepSeek-Coder series (Guo et al., 2024) which has shown exceptional coding performance. However, due to the limited technical details disclosed, we only briefly discuss them in §3.4. Despite this, we applied OSS-INSTRUCT on DeepSeek-Coder-Base 6.7B, resulting in the creation of Magicoder-DS and Magicoder$\mathcal{S}$-DS. In addition to the consistent findings on the previous results with CODELLAMA-PYTHON-7B as the base model, Magicoder-DS and Magicoder$\mathcal{S}$-DS benefit from the more powerful DeepSeek-Coder-Base-6.7B. This advantage is demonstrated by Magicoder$\mathcal{S}$-DS, which achieves a remarkable 76.8 pass@1 on HumanEval. Magicoder$\mathcal{S}$-DS also outperforms DeepSeek-Coder-Instruct-6.7B on HumanEval (+) and MBPP (+) with 8× less finetuning tokens.

To justify the design of OSS-INSTRUCT, *i.e.,* generating instruction-tuning data from open-source references rather

than using the references directly, we demonstrate that fine-tuning the base models with semantically relevant comment-function pairs extracted from open-source projects even negatively impacts the model performance (§4.2).

In general, we make the following contributions:

- We introduce OSS-INSTRUCT, a pioneering approach to enlightening LLMs with open-source code snippets to generate more diverse, realistic, and controllable coding instruction data, which can be leveraged to substantially boost the performance of various LLMs via instruction tuning. It opens a new dimension for creating low-bias and diverse instruction-tuning data from the abundance of open-source references.
- We build the Magicoder series trained with OSS-INSTRUCT and Magicoder$\mathcal{S}$ series trained on a combination of OSS-INSTRUCT and Evol-Instruct. Our evaluation across 6 benchmarks shows that all Magicoders significantly improve the base LLMs. Notably, both Magicoder$\mathcal{S}$-CL and Magicoder$\mathcal{S}$-DS outperform ChatGPT on HumanEval+ with only *7B parameters*.
- We fully open source the model weights, training data, and source code at `https://github.com/ise-uiuc/magicoder` to facilitate future research.

2. OSS-INSTRUCT: Instruction Tuning from Open Source

In this section, we elaborate on our OSS-INSTRUCT approach. From a high level, as shown in Figure 1, OSS-INSTRUCT works by prompting an LLM (*e.g.,* ChatGPT) to generate a coding problem and its solution according to some seed code snippet collected from the wild (*e.g.,* from GitHub). The seed snippet offers controllability of the generation and encourages the LLM to create diverse coding problems that can reflect real-world programming scenarios.

2.1. Generating Coding Problems

OSS-INSTRUCT is powered by seed code snippets that can be easily collected from open source. In this work, we directly adopt `starcoderdata` as our seed corpus, a filtered version of The Stack (Kocetkov et al., 2022) dataset that StarCoder is trained on, containing permissively licensed source code documents in various programming languages. We chose `starcoderdata` because it is widely adopted, includes massive high-quality code snippets, and is even post-processed for data decontamination (Li et al., 2023; Allal et al., 2023). For each code document from the corpus, we randomly extract 1–15 consecutive lines as the seed snippet for the model to gain inspiration from and produce coding problems. In total, we collected 80K initial seed snippets from 80K code documents, 40K from Python, and 5K from each of C++, Java, TypeScript, Shell, C#, Rust, PHP, and Swift respectively. Then, each collected seed code snippet is applied to the prompt template shown in Appendix A.1, which a teacher model takes as input and outputs both a coding problem and its solution.

2.2. Data Cleaning and Decontamination

We perform data cleaning by excluding samples that are identical or share the same seed code snippet. While there exist other sorts of noisiness (*e.g.,* the solution is incomplete) in the generated data, inspired by Honovich et al. (2023), they are not removed as we believe they still contain valuable information for LLMs to learn. More experimental details can be found in Appendix C.3. Finally, we apply the same logic as StarCoder Li et al. (2023) to decontaminate our training data by removing coding problems that contain docstrings or solutions from HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), docstrings from APPS (Hendrycks et al., 2021), prompts from DS-1000 (Lai et al., 2022), or questions from GSM8K (Cobbe et al., 2021). As part of our analysis, the decontamination procedure only filters out 9 additional samples. Since the seed corpus `starcoderdata` has already gone through rigorous data decontamination, this observation suggests that OSS-INSTRUCT is unlikely to introduce additional data leakage beyond the seeds. The eventual OSS-INSTRUCT dataset contains about 75K entries. An overview of the dataset statistics can be found in Appendix A.3.

2.3. Qualitative Examples of OSS-INSTRUCT

Figure 2 shows some qualitative examples of how OSS-INSTRUCT can help LLM get inspiration from a seed code snippet to create new coding problems and solutions. For example, the *shell script* example shows how an LLM crafts a Python coding problem with just one line of shell script. The *library imports* example demonstrates how an LLM can create a realistic machine learning problem using just a few import statements. Meanwhile, the *class signature* instance illustrates the ability of LLM to draw inspiration from an incomplete class definition featuring annotations like `SpringBootApplication` and keywords such as `bank`. From this, the LLM generates a problem that requires implementing a complete banking system based on Spring Boot. Overall, OSS-INSTRUCT can inspire an LLM with distinct code structures and semantics to create diverse coding tasks, including algorithmic challenges, realistic issues, single-function code generation, library-based program completion, whole-program development, and even whole-application construction.

Similarity with HumanEval To study whether our data generation process produces more HumanEval-like prob-

Seed: shell script

```
python3 makeErrorFile.py data/test_dataset_14 14
```

Problem

Create a Python program that generates an **error file** based on a given **dataset**...

Code

```
def generate_error_file(dataset_file, ...):
  error_lines = []
  with open(dataset_file, 'r') as file:
    for line in file:
      ...
  with open(error_file_name, 'w') as error_file:
    for error_line in error_lines:
      error_file.write(error_line + '\n')
if __name__ == "__main__":
  if len(sys.argv) != 3:
    print("Usage: ...")
  else:
    dataset_file = sys.argv[1]
    dataset_number = sys.argv[2]
    generate_error_file(...)
```

Seed: library imports

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

Problem

Create a **reinforcement learning agent** to control an electric motor using the OpenAI **Gym** environment...

Code

```
import numpy as np
import gym_electric_motor as gem
...
env = gem.make("DcSeriesCont-v1")
class DQNAgent:
  def __init__(self, state_dim, action_dim): ...
  def build_model(self): ...
  def act(self, state): ...
  def train(self, state, action, reward, ...): ...
...
for episode in range(episodes):
    state = env.reset()
    state = np.reshape(state, [1, state_dim])
    ...
```

Seed: class signature

```
@SpringBootApplication
@Import({ AxonConfig.class })
public class AxonbankApplication {
  public static void main(String[] args) {
```

Problem

Create a simple Java **Spring Boot application** for a **banking system**...

Code

```
import org.axonframework.commandhandling...
import org.axonframework.config...
...
@SpringBootApplication
@Import({ AxonConfig.class })
public class AxonbankApplication {...}
public class BankAccount {...}
public class CreateAccountCommand {...}
public class DepositFundsCommand {...}
public class WithdrawFundsCommand {...}
public class AccountCreatedEvent {...}
public class FundsDepositedEvent {...}
public class FundsWithdrawnEvent {...}
```

Figure 2: Examples showing how OSS-INSTRUCT generates problems and solutions from seed code snippets. Detailed problem requirements, implementations, and explanations are omitted for brevity. More examples can be found in Appendix A.2.

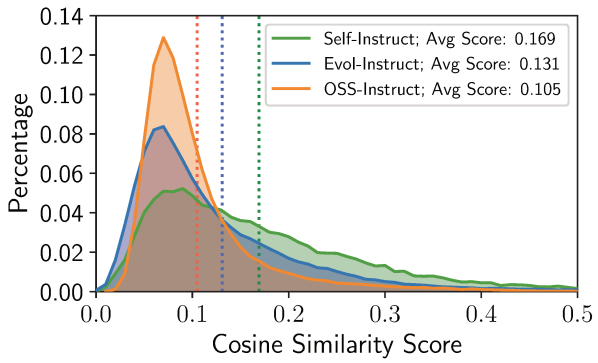

Figure 3: Cosine similarities between HumanEval and synthetic data generated by different methods.

lems or solutions that contribute to high performance, we pair each sample from our 75K dataset with each of the 164 HumanEval (Chen et al., 2021) samples and compute their cosine similarity using TF-IDF (SPARCK JONES, 1972) embeddings. We then associate each OSS-INSTRUCT sample with a HumanEval sample with the highest similarity score. We also compare our dataset against Code Alpaca, a 20K dataset applying SELF-INSTRUCT to code, and `evol-codealpaca-v1` (theblackcat102, 2023), an open-source reproduction of Evol-Instruct containing 110K coding instructions. We resort to the open-source implementation because the official *Code Evol-Instruct* (Luo et al., 2023b) dataset is not released. We decontaminate all the datasets beforehand using the same way discussed in §2.2. Figure 3 shows that OSS-INSTRUCT exhibits the lowest average similarity among all the studied data generation techniques while SELF-INSTRUCT shows the highest average similarity. This result indicates that the improvements from OSS-INSTRUCT are not merely due to including data from the same distribution.

3. Evaluation

We choose CODELLAMA-PYTHON-7B and DeepSeek-Coder-Base 6.7B as the base LLMs. To derive Magicoder series, we first finetune them on 75K synthetic data generated through OSS-INSTRUCT. We then obtain Magicoder$\mathcal{S}$ by continuing finetuning Magicoder with the `evol-codealpaca-v1` dataset, an open-source Evol-Instruct implementation containing about 110K samples. More implementation details and additional evaluation results are listed in Appendices B and C. We also present interesting use cases that reflect the effectiveness of instruction tuning in Appendix D and demonstrate Magicoder's capability to generate complex programs in Appendix E.

3.1. Python Text-to-Code Generation

HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) are two of the most widely used benchmarks for code generation. Each task in these benchmarks includes a task description (*e.g.,* docstring) as the prompt, where LLMs generate corresponding code whose correctness is checked by a handful of test cases. Because tests in these benchmarks can be insufficient, for more rigorous evaluation, we use HumanEval+ and MBPP+, both powered by the EvalPlus framework (Liu et al., 2023b) to obtain 80×/35× more tests. Following prior work (Liu et al., 2023b; Chen et al., 2023), for each task and LLM we use greedy decoding to generate one sample and focus on comparing the pass@1 metric.

We consider a wide range of baseline models, including

Table 1: Pass@1 (%) results of different LLMs on HumanEval (+) and MBPP (+) computed with greedy decoding. The abbreviations "CL" and "SC" refer to the base models CODELLAMA-PYTHON and StarCoder, respectively. We report the results consistently from the EvalPlus (Liu et al., 2023b) Leaderboard.

Model	Release Date	Size	Benchmark		Open-Source	
			HumanEval (+)	MBPP (+)	Weight	Data
GPT-3.5 Turbo	Nov 2023	-	72.6 (65.9)	81.7 (69.4)	○	○
GPT-4 Turbo	Nov 2023	-	**85.4** (**81.7**)	**83.0** (**70.7**)	○	○
CODELLAMA-PYTHON	Aug 2023	34B	51.8 (42.7)	67.2 (52.9)	●	○
WizardCoder-CL	Sep 2023	34B	73.2 (64.6)	73.2 (59.9)	●	○
CodeT5+	May 2023	16B	31.7 (26.2)	54.6 (44.4)	●	●
CodeGen-Mono	Mar 2022	16B	32.9 (27.4)	52.6 (43.6)	●	●
StarCoder	May 2023	15B	34.1 (29.3)	55.1 (46.1)	●	●
CODELLAMA-PYTHON	Aug 2023	13B	42.7 (36.6)	61.2 (50.9)	●	○
WizardCoder-SC	Sep 2023	15B	51.9 (45.1)	61.9 (50.6)	●	○
StarCoder	May 2023	7B	24.4 (20.7)	33.1 (28.8)	●	●
Mistral	Oct 2023	7B	28.7 (23.2)	50.1 (40.9)	●	○
CodeT5+	May 2023	6B	29.3 (23.8)	51.9 (40.9)	●	●
CodeGen-Mono	Mar 2022	6B	29.3 (25.6)	49.9 (42.1)	●	●
CODELLAMA-PYTHON	Aug 2023	7B	37.8 (34.1)	57.6 (45.4)	●	○
WizardCoder-CL	Sep 2023	7B	48.2 (40.9)	56.6 (47.1)	●	○
Magicoder-CL	Dec 2023	7B	60.4 (55.5)	64.2 (52.6)	●	●
Magicoder$\mathcal{S}$-CL	Dec 2023	7B	**70.7** (**66.5**)	**68.4** (**56.6**)	●	●

CODELLAMA-PYTHON (Rozière et al., 2023), WizardCoder (Luo et al., 2023b), GPT-3.5 Turbo (OpenAI, 2022), GPT-4 Turbo (OpenAI, 2023), StarCoder (Li et al., 2023), CodeT5+ (Wang et al., 2023b), CodeGen-Mono (Nijkamp et al., 2023), and Mistral (Jiang et al., 2023a). All the results are consistently reported from the EvalPlus (Liu et al., 2023b) leaderboard (EvalPlus hash: `1895d2f`).

Table 1 shows the pass@1 results of different LLMs on these benchmarks. From the results, we can first observe that Magicoder-CL has a clear improvement over the base CODELLAMA-PYTHON-7B, and outperforms all studied open-source models except CODELLAMA-PYTHON-34B and WizardCoder-CL-34B. Notably, Magicoder-CL surpasses WizardCoder-SC-15B and has a substantial improvement on HumanEval and HumanEval+ over CODELLAMA-PYTHON-34B. Magicoder$\mathcal{S}$-CL demonstrates further improvements by being trained with the orthogonal Evol-Instruct method. Magicoder$\mathcal{S}$-CL outperforms ChatGPT and all other open-source models on HumanEval+. Moreover, although it scores slightly lower than WizardCoder-CL-34B and ChatGPT on HumanEval, it surpasses both of them on the more rigorous HumanEval+ dataset, indicating that Magicoder$\mathcal{S}$-CL may produce more robust code.

3.2. Multilingual Code Generation

In addition to Python, as shown in Table 2, we perform an extensive evaluation on 6 widely used programming languages, *i.e.,* Java, JavaScript, C++, PHP, Swift, and Rust, using the MultiPL-E benchmark (Cassano et al., 2022). We report available results from the WizardCoder paper (Luo et al., 2023b) and evaluate our models consistently through `bigcode-evaluation-harness` (Ben Allal et al., 2022). We skip proprietary models such as ChatGPT and GPT-4 as they are not supported by the framework. Due to a significant inference latency when running WizardCoder-CL-7B using the harness in our environment, we choose not to include it in our analysis.

The results indicate that Magicoder-CL improves the base CODELLAMA-PYTHON-7B by a large margin among all the studied programming languages. Moreover, Magicoder-CL also achieves better results than the SOTA 15B WizardCoder-SC among half of the programming languages. Additionally, Magicoder$\mathcal{S}$-CL demonstrates further improvement over Magicoder-CL on all programming languages, achieving comparable performance against WizardCoder-CL-34B with only 7B parameters. It is worth noting that Magicoder-CL is only trained with very limited multilingual data but still outperforms other LLMs with similar or even larger sizes. Also, although the harness

evaluates models in *completion* formats which are for base models, Magicoders still show significant improvements despite being only *instruction-tuned*. This implies that LLMs can learn knowledge from the data beyond its format.

3.3. Code Generation for Data Science

The DS-1000 dataset (Lai et al., 2022) contains 1K distinct data science coding issues ranging from 7 popular data science libraries in Python. It evaluates the realistic and practical use case of an LLM and offers unit tests for validating each problem. DS-1000 has both *completion* and *insertion* modes, but here we only evaluate *completion* because the base CODELLAMA-PYTHON does not support infilling. Table 3 shows the evaluation results where we include the recent INCODER (Fried et al., 2023), CodeGen (Nijkamp et al., 2023), Code-Cushman-001 (Microsoft, 2023a), StarCoder (Li et al., 2023), CODELLAMA-PYTHON (Rozière et al., 2023), and WizardCoder (Luo et al., 2023b). We can see from the table that Magicoder-CL-7B already outperforms all the baselines we evaluate, including state-of-the-art WizardCoder-CL-7B and WizardCoder-SC-15B. Magicoder$\mathcal{S}$-CL-7B further breaks the limit by introducing an 8.3 percentage point absolute improvement over WizardCoder-SC-15B.

3.4. Comparison with DeepSeek-Coder

DeepSeek-Coder (Guo et al., 2024) is a series of models released concurrently to our work and they demonstrate superior coding performance. We only briefly discuss it in this section because its data and instruction tuning details are not publicly available at the time of writing. We apply the same finetuning strategy on DeepSeek-Coder-Base-6.7B as we performed on CODELLAMA-PYTHON-7B, leading to Magicoder-DS and Magicoder$\mathcal{S}$-DS. Table 4 shows a similar trend as Table 1 that the base model can be significantly improved after applying OSS-INSTRUCT. Remarkably, the Magicoder$\mathcal{S}$-DS variant surpasses DeepSeek-Coder-Instruct-6.7B on all the benchmarks with $\times 8$ fewer training tokens, and it also closely matches DeepSeek-Coder-Instruct-33B on these datasets.

4. Ablations of Data Source

4.1. Impact of the Language Distribution

To understand the correlation between the programming languages appearing in the training data and the downstream performance of different languages, we conduct an additional ablation study about the training data. We classify the 75K training data into approximately 43K Python-only, and 32K non-Python data according to whether ```` ```python ```` is a substring of the generated data. We do not classify the data based on the seed code snippet because LLMs performing OSS-INSTRUCT may produce code in a different programming language than the seed.

Table 5 shows the evaluation results, where we consistently finetune the base CODELLAMA-PYTHON-7B for 2 epochs on different data partitions using the same training hyperparameters explained in Appendix B. From the table, we can see that, as can be imagined, training on Python or non-Python data can substantially boost the performance of the base model in Python or non-Python tasks, respectively. Interestingly, instruction tuning on different programming languages can still boost the overall coding performance that includes out-of-distribution languages. For example, when trained on only non-Python data, Magicoder-CL still achieves a 10.4 percentage point improvement over the base model in the Python-only evaluation. This implies LLMs can establish correlations between different programming languages and perform transfer learning of deeper code semantics. Finally, we observe a more significant boost in Python evaluation when combining data from both sources, with a slight decrease in multilingual performance compared with only finetuning on multilingual data. We attribute this decrease to the dominant amount of Python data (around 57%) during instruction tuning.

4.2. OSS-INSTRUCT vs. Direct Finetuning

The fact that OSS-INSTRUCT gets an LLM inspired from open-source code snippets may lead to a natural question: why not directly finetuning on these open-source code? To answer this question, we follow CodeSearchNet (Husain et al., 2020) to mine semantically relevant *comment-function* pairs from the same seed document corpus we use to construct the 75K OSS-INSTRUCT dataset. We then train the model to predict the function bodies from the function signatures and comments. We prioritize comment-function pairs that overlap with our *75K seed snippets*, resulting in about 11K data points. To align with our 75K samples, we collect the remaining 64K samples using the whole corpus of *75K seed documents*. Eventually, we have the same number of comment-function pairs with OSS-INSTRUCT data.

We finetune the base CODELLAMA-PYTHON-7B for 2 epochs using the paired data, following the same training setup discussed in Appendix B. From Table 6, we observe that finetuning on 75K paired comment-function data even worsens the base model, while OSS-INSTRUCT helps to introduce a substantial boost. We conjecture that the degradation is owing to the substantial noise and inconsistency that exists intrinsically in the data pairs, even though these paired data exhibit *very similar* format as HumanEval or MultiPL-E problems. This further shows that data factuality, rather than the format, is essential to code instruction tuning. It also indicates the superiority of OSS-INSTRUCT which can translate these loosely related code fragments

Table 2: Pass@1 results of different LLMs on MultiPL-E (Cassano et al., 2022) following the same hyperparameter settings as the WizardCoder paper (Luo et al., 2023b): `temperature` $= 0.2$, `top_p` $= 0.95$, `max_length` $= 512$, and `num_samples` $= 50$. We evaluate all 7B models using `bigcode-evaluation-harness` (Ben Allal et al., 2022) and report other results from WizardCoder.

Model	Size	Programming Language					
		Java	JavaScript	C++	PHP	Swift	Rust
CODELLAMA	34B	40.2	41.7	41.4	40.4	35.3	38.7
CODELLAMA-PYTHON	34B	39.5	44.7	39.1	39.8	34.3	39.7
CODELLAMA-INSTRUCT	34B	41.5	45.9	41.5	37.0	37.6	39.3
WizardCoder-CL	34B	**44.9**	**55.3**	**47.2**	**47.2**	**44.3**	**46.2**
StarCoderBase	15B	28.5	31.7	30.6	26.8	16.7	24.5
StarCoder	15B	30.2	30.8	31.6	26.1	22.7	21.8
WizardCoder-SC	15B	35.8	41.9	39.0	39.3	33.7	27.1
CODELLAMA	7B	29.3	31.7	27.0	25.1	25.6	25.5
CODELLAMA-PYTHON	7B	29.1	35.7	30.2	29.0	27.1	27.0
Magicoder-CL	7B	36.4	45.9	36.5	39.5	33.4	30.6
Magicoder$\mathcal{S}$-CL	7B	**42.9**	**57.5**	**44.4**	**47.6**	**44.1**	**40.3**

Table 3: Pass@1 results on DS-1000 (completion format) with `temperature` $= 0.2$, `top_p` $= 0.5$, `max_length` $= 1024$, and `num_samples` $= 40$, following the same hyperparameter setting used in WizardCoder (Luo et al., 2023b). We evaluate all the 7B models with their preferred prompt formats and report other results from WizardCoder.

Model	Size	+ 155 Matplotlib	+ 220 NumPy	+ 291 Pandas	+ 68 PyTorch	+ 106 SciPy	+ 115 Sklearn	+ 45 TensorFlow	= 1000 Overall
INCODER	6.7B	28.3	4.4	3.1	4.4	2.8	2.8	3.8	7.4
CodeGen-Mono	16B	31.7	10.9	3.4	7.0	9.0	10.8	15.2	11.7
Code-Cushman-001	-	40.7	21.8	7.9	12.4	11.3	18.0	12.2	18.1
StarCoder	15B	51.7	29.7	11.4	21.4	20.2	29.5	24.5	26.0
WizardCoder-SC	15B	55.2	33.6	16.7	26.2	24.2	24.9	26.7	29.2
CODELLAMA-PYTHON	7B	55.3	34.5	16.4	19.9	22.3	17.6	28.5	28.0
WizardCoder-CL	7B	53.5	34.4	15.2	25.7	21.0	24.5	28.9	28.4
Magicoder-CL	7B	54.6	34.8	19.0	24.7	25.0	22.6	28.9	29.9
Magicoder$\mathcal{S}$-CL	7B	**55.9**	**40.6**	**28.4**	**40.4**	**28.8**	**35.8**	**37.6**	**37.5**

Table 4: Pass@1 (greedy decoding) comparison between Magicoder and DeepSeek-Coder (Guo et al., 2024) on HumanEval (+) and MBPP (+). DeepSeek-Coder results are reported from EvalPlus (Liu et al., 2023b) Leaderboard.

Model	Size	Training Tokens	Benchmark		Open-Source	
			HumanEval (+)	MBPP (+)	Weight	Data
DeepSeek-Coder-Base	1.3B	2T	-	55.4 (46.9)	●	○
	6.7B	2T	47.6 (39.6)	70.2 (56.6)	●	○
	33B	2T	51.2 (43.3)	-	●	○
DeepSeek-Coder Instruct	1.3B	+2B	64.6 (58.5)	63.7 (53.1)	●	○
	6.7B	+2B	**73.8** (**70.1**)	**72.7** (**63.4**)	●	○
	33B	+2B	**78.7** (**72.6**)	**78.7** (**66.7**)	●	○
Magicoder-DS	6.7B	+90M	66.5 (60.4)	75.4 (61.9)	●	●
Magicoder$\mathcal{S}$-DS	6.7B	+240M	**76.8** (**70.7**)	**75.7** (**64.4**)	●	●

Table 5: Ablation study of using different programming languages as training data. We show the pass@1 results on HumanEval+ (Liu et al., 2023b) for Python and the average pass@1 results on MultiPL-E (Cassano et al., 2022) for the same set of programming languages used in Table 2 (*i.e.,* Java, JavaScript, C++, PHP, Swift, and Rust). All the variants are finetuned with 2 epochs and evaluated through greedy-decoding.

Model (7B)	Finetuning Data	Python (HumanEval+)	Others (MultiPL-E)
CODELLAMA-PYTHON	-	34.1	29.6
Magicoder-CL	Python (43K)	47.6	32.7
Magicoder-CL	Others (32K)	44.5	**38.3**
Magicoder-CL	Both (75K)	**55.5**	37.8

Table 6: Comparison between OSS-INSTRUCT and directly finetuning on comment-function pairs with CODELLAMA-PYTHON-7B as the base model.

Finetuning Data	HumanEval+	MultiPL-E
Base model w/o finetuning	34.1	29.6
Comment-function pairs (75K)	34.1	24.1
OSS-INSTRUCT (75K)	**55.5**	**37.8**

into semantically-consistent instruction-tuning data.

4.3. OSS-INSTRUCT with A Less Powerful Teacher

In this section, we explore the factors contributing to the effectiveness of OSS-INSTRUCT beyond just the distillation of the teacher model. We propose two potential key reasons. First, since the base model is pretrained with comprehensive code data, the distillation process likely activates the model's internal capabilities, leading to improved performance in coding tasks. Second, OSS-INSTRUCT uses seed code snippets to generate problem-solution pairs in one shot. These seed snippets provide valuable context, enabling the model to create better solutions than a plain teacher model lacking such seed information. These enhanced solutions can then be used to train more effective student models. To verify these points, we conduct an additional experiment by generating a subset of 20K OSS-INSTRUCT data using Mixtral-8x7B-Instruct-v0.1 (Jiang et al., 2024), a state-of-the-art, general-purpose, open-source LLM.

Table 7: Pass@1 on HumanEval+ and MBPP+ when finetuning CODELLAMA-PYTHON-7B for 2 epochs on 20K OSS-INSTRUCT data generated by Mixtral-8x7B-Instruct-v0.1 (Jiang et al., 2024).

Model	HumanEval+	MBPP+
Mixtral-8x7B-Instruct-v0.1	39.6	47.4
CODELLAMA-PYTHON-7B	34.1	45.4
Magicoder-CL-Mixtral-7B	**55.5**	**50.4**

Table 7 indicates that Magicoder-CL-Mixtral-7B not only significantly improves over the base CODELLAMA-PYTHON, but is also better than Mixtral-8x7B-Instruct-v0.1 (*i.e.,* the teacher model) across HumanEval+ and MBPP+. These results suggest that OSS-INSTRUCT is not simply distilling a teacher model, but also triggering the base model's own capability and effectively leveraging the information encapsulated in seed code snippets.

5. Related Work

Foundation models for code Trained over billions of lines of code, LLMs have demonstrated outstanding performance in a wide range of software engineering tasks, including code generation (Chen et al., 2021; Austin et al., 2021), program repair (Xia & Zhang, 2022; Wei et al., 2023; Xia et al., 2023b; Jiang et al., 2023b; Bouzenia et al., 2024), and software testing (Xia et al., 2023a; Deng et al., 2023; Yuan et al., 2023; Schäfer et al., 2023; Lemieux et al., 2023). In particular, prominent base models, such as CodeGen (Nijkamp et al., 2023), CodeT5 (Wang et al., 2021), StarCoder (Li et al., 2023), and CODELLAMA (Rozière et al., 2023), are pre-trained over a huge number of codebase from scratch, establishing the fundamental ability of general code generation and understanding. More recent code LLMs, such as DeepSeek-Coder (Guo et al., 2024) and StarCoder2 (Lozhkov et al., 2024), additionally organize the pretraining data at the repository level to enhance the model's contextual understanding capabilities. Furthermore, these base models are also finetuned (Luo et al., 2023b) or prompted (Chen et al., 2023) to unlock their true potential to specialize in solving domain-specific coding tasks.

Instruction tuning with synthetic data Instruction tuning aims to improve pretrained LLMs by finetuning them with a mixture of instructions and corresponding responses (Wei et al., 2022). However, obtaining high-quality instructional data is oftentimes laborious. Hence, researchers are increasingly focusing on the development of methods to generate synthetic instruction data. Wang et al. (2023a) introduces SELF-INSTRUCT, where a founda-

tion LLM (GPT-3 (Brown et al., 2020)) is used to generate synthetic instruction-response pairs with carefully crafted prompts. The *same* LLM is then instruction-tuned on the synthetic data to distill such self-generated knowledge. This technique has been further extended to create synthetic data with different LLMs. For example, Alpaca (Taori et al., 2023) and Code Alpaca (Chaudhary, 2023) apply SELF-INSTRUCT to finetune LLAMA with ChatGPT-generated instructions. To improve SELF-INSTRUCT, WizardLM (Xu et al., 2023) and WizardCoder (Luo et al., 2023a) propose Evol-Instruct and Code Evol-Instruct by guiding ChatGPT with heuristic prompts to make the synthetic data more complex and diverse. More recently, Gunasekar et al. (2023) shows that textbook-quality synthetic data alone can help the model achieve remarkable coding and reasoning capabilities. Orthogonal to all existing methods, our proposed OSS-INSTRUCT allows LLMs to get inspired from real-world code snippets for better controllability, quality, and creativity in coding tasks.

Evaluating LLMs for code Most code benchmarks evaluate LLMs on generating *single-function* programs from natural language descriptions. Such benchmarks include HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), APPS (Hendrycks et al., 2021), and CodeContests (Li et al., 2022). A handful of manual tests are used to assess the functional correctness of LLM-generated solutions. However, insufficient tests can lead to false negatives. Consequently, the EvalPlus framework (Liu et al., 2023b) produces HumanEval+ and MBPP+ by extending 80×/35× more tests. To address dataset contamination issues, researchers propose LiveCodeBench (Jain et al., 2024), which compiles fresh coding problems not included in model training, and EvoEval (Xia et al., 2024), which strategically leverages LLMs to evolve existing benchmarks into new coding tasks. Meanwhile, there are comprehensive benchmarks evaluating code generation for data science (DS-1000 (Lai et al., 2022)), addressing open-source issues (SWE-bench (Jimenez et al., 2023)), and repository-level code generation (CROSSCODEEVAL (Ding et al., 2023) and RepoEval (Zhang et al., 2023)).

6. Conclusion and Future Work

We propose OSS-INSTRUCT, a novel data generation method using Large Language Models to generate diverse coding challenges from open-source code snippets. This approach enables Magicoder, which significantly improves the base LLM. Despite having less than 7B parameters, it can outperform all evaluate LLMs with less than or equal to 16B parameters, including the 15B WizardCoder. Combining OSS-INSTRUCT with Evol-Instruct allows us to build the enhanced Magicoder$\mathcal{S}$ models. They achieve remarkable results by rivaling leading models like ChatGPT in HumanEval benchmarks. We fully open source the model weights, training data, and source code, to enable future research in LLMs for code. In the near future, we will apply OSS-INSTRUCT to larger base models. We will also continue advancing OSS-INSTRUCT by generating higher-quality data with a strategically designed distribution of the seed code snippets and with more advanced teacher LLMs such as GPT-4.

Acknowledgement

We thank all the reviewers for their insightful comments and suggestions for our paper. This work was partially supported by NSF grant CCF-2131943, as well as Kwai Inc.

Impact Statement

This work is motivated to boost large language models in terms of their code generation and understanding capabilities through instruction tuning. The proposed OSS-INSTRUCT method leverages the abundance of open source to generate diverse and controllable instruction data. We expect this idea to also foster innovative software solutions tailored to domain-specific needs, particularly in areas where real data is private and scarce, by generating extensive synthetic data. Additionally, our method reinforces the value of community-driven content and knowledge sharing by incorporating open-source code as references.

However, it is essential to recognize the potential for misuse, such as the deliberate generation of vulnerable code that can be exploited for malicious purposes. Ultimately, adhering to ethical guidelines is crucial to ensure the responsible use of this technique.

References

Allal, L. B., Li, R., Kocetkov, D., Mou, C., Akiki, C., Ferrandis, C. M., Muennighoff, N., Mishra, M., Gu, A., Dey, M., Umapathi, L. K., Anderson, C. J., Zi, Y., Poirier, J. L., Schoelkopf, H., Troshin, S., Abulkhanov, D., Romero, M., Lappert, M., Toni, F. D., del Río, B. G., Liu, Q., Bose, S., Bhattacharyya, U., Zhuo, T. Y., Yu, I., Villegas, P., Zocca, M., Mangrulkar, S., Lansky, D., Nguyen, H., Contractor, D., Villa, L., Li, J., Bahdanau, D., Jernite, Y., Hughes, S., Fried, D., Guha, A., de Vries, H., and von Werra, L. Santacoder: don't reach for the stars!, 2023.

Austin, J., Odena, A., Nye, M. I., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C. J., Terry, M., Le, Q. V., and Sutton, C. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732.

Ben Allal, L., Muennighoff, N., Kumar Umapathi,

L., Lipkin, B., and von Werra, L. A framework for the evaluation of code generation models. https://github.com/bigcode-project/bigcode-evaluation-harness, 2022.

Bouzenia, I., Devanbu, P., and Pradel, M. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134*, 2024.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

Cambronero, J., Gulwani, S., Le, V., Perelman, D., Radhakrishna, A., Simon, C., and Tiwari, A. Flashfill++: Scaling programming by example by cutting to the chase. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. doi: 10.1145/3571226. URL https://doi.org/10.1145/3571226.

Cassano, F., Gouwar, J., Nguyen, D., Nguyen, S., Phipps-Costin, L., Pinckney, D., Yee, M.-H., Zi, Y., Anderson, C. J., Feldman, M. Q., Guha, A., Greenberg, M., and Jangda, A. Multipl-e: A scalable and extensible approach to benchmarking neural code generation, 2022.

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

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code, 2021.

Chen, X., Lin, M., Schärli, N., and Zhou, D. Teaching large language models to self-debug, 2023.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems, 2021.

Deng, Y., Xia, C. S., Peng, H., Yang, C., and Zhang, L. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models, 2023.

Ding, Y., Wang, Z., Ahmad, W. U., Ding, H., Tan, M., Jain, N., Ramanathan, M. K., Nallapati, R., Bhatia, P., Roth, D., and Xiang, B. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. URL https://openreview.net/forum?id=wgDcbBMSfh.

Feng, Y., Martins, R., Bastani, O., and Dillig, I. Program synthesis using conflict-driven learning. *SIGPLAN Not.*, 53(4):420–435, jun 2018. ISSN 0362-1340. doi: 10.1145/3296979.3192382. URL https://doi.org/10.1145/3296979.3192382.

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, S., Zettlemoyer, L., and Lewis, M. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=hQwb-lbM6EL.

Gulwani, S., Polozov, O., and Singh, R. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017. ISSN 2325-1107. doi: 10.1561/2500000010. URL http://dx.doi.org/10.1561/2500000010.

Gunasekar, S., Zhang, Y., Aneja, J., Mendes, C. C. T., Giorno, A. D., Gopi, S., Javaheripi, M., Kauffmann, P., de Rosa, G., Saarikivi, O., Salim, A., Shah, S., Behl, H. S., Wang, X., Bubeck, S., Eldan, R., Kalai, A. T., Lee, Y. T., and Li, Y. Textbooks are all you need, 2023.

Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y. K., Luo, F., Xiong, Y., and Liang, W. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and Steinhardt, J. Measuring coding challenge competence with apps, 2021.

Honovich, O., Scialom, T., Levy, O., and Schick, T. Unnatural instructions: Tuning language models with (almost) no human labor. In Rogers, A., Boyd-Graber, J., and Okazaki, N. (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14409–14428, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.806. URL https://aclanthology.org/2023.acl-long.806.

Hugging Face. Hugging face: The ai community building the future. https://huggingface.co/, 2023. Accessed: 2023-12-01.

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. Codesearchnet challenge: Evaluating the state of semantic code search, 2020.

Jain, N., Han, K., Gu, A., Li, W.-D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., and Stoica, I. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., de las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L. R., Lachaux, M.-A., Stock, P., Scao, T. L., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. Mistral 7b, 2023a.

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., de las Casas, D., Hanna, E. B., Bressand, F., Lengyel, G., Bour, G., Lample, G., Lavaud, L. R., Saulnier, L., Lachaux, M.-A., Stock, P., Subramanian, S., Yang, S., Antoniak, S., Scao, T. L., Gervet, T., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. Mixtral of experts, 2024.

Jiang, N., Liu, K., Lutellier, T., and Tan, L. Impact of code language models on automated program repair, 2023b.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues?, 2023.

Kocetkov, D., Li, R., Allal, L. B., Li, J., Mou, C., Ferrandis, C. M., Jernite, Y., Mitchell, M., Hughes, S., Wolf, T., Bahdanau, D., von Werra, L., and de Vries, H. The stack: 3 tb of permissively licensed source code, 2022.

Lai, Y., Li, C., Wang, Y., Zhang, T., Zhong, R., Zettlemoyer, L., tau Yih, S. W., Fried, D., Wang, S., and Yu, T. Ds-1000: A natural and reliable benchmark for data science code generation, 2022.

Lemieux, C., Inala, J. P., Lahiri, S. K., and Sen, S. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 919–931. IEEE, 2023.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhuo, T. Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., Gontier, N., Meade, N., Zebaze, A., Yee, M.-H., Umapathi, L. K., Zhu, J., Lipkin, B., Oblokulov, M., Wang, Z., Murthy, R., Stillerman, J., Patel, S. S., Abulkhanov, D., Zocca, M., Dey, M., Zhang, Z., Fahmy, N., Bhattacharyya, U., Yu, W., Singh, S., Luccioni, S., Villegas, P., Kunakov, M., Zhdanov, F., Romero, M., Lee, T., Timor, N., Ding, J., Schlesinger, C., Schoelkopf, H., Ebert, J., Dao, T., Mishra, M., Gu, A., Robinson, J., Anderson, C. J., Dolan-Gavitt, B., Contractor, D., Reddy, S., Fried, D., Bahdanau, D., Jernite, Y., Ferrandis, C. M., Hughes, S., Wolf, T., Guha, A., von Werra, L., and de Vries, H. Starcoder: may the source be with you!, 2023.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D. J., Sutherland Robson, E., Kohli, P., de Freitas, N., Kavukcuoglu, K., and Vinyals, O. Competition-level code generation with alphacode. *Science*, 378 (6624):1092–1097, December 2022. ISSN 1095-9203. doi: 10.1126/science.abq1158. URL http://dx.doi.org/10.1126/science.abq1158.

Liu, J., Peng, J., Wang, Y., and Zhang, L. Neuri: Diversifying dnn generation via inductive rule inference. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, pp. 657–669, New York, NY, USA, 2023a. Association for Computing Machinery. ISBN 9798400703270. doi: 10.1145/3611643.3616337. URL https://doi.org/10.1145/3611643.3616337.

Liu, J., Xia, C. S., Wang, Y., and Zhang, L. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023b. URL https://openreview.net/forum?id=1qvx610Cu7.

Lozhkov, A., Li, R., Allal, L. B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y., Liu, T., Tian, M., Kocetkov, D., Zucker, A., Belkada, Y., Wang, Z., Liu, Q., Abulkhanov, D., Paul, I., Li, Z., Li, W.-D., Risdal, M., Li, J., Zhu, J., Zhuo, T. Y., Zheltonozhskii, E., Dade, N. O. O., Yu, W., Krauß, L., Jain, N., Su, Y., He, X., Dey, M., Abati, E., Chai, Y., Muennighoff, N., Tang, X., Oblokulov, M., Akiki, C., Marone, M., Mou, C., Mishra, M., Gu, A., Hui, B., Dao, T., Zebaze, A., Dehaene, O., Patry, N., Xu, C., McAuley, J., Hu, H.,

Scholak, T., Paquet, S., Robinson, J., Anderson, C. J., Chapados, N., Patwary, M., Tajbakhsh, N., Jernite, Y., Ferrandis, C. M., Zhang, L., Hughes, S., Wolf, T., Guha, A., von Werra, L., and de Vries, H. Starcoder 2 and the stack v2: The next generation, 2024.

Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., and Jiang, D. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023a.

Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., and Jiang, D. Wizardcoder: Empowering code large language models with evol-instruct, 2023b.

Microsoft. Azure openai service models. https://learn.microsoft.com/en-us/azure/cognitive-services/openai/concepts/models, 2023a.

Microsoft. GitHub Copilot – Your AI pair programmer. https://github.com/features/copilot, 2023b.

Muennighoff, N., Liu, Q., Zebaze, A., Zheng, Q., Hui, B., Zhuo, T. Y., Singh, S., Tang, X., von Werra, L., and Longpre, S. Octopack: Instruction tuning code large language models, 2023.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=iaYcJKpY2B_.

Olausson, T. X., Inala, J. P., Wang, C., Gao, J., and Solar-Lezama, A. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=y0GJXRungR.

OpenAI. Chatgpt: Optimizing language models for dialogue. https://openai.com/blog/chatgpt/, 2022.

OpenAI. Gpt-4 technical report, 2023.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. Code llama: Open foundation models for code, 2023.

Schäfer, M., Nadi, S., Eghbali, A., and Tip, F. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 2023.

Services, A. W. AI Code Generator - Amazon CodeWhisperer - AWS. https://aws.amazon.com/codewhisperer/, 2023.

Shazeer, N. and Stern, M. Adafactor: Adaptive learning rates with sublinear memory cost, 2018.

SPARCK JONES, K. A statistical interpretation of term specificity and its application in retrieval. 28(1):11–21, 2023/11/30 1972. doi: 10.1108/eb026526. URL https://doi.org/10.1108/eb026526.

Su, H., Shi, W., Kasai, J., Wang, Y., Hu, Y., Ostendorf, M., Yih, W.-t., Smith, N. A., Zettlemoyer, L., and Yu, T. One embedder, any task: Instruction-finetuned text embeddings. 2022. URL https://arxiv.org/abs/2212.09741.

Taori, R., Gulrajani, I., Zhang, T., Dubois, Y., Li, X., Guestrin, C., Liang, P., and Hashimoto, T. B. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

theblackcat102. The evolved code alpaca dataset. https://huggingface.co/datasets/theblackcat102/evol-codealpaca-v1, 2023.

Wang, X., Dillig, I., and Singh, R. Program synthesis using abstraction refinement. *Proc. ACM Program. Lang.*, 2 (POPL), dec 2017. doi: 10.1145/3158151. URL https://doi.org/10.1145/3158151.

Wang, Y., Wang, W., Joty, S., and Hoi, S. C. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Moens, M.-F., Huang, X., Specia, L., and Yih, S. W.-t. (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.685. URL https://aclanthology.org/2021.emnlp-main.685.

Wang, Y., Kordi, Y., Mishra, S., Liu, A., Smith, N. A., Khashabi, D., and Hajishirzi, H. Self-instruct: Aligning language models with self-generated instructions. In Rogers, A., Boyd-Graber, J., and Okazaki, N. (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13484–13508, Toronto, Canada, July 2023a. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.754. URL https://aclanthology.org/2023.acl-long.754.

Wang, Y., Le, H., Gotmare, A. D., Bui, N. D. Q., Li, J., and Hoi, S. C. H. Codet5+: Open code large language models for code understanding and generation, 2023b.

Wei, J., Bosma, M., Zhao, V. Y., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M., and Le, Q. V. Finetuned language models are zero-shot learners, 2022.

Wei, Y., Xia, C. S., and Zhang, L. Copiloting the copilots: Fusing large language models with completion engines for automated program repair, 2023.

Xia, C. S. and Zhang, L. Less training, more repairing please: Revisiting automated program repair via zero-shot learning, 2022.

Xia, C. S. and Zhang, L. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385*, 2023.

Xia, C. S., Paltenghi, M., Tian, J. L., Pradel, M., and Zhang, L. Universal fuzzing via large language models, 2023a.

Xia, C. S., Wei, Y., and Zhang, L. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1482–1494, 2023b. doi: 10.1109/ICSE48619.2023.00129.

Xia, C. S., Deng, Y., and Zhang, L. Top leaderboard ranking = top coding proficiency, always? evoeval: Evolving coding benchmarks via llm, 2024.

Xu, C., Sun, Q., Zheng, K., Geng, X., Zhao, P., Feng, J., Tao, C., and Jiang, D. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244*, 2023.

Yu, Y., Zhuang, Y., Zhang, J., Meng, Y., Ratner, A., Krishna, R., Shen, J., and Zhang, C. Large language model as attributed training data generator: A tale of diversity and bias, 2023.

Yuan, Z., Lou, Y., Liu, M., Ding, S., Wang, K., Chen, Y., and Peng, X. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207*, 2023.

Zhang, F., Chen, B., Zhang, Y., Keung, J., Liu, J., Zan, D., Mao, Y., Lou, J.-G., and Chen, W. Repocoder: Repository-level code completion through iterative retrieval and generation, 2023.