How Do Your Code LLMs perform? Empowering Code Instruction Tuning with Really Good Data

Anonymous ACL submission

Abstract

Recently, there has been a growing interest in studying how to construct better code instruction tuning data. However, we find Code models trained with these datasets exhibit high performance on HumanEval but perform worse on other benchmarks such as LiveCodeBench. Upon further investigation, we discover many datasets suffer from significant data leakage. After cleaning up most of the leaked data, we find that some datasets previously considered high-quality perform poorly. This discovery reveals a new challenge: identifying which dataset genuinely qualify as high-quality code instruction data. To address this, we propose an efficient code data selection strategy for selecting samples. Our approach is based on three dimensions: instruction complexity, response quality, and instruction diversity. Based on our selected data, we present XCoder, a family of models finetuned from LLaMA3. Experiments show Xcoder achieves new state-of-the-art performance using fewer training data, which verify the effectiveness of our data strategy. Moreover, we perform a comprehensive analysis on the data composition and find existing code datasets have different characteristics according to their construction methods, which provide new insights for future code LLMs.

1 Introduction

Code pre-trained models have achieved remarkable progress in the era of large language models (LLMs), such as Codex (Chen et al., 2021b), AlphaCode (Li et al., 2022), PaLM-Coder (Chowdhery et al., 2022) and StarCoder (Li et al., 2023). Training on large code corpora (Kocetkov et al., 2022) has been shown to enhance the coding capabilities of current LLMs (Lozhkov et al., 2024b; Rozière et al., 2023). In addition to costly pre-training, recent research has garnered increased interest in code instruction tuning and obtains promising results on several code benchmarks (Chaudhary, 2023; Luo et al., 2023a; Wei et al., 2023a; Muennighoff et al., 2023a; Wang et al., 2024).

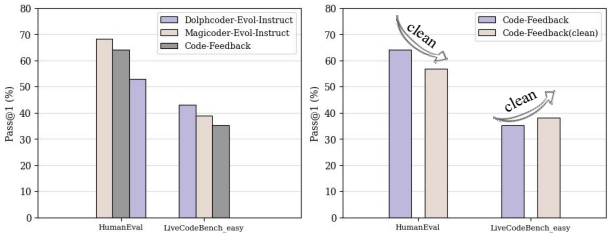

Figure 1: The left figure shows performance comparison on different benchmarks and the right displays varying results after data decontamination. Magicoder Evol-Instruct and Code-Feedback may have data leakage on HumanEval.

Differing from the high demand of pre-training for data quantity, instruction tuning aligns existing model abilities towards a desired direction using high-quality but much smaller datasets. To construct code instruction datasets, earlier research predominantly relies on heuristic automation (e.g. distillation from ChatGPT) or manual selection. For example, Code Alpaca (Chaudhary, 2023) and WizardCoder (Luo et al., 2023a) use distillation signals from ChatGPT via self-instruct and evol-instruct. Other methods such as OctoPack (Muennighoff et al., 2023a) and Magicoder (Wei et al., 2023a) construct code instructions from pre-training code corpora. Although these code instruction datasets seem excellent on popular code benchmarks like HumanEval[1], we find some of them dramatically drop on another contamination-free benchmark LiveCodeBench (Jain et al., 2024b) which continuously collects new problems over time from online contests. As shown in Figure 1, Magicoder Evol-Instruct and Code-Feedback (Zheng et al., 2024) achieve top ranks on HumanEval but drop on LiveCodeBench. We perform a further decontamination process and find that several existing code models achieve abnormally high performance

[1] https://github.com/openai/human-eval

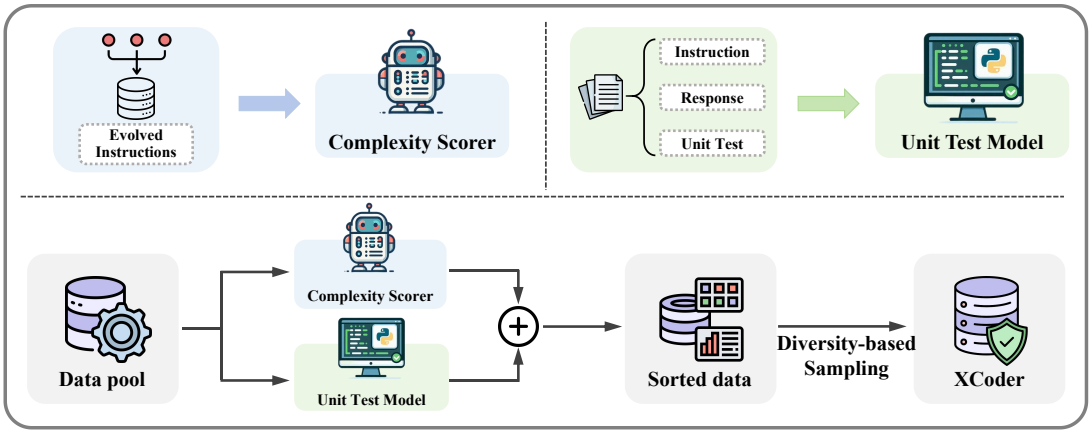

Figure 2: Illustration of our data selection approach.

on HumanEval because of the potential use of the benchmark or benchmark-similar data. Thus, it remains unclear what good code instruction data is and how these datasets actually work. Besides, all the data come from different pipelines and have no unified principle to ensure good quality. We need to systematically define what constitutes good examples of data for code instruction tuning and establish an effective principle for achieving competitive performance using only highly valuable samples.

In this work, we aim to define the characteristics of good data for code instruction tuning based on a diverse range of existing code datasets. Our goal is to select the most influential samples through a comprehensive and quantitative data assessment measure. Drawing inspiration from Liu et al. (2024); Ni et al. (2024), we propose a paradigm of data-efficient instruction tuning for code capabilities. Generally, we assume good code samples are complex, of high quality, and diverse. For the complexity aspect, we adopt the evolved complexity scorer to predict the complexity of a given instruction. The scorer is trained on evolved samples via the complexity prompt (Luo et al., 2023a) with ChatGPT. For the aspect of quality, we train a verified model to generate multiple test cases given an (instruction, response) pair and evaluate its quality via the pass rate of the generated test cases. For the aspect of diversity, we select the sample with a large distance to a data pool via instruction embeddings. Combining the three measures, our simple but effective data selection strategy pursues valuable code instruction data and achieves more efficient instruction tuning where fewer training samples yield performance on par with, or even surpassing, models trained on significantly larger datasets. Moreover, we also analyze the composition of our selected data mixture and give suggestions for future code instruction tuning research.

We present XCoder, a family of models fine-tuned from LLaMA3[2] using our selected code instruction data mixture. Experiments on LiveCodeBench and HumanEval demonstrate that XCoder is able to outperform or be on par with state-of-the-art code instruction models such as WizardCoder (Luo et al., 2023a), Magicoder (Wei et al., 2023a), StarCoder2-Instruct[3] and OpenCodeInterpreter (Zheng et al., 2024) while using fewer automatically selected data examples. For example, XCoder-8B based on LLaMA3-8B achieves 43.66 LiveCodeBench-Easy and 54.9 HumanEval when trained on only 40K data samples. Besides, our XCoder-70B based on LLaMA3-70B achieves top-tier results compared to the state-of-the-art open-source models, just behind the recently released Codestral[4].

2 Deep Dive into Existing Datasets

We present mainstream and open-source Code Instruction Tuning datasets in Appendix Table 7. And then we select several influential datasets from these for training and test their performance on the HumanEval and LiveCodeBench benchmarks, with the results shown in Table 1.

From the results, we observe that different training datasets lead to significant performance differences on HumanEval, but the differences on LiveCodeBench are minimal. This phenomenon leads us to suspect whether the remarkably high performance of some data in HumanEval is due to data leakage. Therefore, we propose the **T**est **L**eakage

[2]https://llama.meta.com/llama3/
[3]https://github.com/bigcode-project/starcoder2-self-align
[4]https://mistral.ai/news/codestral/

Dataset	Size	TLI	HumanEval		LiveCodeBench	
			Base-Pass@1	Plus-Pass@1	Pass@1	Easy-Pass@1
Codefuse-Evol-Instruct	66862	8.9	61.0	53.7	13.5	34.5
+Clean	66404 (-0.7%)	4.8 (-4.1)	59.1 (-1.9)	53.7 (0)	12.3 (-1.3)	33.1 (-1.4)
Magicoder-Evol-Instruct	111183	43.2	68.3	64.0	15.3	38.7
+Clean	108063 (-2.8%)	4.9 (-4.0)	65.9 (-2.4)	59.8 (-4.2)	13.0 (-2.3)	34.5 (-4.2)
Code-Feedback	66383	30.5	64.0	57.3	13.8	35.2
+Clean	64134 (-3.4%)	4.6 (-25.9)	56.7 (-7.3)	51.8 (-5.5)	14.8 (+1.0)	38.0 (+2.8)

Table 1: Comparison of performance across three datasets with data leakage and their cleaned versions on HumanEval and LiveCodeBench. TLI measures the extent of data leakage in the training set on HumanEval. Size and performance changes after cleaning are highlighted in red.

Index (TLI) to detect the degree of data leakage for each dataset in the test set.

TLI The Test Leakage Indicator is a metric for quantifying the extent of data leakage from a training set to a test set. To compute TLI, n-grams are generated for both datasets, and the overlap between the n-grams of each test sample and those of all training samples is measured. The similarity score $S(t_i, r_j)$ between a test sample t_i and a training sample r_j is calculated as the fraction of common n-grams over the total n-grams in the test samples. For each test sample, the maximum similarity score among all training samples is recorded. The final TLI metric is the average of these maximum similarity scores across all test set. Higher TLI values indicate greater risks of leakage, highlighting significant similarities between the training and test data.

We calculate the TLI metrics for different datasets on HumanEval, as shown in Table 1. More dataset can be viewed in Appendix B. we find that most datasets maintain a TLI of around 5% on HumanEval, but Codefuse-Evol-Instruct, Magicoder-Evol-Instruct, and Code-Feedback exhibit TLI indices exceeding 30%. Therefore, we further clean these datasets ensuring that the TLI of all cleaned datasets is controlled at 5%, and then conduct re-experiments with these datasets. From the result we can observe that the cleaned datasets, after filtering only a small portion, show a significant performance drop on HumanEval, but their performance on LiveCodeBench remains almost unchanged or even slightly improved. For example, after filtering out 3.4% samples from the Code-Feedback dataset, its performance on the HumanEval Base-Pass@1 metric drops by 7.3%, but its performance on LiveCodeBench slightly increases. This further substantiates the presence of data leakage. Additionally, we discover numerous cases where the training data are almost identical to the test data in HumanEval, confirming the serious data leakage in these datasets,which is a serious issue, causing many datasets that are considered high-quality to perform well solely because the training set is similar to the test set. The leaked cases can be viewed in Appendix B.

3 What Characteristics Do Good Data Have

Inspired by Deita (Liu et al., 2024), we select the samples in the Data Pool from three dimensions: instruction complexity, response quality, and instruction diversity. For a data pool P, we first use the a complexity score C and Unit Test Model U to calculate the complexity score c and quality score q for each data. Then, we use linearly combine c' and q' to obtain a score s representing complexity and quality. Finally, we sort the data pool P and apply the Diversity-based Sampling to iteratively select samples from the data pool into the final training set D, until D reaches the budget size. Our data selection approach is illustrated in the Figure 2 and Algorithm 1. The details of Complexity Score, Unit Test Model and Diversity-based Sampling are as follows.

3.1 Instruction Complexity: Complexity Scorer

Inspired by Evol Complexity(Liu et al., 2024), which is a complexity measure based on evolution. We use evolved instructions to train our complexity scorer. Specifically, we use Self-Instruct to obtain a small-scale dataset $Seed = \{S_1, S_2, \ldots, S_N\}$ as the seed for evolution. Then, we apply the In-Depth Evolving Prompt from WizardCoder for M rounds of evolution. This process results in an instruction set where each seed instruction s_i has M evolved instructions and their corresponding rounds $\{(S_i, 0), (I_1, 1), \ldots, (I_M, M)\}$. We then treat the rounds as a complexity measure and train

Algorithm 1 Data Selection Algorithm For XCoder

Input: Code Instructing Tuning Data Pool $P = \{(I_1, R_1), (I_2, R_2), \ldots, (I_N, R_N)\}$, Number of data samples to be selected Q, Complexity Scorer C, Unit Test Model U, Code Interpreter E, Hyperparameter τ, Weight α
Output: The selected subset D
Initialize Empty Dataset D
for $i = 1$ **to** N **do**
  $c_i \leftarrow \text{C}(\text{I}_i)$
  $u_i \leftarrow \text{U}(\text{I}_i, R_i)$
  $q_i \leftarrow \text{E}(\text{u}_i)$
end for
for $i = 1$ **to** N **do**
  $c_i' \leftarrow Normalized(\text{c}_i)$
  $q_i' \leftarrow Normalized(\text{q}_i)$
  $s_i \leftarrow \alpha \times c_i' + (1 - \alpha) \times q_i'$
end for
$P^* \leftarrow sort(P, key = s, reverse = True)$
for $k = 1$ **to** N **do**
  // $distance(I_k, D)$ denotes the distance between I_k and its nearest neighbor in D
  if $distance(I_k, D) < \tau$ **then**
    $D \leftarrow D \cup \{(I_k, R_k)\}$
  end if
  if $|D| \geq Q$ **then**
    break
  end if
end for

the complexity scorer to predict the complexity score given the input instruction. In multi-turn dialogues, we score each turn separately and use the sum of them as the final score.

3.2 Response Quality: Unit Test Model

In this work, we train a unit test model which can generate a complete executable unit test program based on the given instruction and code snippet for testing. We denote the instruction as I, the code solution as R, and the generated unit test as T. Our unit test model U can be formulated as:

$$T = U(I, R)$$

We collect numerous samples containing instructions, code solutions, and test cases as the training dataset $\{(I_i, R_i, T_i)\}_{i=1}^{N}$. We show some cases output by our unit test model in Appendix C.

3.3 Instruction Diversity: Diversity-based Sampling

We use Diversity-based Sampling method to ensure the diversity of the selected data. The iterative method selects samples P_i one by one from the pool P, and when p_i contributes to the diversity of the selected dataset D, it is added to D. This process continues until the budget Q is reached or all samples p_i in P have been enumerated. Specifically, the benefit of the diversity brought by the newly considered sample p_i can be formulated as an indicator function $F(p_i, D) := distantce(p_i, D) < \tau$, which equals 1 only when $F(p_i, D)$ is true, otherwise it is 0. Only when $F(p_i, D)$ equals 1, p_i will be added to D. We use the embedding distance between the sample p_i and its nearest neighbor in D to calculate $distantce(p_i, D)$. And τ is a hyperparameter.

4 Experiments

4.1 Benchmarks

- **HumanEval**: HumanEval(Chen et al., 2021a) is a widely researched benchmark test for code language models, specifically designed to evaluate the ability of code generation. It includes 164 hand-written programming problems, each problem includes a function signature, docstring, body, and several unit tests, with an average of 7.7 tests per problem.

- **LiveCodeBench**: LiveCodeBench(Jain et al., 2024a) is a comprehensive and pollution-free benchmark for evaluating Large Language Models in code assessment. It updates new problems in real-time from competitions on three competitive platforms (LeetCode, AtCoder, and CodeForces).

4.2 Implementaion Details

Data Pools: To construct the best Code Instruction Tuning dataset, we gathered various available open-source datasets, as detailed in Table 1. This resulted in a collection of 2.5M data samples. However, this amount of data is excessively large. To control the size of the Data Pools, we implemented a straightforward filtering process according to the following rules: Firstly, We include datasets proposed by academic work: Magicoder-OSS-Instruct, Magicoder-Evol-Instruct, and CodeFeedback. And then we select the longest 200K samples to add to the Data Pools. Following this,

Dataset	Size	LiveCodeBench		HumanEval	
		Pass@1	Easy-Pass@1	Base-Pass@1	Plus-Pass@1
Code-Alpaca	20k	0.0	0.0	30.5	25.6
StarCoder2-Self-Align	50k	9.5	24.7	37.8	34.8
Codefuse-Evol-Instruct*	66k	12.3	33.1	59.1	53.7
Magicoder-OSS-Instruct	75k	12.8	33.8	54.3	50.0
Magicoder-Evol-Instruct*	100k	13.0	34.5	**65.9**	**59.8**
Code-Feedback-Clean*	64k	14.8	38.0	56.7	51.8
XCoder	40k	16.5	**43.7**	54.9	50.6
XCoder	80k	**16.8**	**43.7**	57.3	53.0

Table 2: Comparison of the performance using XCoder data and other mainstream data on HumanEval and LiveCodeBench. All models are trained based on Llama3—8B-Base and use greedy decoding. For HumanEval, we report both Base-Pass@1 and Plus-Pass@1 results, where Plus-Pass@1 uses more test cases compared to Base-Pass@1 during evaluation. On LiveCodeBench, we report Pass@1 and Easy-Pass@1 results, with Easy-Pass@1 considering only problems categorized as easy, making it more stable and providing better differentiation than Pass@1. * means that this dataset has data leakage, and we performe a simple decontamination.

we sort the data by complexity score and add the top 200K highest-scoring samples. Finally, we performed deduplication on the Data Pools, resulting in a final dataset of 336K samples.

Complexity Scorer: We use ChatGPT to evolve the dataset over 4 iterations on Code-Alpaca as the training set and train on Llama3-8B-Instruct with a learning rate of 2e-5 for 1 epoch.

Unit Test Model: We use 6k TACO data to train our unit test model based on Llama3-70B-Base. TACO is a dataset for code generation that each sample contains question, code solutions and test cases. We train the final unit test model using a learning rate of 5e-6 over 3 epochs.

Diversity: We use Llama3-8B-Base to get the embedding. We set to 0.945 which means we consider an example p_i could increase the diversity of selected dataset D when the embedding distance between xpi and its nearest neighbor is smaller than 0.945.

4.3 Main Results

To validate the effectiveness of XCoder, we conducted experiments on Llama3-8B-Base, with the results shown in Table 2. From the results we can observe that XCoder achieves the best results on LiveCodeBench among other open-source dataset. It also also achieves the best level performance on HumanEval among the non-leak datasets. Additionally, we observe that XCoder is highly efficient with samples, achieving superior performance on LiveCodeBench with only 40K data compared to baseliens on LiveCodeBench. As the data size increases further, XCoder continues to improve on HumanEval, surpassing Code-Feedback, which contains leaked data. We also notice that Magicoder-Evol-Instruct and Codefuse-Evol-Instruct still achieve leading results on HumanEval. The reason may be that the decontamination algorithm cannot completely filter out all leaked data, so some data leakage still exists within these training sets on HumanEval.

We also train XCoder-70B based on Llama3-70B-Base. Figure 7 shows that XCoder-70B is one of the best open-source Code LLMs.

Method	Data Size	LiveCodeBench	
		Pass@1	Easy-Pass@1
Random	40k	11.5	31.0
Complexity	40k	13.3	34.5
+ Quality	40k	15.0	39.4
+ Diversity	40k	**16.5**	**43.7**
Random	80k	11.8	30.3
Complexity	80k	15.0	37.3
+ Quality	80k	**16.8**	41.6
+ Diversity	80k	**16.8**	**43.7**

Table 3: We conduct ablation experiments based on Llama3-8B-Base with two data sizes to validate the effectiveness of each dimension.

4.4 Analysis

4.4.1 Ablation Study

To validate the effectiveness of each data selecting dimension, we conducted ablation experiments with the results shown in Table **??**. As observed across both data sizes, the model's final performance on LiveCodeBench improves with the addition of each dimension, indicating the effectiveness of each dimension.

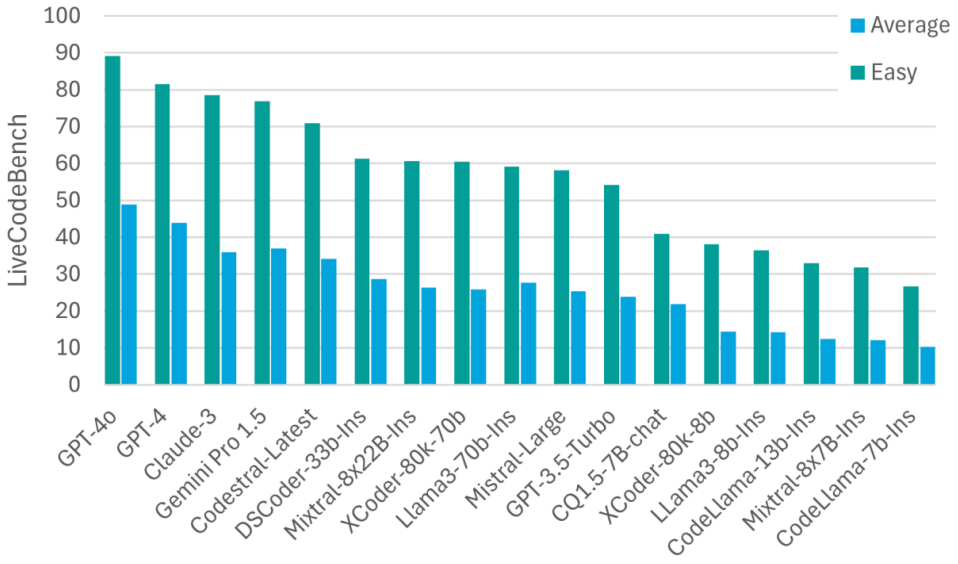

Figure 3: Comparison of the performance of XCoder and other mainstream models on HumanEval and LiveCodeBench. Results for other models are sourced from Eval Plus Leaderboard(Eva) and LiveCodeBench Leaderboard(Liv). For XCoder, we maintain the same settings with other models, where for HumanEval we use a greedy decoding strategy and for LiveCodeBench we use 0.2 temperature, sampling 10 solutions for each question. The full name of CQ-7B-Chat is CodeQwen15-7B-chat.

Measures	Data Size	LiveCodeBench	
		Pass@1	Easy-Pass@1
Random	40k	11.5	31.0
PPL	40k	11.8	31.0
Length	40k	13.0	33.1
Complexity Scorer	40k	**13.6**	**34.5**

Table 4: Comparison of performance on LiveCodeBench using different complexity measurement methods. All models are trained based on Llama3-8B-Base and use Greedy decoding. We calculate PPL for each data point using Llama3-8B-Base. For the length strategy, we only count the instruction length.

Method	BoN-Base-Pass@1	BoN-Plus-Pass@1
Random	62.6	54.9
GPT-4	72.6	62.8
Unit Test Model	76.2	65.2

Table 5: Performance comparison of various methods for evaluating code quality. We report the Best-of-N metric on HumanEval. "Random" indicates selecting a solution randomly from the candidates. "GPT-4" involves direct evaluation of each candidate using GPT-4-0409. The "Unit Test Model" represents using our unit test model to generate and rank based on test cases passed.

4.4.2 Complexity Dimension

Table 4 illustrates the performance of models trained on 40K selected data samples using various complexity measures on LiveCodeBench. The Complexity Scorer measure exhibits the best performance across all measures, surpassing the Random method by 2.1% on Pass@1 and by 3.5% on Easy-Pass@1. The results also indicate that instruction length is a good measure for observing the Code Instruction Tuning data, second only to Complexity Scorer, which contrasts with observations made on general alignment data. Interestingly, perplexity, as an intuitive measure of complexity, performs comparably to the random selection method, consistent with observations by Liu et al. (2024).

4.4.3 Quality Dimension

Using Unit Test for Ranking To validate our Unit Test Model's ability to rank the quality of code, we conducted the following experiment. Specifically, we generate 10 candidate solutions for each question in HumanEval, then use our unit test model to generate test cases for each solution, ranking them based on the number of test cases passed. We select the best one as the final solution. And we use random selection from the candidate solutions as the baseline. The results are shown in Table 5. Additionally, we consider another method where using LLMs to output the correctness of the code directly. We choose GPT-4-0409 to do that. From the results, we observe that compared to random selection, using the unit test model significantly improves the accuracy of the chosen answers, with an increase of nearly 13.6% in the Base-Pass@1 met-

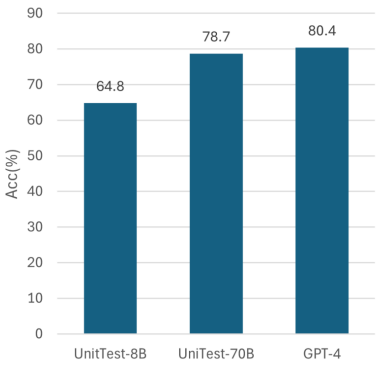

Figure 4: Comparison of the accuracy of Unit Test Models trained on different sizes when generating test cases. We also additionally evaluated the ability of GPT-4 to generate test cases.

ric and 10.3% in the Plus-Pass@1 metric. Notably, the unit test model trained on Llama3-80B-Base also outperforms GPT-4, with improvements of around 3% in both metrics.

From the results, we can observe that using unit tests improves the BoN-Pass@1 metric by approximately 14%, which is higher than merely using language model judgment. However, we also notice a gap in evaluation accuracy per solution compared to GPT-4. We believe this discrepancy may arise because, for unit tests, a solution must pass all the test cases to be considered correct. Any error in generating a test case can cause the solution to fail. Nevertheless, the effectiveness of unit tests in the Best-of-N metric demonstrates that this approach might be more suitable for ranking the quality of code solutions.

Accuracy of Generated Test Cases We also experimented with the impact of different model sizes on the accuracy of the Unit Test Model in generating test cases. Specifically, we instructed the model to generate 10 test cases for the golden solutions in HumanEval, execute them, and count the number of passing test cases. The results are shown in Figure 4. Additionally, we evaluated GPT-4's capability in generating test cases.

We observed that increasing the model parameters significantly improves the accuracy of generating test cases, from 64.8% to 78.7%. Further, we find that the test case model trained on Llama3-70B performs very close to GPT-4 in generating test cases, with a difference of less than 2%.

4.4.4 Data Scaling

To study the impact of our data selection strategy on data scaling efficiency, we conduct experiments using different data budgets. Table 6 shows the experimental results of XCoder outperforms randomly sampled data across different data sizes. Surprisingly, XCoder achieves performance comparable to using 160K training samples with only 10K samples, and it matches the performance of using the full dataset at 80K samples. This demonstrates the high efficiency of XCoder's data samples and the effectiveness of XCoder in data selection.

Method	Data Size	LiveCodeBench	
		Pass@1	Easy-Pass@1
Random	10k	9.8	26.1
Random	40k	11.5	31.0
Random	80k	11.8	30.3
Random	160k	15.0	38.8
Random	320k	16.8	44.4
XCoder	10k	14.5	38.0
XCoder	40k	16.5	43.7
XCoder	80k	16.8	43.7
XCoder	160k	**17.0**	**44.4**

Table 6: Comparison of performance on LiveCodeBench with different datasets as the data scales up. We conducted the training on Llama3-8B-Base.

4.5 Data Analysis

In this section, we analyze the data composition of XCoder, reassess the strengths and weaknesses of different data sources, and develop new insights into different data generation methods. The data composition of XCoder is shown in Figure 5.

Complexity: Figure 5(a) shows the contribution of different data sources in the top 160K complexity samples. We observe that the multi-turn CodeFeedback dataset, which includes code refinement data, contributes the largest amount of samples. And OctoPack, which uses real Git commit information as instructions, results in limited instruction complexity and contributes only 0.1%. However, We also observe that StarCoder2-Self-Align contributes the second largest amount of samples, indicating that, besides Evol-Instruct, converting pretraining data appropriately can also yield complex instructions.

Quality: In Figure 5(b), we observe a data composition that is markedly different from that of complexity. Despite having very low complexity, OctoPack, which uses real code data, exhibits higher quality. Moreover, we notice that although Magicoder-Evol-Instruct, which used GPT-4, contribute a similar number of complexity samples, there is a significant difference in quality. More capable models clearly generate higher quality train-

ing data. And Code Alpaca, generated with text-davinci-003, contributes the fewest high-quality samples. Notably, StarCoder2-Self-Align data contributes a considerable amount, which we think is potentially due to its use of self-synthesized test cases and the rejection of samples that do not execute correctly.

Diversity: Figures 5(c) and 5(d) illustrate the impact on data composition when applying the diversity dimension. We find that after applying diversity measures, OctoPack jumps from the lowest contribution to the second highest, indicating, consistent with researchers' expectations, that instruction data constructed from real-world data exhibits better diversity.

Overall, we find that in terms of complexity: besides Evol-Instruct, generating instructions from real data approximately can also achieve sufficient complexity. Moreover, reasonable transformation of data forms, such as converting single-turn data to multi-turn data and covering a richer variety of task types, can further enhance the complexity of instructions. In terms of quality: datasets constructed with real code data generally exhibit better quality. Using appropriate feedback signals to filter the training set can also lead to a high-quality dataset. And the quality of model-distilled datasets is often related to the capability of the model. In terms of diversity: real-world data tends to be more diverse than Evol data generated using fixed seeds.

5 Related Work

Code Instruction Tuning Recently, LLMs have shown remarkable abilities in understanding and generating code(OpenAI, 2023; Bai et al., 2023; Daya Guo, 2024; Lozhkov et al., 2024a). Code instruction tuning is a neceesarry step for models to accurately understant human instructions and generate relevant code responses. Many works foucs on annotating a high-quality code instruction tuning dataset(Lei et al., 2024). Self-Instruct (Chaudhary, 2023) used text-davinci-003 to generate 20K instruction data. Luo et al. (2023b) apply the Evol-Instruct method proposed by WizadLM(Xu et al., 2023) to 20K Code-Alpaca dataset and get a 79K code instruction dataset. Muennighoff et al. (2023b) take git commits as natural instruction data. They collect 4TB git commits across 350 programming language to fine-tuning OctoCoder. (Wei et al., 2023b) proposed OSS-Instruct, which leverages open-source code cnippets to generate high-quality instructions. They also propose Magicoder-Evol-Instruct dataset and train Magicoder-S, which is the first 7B model to exceed 70% on HumanEval Pass@1. However, we find this dataset suffers from serious data contamination. Based on Magicoder-Evol-Instruct, OpenCodeInterpreter(Zheng et al., 2024) and AutoCoder(Lei et al., 2024) leverages GPT-4 and Code Interpreter as code feedback to generate multi-turn instruction data which instruct model to refine incorrect code snippets acrroding to feedback information.

Data Selection for LLMs Data selection is crucial for LLMs during the instruction fine-tuning phase. While instruction fine-tuning primarily relies on a large volume of data, research such as LIMA (Zhou et al., 2024) indicates that data quality is more critical than quantity. INSTRUCT-MINING (Cao et al., 2023) introduces a linear rule-based method for assessing the quality of instructional data. Furthermore, Instag (Lu et al., 2023) uses an open-domain, tag-based data filtering method, achieving promising performance in MT-Bench. Deita (Liu et al., 2024) integrates a multifaceted approach for selecting instruction data, focusing on complexity, quality, and diversity. WaveCoder (Yu et al., 2023) centers on enhancing LLMs through an instruction improvement technique, incorporating generated data with a particular emphasis on the data filtering phase.

6 Conclusion And Future Work

Code LLMs have raised great interest in current LLM research and plenty of code instruction datasets are proposed over time. However, although many of them claim that good results are achieved on the popular benchmark HumanEval, we find several datasets may have data leakage by using benchmark samples as seed data in self-instruct or evolve-instruct. In this paper, we aim to identify which dataset genuinely qualifies as high-quality code instruction data and propose an efficient code data selection strategy for selecting valuable samples. Based on three dimensions of assessing data, we present XCoder, a family of models finetuned from LLaMA3 on our selected dataset. XCoder achieves superior performance than the SOTA baselines using fewer training samples. From the composition of our selected data mixture, we find existing code datasets have different characteristics corresponding to their construction methods, which provide new insights for developing better code LLMs.

7 Limitation

Our limitations are two-fold: (1) We only explore our method on the Llama3-Base model. More experiments on different model bases are needed to confirm our conclusions. (2) We only focus on the Code Generation task, and in the future, we need to incorporate data containing more tasks.

8 Broader Impacts

Similar to the other LLMs, our XCoder could also generate unethical, harmful, or misleading information, which is not considered in our work. Future research to address the ethical and societal implications is needed. XCoder is also susceptible to hallucination in ungrounded generation use cases due to its smaller size. This model is solely designed for research settings, and its testing has only been carried out in such environments. It should not be used in downstream applications, as additional analysis is needed to assess potential harm or bias in the proposed application

References

Bigcode/self-oss-instruct-sc2-exec-filter-50k. https://huggingface.co/datasets/bigcode/self-oss-instruct-sc2-exec-filter-50k.

a. Codefuse-ai/codeexercise-python-27k. https://huggingface.co/datasets/codefuse-ai/CodeExercise-Python-27k.

b. Codefuse-ai/evol-instruction-66k. https://huggingface.co/datasets/codefuse-ai/Evol-instruction-66k.

a. Cognitivecomputations/code-290k-sharegpt-vicuna. https://huggingface.co/datasets/cognitivecomputations/Code-290k-ShareGPT-Vicuna.

b. Cognitivecomputations/leet10k-alpaca. https://huggingface.co/datasets/cognitivecomputations/leet10k-alpaca.

c. Cognitivecomputations/oa_leet10k. https://huggingface.co/datasets/cognitivecomputations/oa_leet10k.

Evalplus leaderboard. https://evalplus.github.io/leaderboard.html. (Accessed on 06/16/2024).

Glaiveai/glaive-code-assistant-v3. https://huggingface.co/datasets/glaiveai/glaive-code-assistant-v3.

Juyongjiang/codeup: Codeup: A multilingual code generation llama2 model with parameter-efficient instruction-tuning on a single rtx 3090. https://github.com/juyongjiang/CodeUp.

Livecodebench leaderboard. https://livecodebench.github.io/leaderboard.html. (Accessed on 06/16/2024).

Sahil2801/codealpaca-20k. https://huggingface.co/datasets/sahil2801/CodeAlpaca-20k.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Yihan Cao, Yanbin Kang, and Lichao Sun. 2023. Instruction mining: High-quality instruction data selection for large language models. *arXiv preprint arXiv:2307.06290*.

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

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021b. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam M. Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Benton C.

Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier García, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Díaz, Orhan Firat, Michele Catasta, Jason Wei, Kathleen S. Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. Palm: Scaling language modeling with pathways. *J. Mach. Learn. Res.*, 24:240:1–240:113.

Dejian Yang Zhenda Xie Kai Dong Wentao Zhang Guanting Chen Xiao Bi Y. Wu Y.K. Li Fuli Luo Yingfei Xiong Wenfeng Liang Daya Guo, Qihao Zhu. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024a. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida I. Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024b. Livecodebench: Holistic and contamination free evaluation of large language models for code. *ArXiv*, abs/2403.07974.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The stack: 3 tb of permissively licensed source code. *ArXiv*, abs/2211.15533.

Bin Lei, Yuchen Li, and Qiuwu Chen. 2024. Autocoder: Enhancing code large language model with AIEV-INSTRUCT. *Preprint*, arXiv:2405.14906.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nourhan Fahmy, Urvashi Bhattacharyya, W. Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jana Ebert, Tri Dao, Mayank Mishra, Alexander Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean M. Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! *ArXiv*, abs/2305.06161.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom, Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de, Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey, Cherepanov, James Molloy, Daniel Jaymin Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de, Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378:1092 – 1097.

Wei Liu, Weihao Zeng, Keqing He, Yong Jiang, and Junxian He. 2024. What makes good data for alignment? a comprehensive study of automatic data selection in instruction tuning. *Preprint*, arXiv:2312.15685.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024a. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan L. Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, W. Yu, Lucas Krauss, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alexander Gu, Binyuan Hui, Tri

Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sébastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024b. Starcoder 2 and the stack v2: The next generation. *ArXiv*, abs/2402.19173.

Keming Lu, Hongyi Yuan, Zheng Yuan, Runji Lin, Junyang Lin, Chuanqi Tan, Chang Zhou, and Jingren Zhou. 2023. instag: Instruction tagging for analyzing supervised fine-tuning of large language models. *Preprint*, arXiv:2308.07074.

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

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

Niklas Muennighoff, Qian Liu, Qi Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and S. Longpre. 2023a. Octopack: Instruction tuning code large language models. *ArXiv*, abs/2308.07124.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023b. Octopack: Instruction tuning code large language models. *Preprint*, arXiv:2308.07124.

Xinzhe Ni, Yeyun Gong, Zhibin Gou, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2024. Exploring the mystery of influential data for mathematical reasoning. *ArXiv*, abs/2404.01067.

OpenAI. 2023. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, I. Evtimov, Joanna Bitton, Manish P Bhatt, Cristian Cantón Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre D'efossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *ArXiv*, abs/2308.12950.

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

Yejie Wang, Keqing He, Guanting Dong, Pei Wang, Weihao Zeng, Muxi Diao, Yutao Mou, Mengdi Zhang, Jingang Wang, Xunliang Cai, and Weiran Xu. 2024. Dolphcoder: Echo-locating code large language models with diverse and multi-objective instruction tuning. *ArXiv*, abs/2402.09136.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023a. Magicoder: Source code is all you need. *ArXiv*, abs/2312.02120.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023b. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*.

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

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2023. Wavecoder: Widespread and versatile enhanced instruction tuning with refined data generation. *arXiv preprint arXiv:2312.14187*.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. *ArXiv*, abs/2402.14658.

Chunting Zhou, Pengfei Liu, Puxin Xu, Srinivasan Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, et al. 2024. Lima: Less is more for alignment. *Advances in Neural Information Processing Systems*, 36.

A Other Implementation Details

Training Details: We trained on Llama3-8B-Base and Llama3-70B-Base. For the 8B model, we train with a learning rate of 2e-5, while for the 70B model, we use a learning rate of 5e-6. All models are trained for 2 epochs. The batch size during training varies according to the dataset size: for datasets with fewer than 40K samples, the batch size is set to 256; for datasets between 40K and 80K samples, the batch size is set to 512; for datasets between 80K and 160K samples, the batch size is set to 1024; and for datasets larger than 160K samples, the batch size is set to 2048.

B Case Study on Data Leakage

We show examples of data leakage in Codefuse-Evol-Instruct, Magicoder-Evol-Instruct and Code-Feed-back in Figure 6. The statistical information of data leakage can be seen in Table 8.

Dataset	Data Size	Instruction Source	Response Source
Code-290k-ShareGPT-Vicuna (Cog, a)	289k	-	-
CodeExercise-Python-27k (Cod, a)	270k	GPT	GPT
NL2Code (Juy)	19k	GPT(Self-Instruct)	GPT
Glaive-code-assistant-v3-(Gla)	950k	Glaive	Glaive
oa_leet10k (Cog, c)	23k	-	-
CodeAlpaca-20k (Sah)	20k	GPT(Self-Instruct)	GPT
Evol-instruction-66k (Cod, b)	66k	GPT(Evol-Instruct)	GPT
Magicoder-Evol-Instruct-110K (Wei et al., 2023b)	110k	GPT(Evol-Instruct)	GPT
Magicoder-OSS-Instruct-75K (Wei et al., 2023b)	75k	GPT(OSS-Instruct)	GPT
CommitPackFT (Muennighoff et al., 2023b)	702k	GitHub	GitHub
self-oss-instruct-sc2-exec-filter-50k (Big)	50k	StarCoder2(OSS-Instruct)	StarCoder2
Leet10k_alpaca (Cog, b)	10k	-	-

Table 7: Mainstream and open-source Code Instruction Tuning datasets. Self-Instruct(Taori et al., 2023) uses LLMs to generate new instructions based on a seed instruction set. Evol-Instruct(Xu et al., 2023; Luo et al., 2023b) use In-Depth Prompts to generate more compelxity instructions. OSS-Instruct(Wei et al., 2023b) synthesises diversity instructions through real code snippests.

Dataset	Size	TLI	HumanEval		LiveCodeBench	
			Base-Pass@1	Plus-Pass@1	Pass@1	Easy-Pass@1
Code-Alpaca	20022	3.4	30.5	25.6	0.0	0.0
StarCoder2-Self-Align	50661	4.7	37.8	34.8	9.5	24.7
Magicoder-OSS-Instruct	75197	4.5	54.3	50.0	12.8	33.8
Codefuse-Evol-Instruct	66862	8.9	61.0	53.7	13.5	34.5
Magicoder-Evol-Instruct	111183	43.2	68.3	64.0	15.3	38.7
Code-Feedback	66383	30.5	64.0	57.3	13.8	35.2
Codefuse-Evol-Instruct-clean	66404 (-0.7%)	4.8 (-4.1)	59.1 (-1.9)	53.7 (0)	12.3 (-1.3)	33.1 (-1.4)
Magicoder-Evol-Instruct-clean	108063 (-2.8%)	4.9 (-4.0)	65.9 (-2.4)	59.8 (-4.2)	13.0 (-2.3)	34.5 (-4.2)
Code-Feedback-clean	64134 (-3.4%)	4.6 (-25.9)	56.7 (-7.3)	51.8 (-5.5)	14.8 (+1.0)	38.0 (+2.8)

Table 8: Data Leakage Statistics on HumanEval

C Example of input and output for unit test model

We present an input and output case of unit test model in Figure 8.

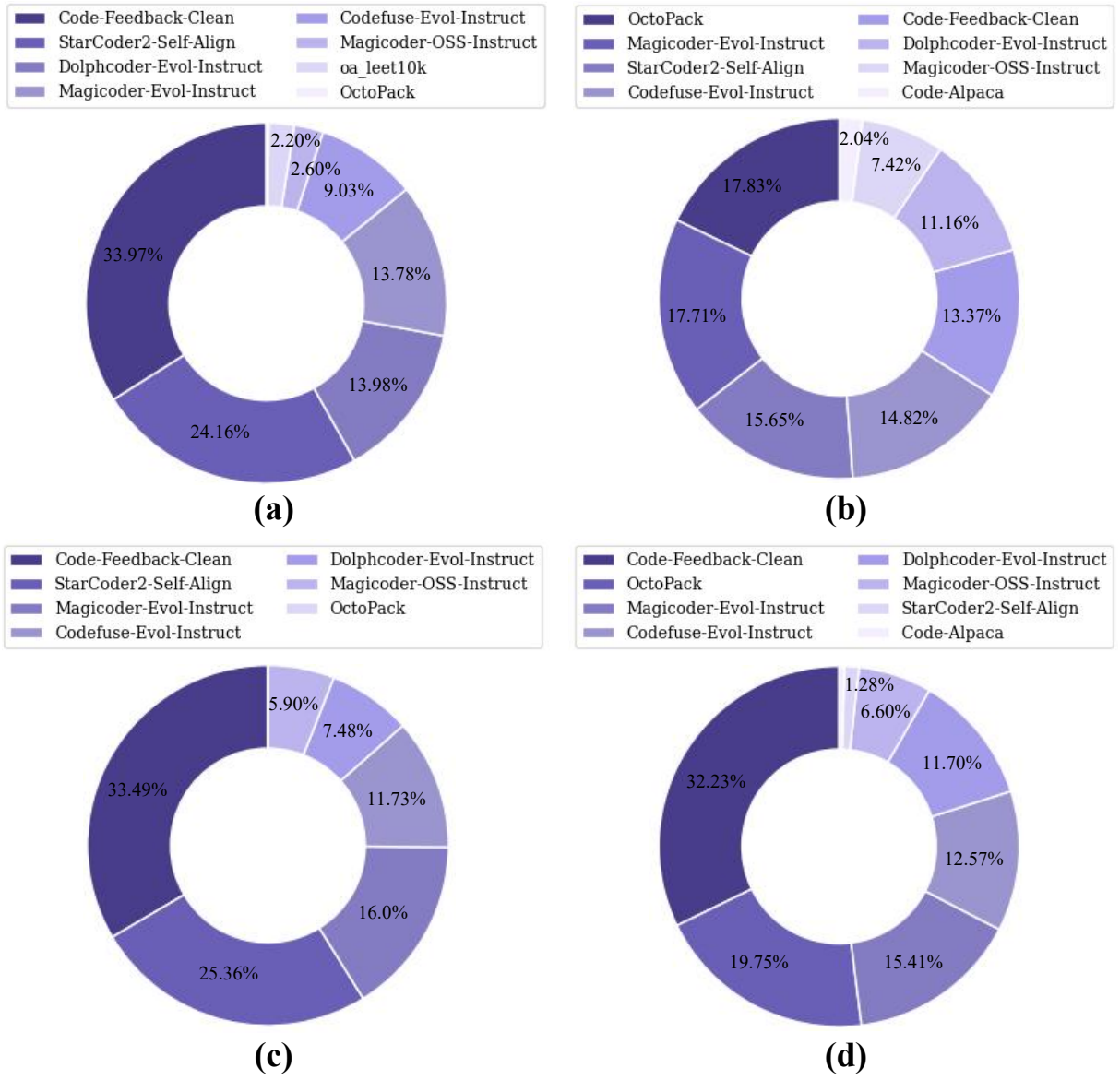

Figure 5: The contribution ratio of different data sources to XCoder, with (a) representing the source of the 160K samples with the highest complexity, (b) representing the 160K samples with the highest quality, and (c) and (d) reflecting which dataset has better diversity.

Figure 6: Examples of data leakage.

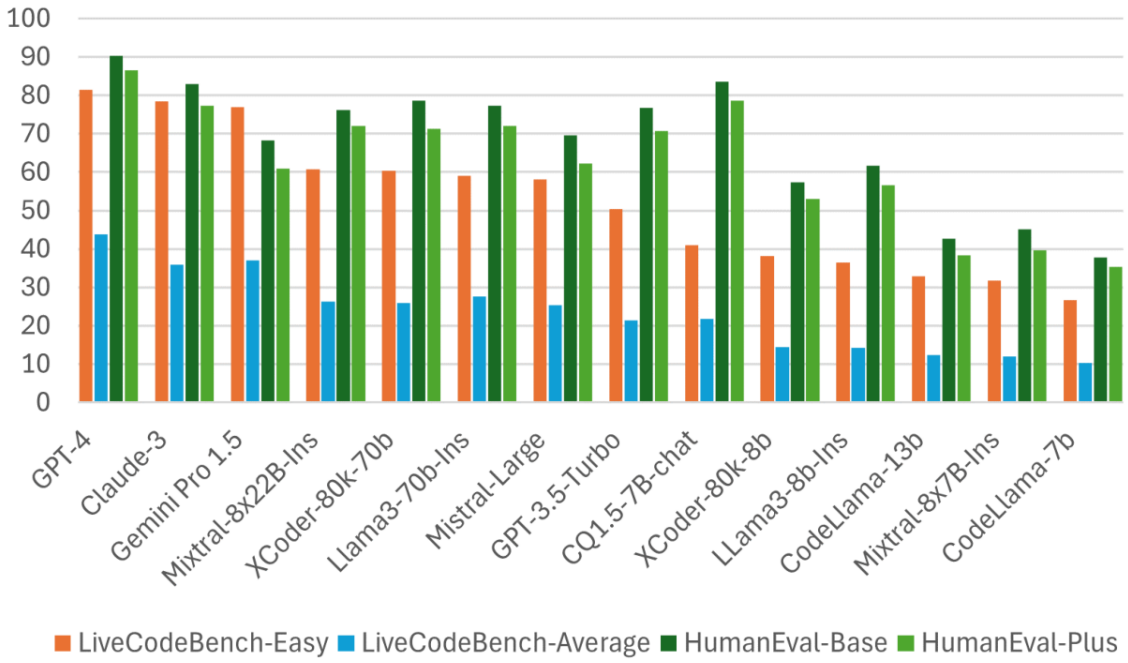

Figure 7: Comparison of the performance of XCoder and other mainstream models on HumanEval and LiveCodeBench. Results for other models are sourced from Eval Plus Leaderboard(Eva) and LiveCodeBench Leaderboard(Liv). For XCoder, we maintain the same settings with other models, where for HumanEval we use a greedy decoding strategy and for LiveCodeBench we use 0.2 temperature, sampling 10 solutions for each question. The full name of GPT-4, Glaude-3, Gemini Pro 1.5, GPT-3.5-Turbo and CQ-7B-Chat are GPT-4o-2024-05-13, GPT-4-Turbo-2024-04-09, Claude-3-opus, Gemini Pro 1.5-May, GPT-3.5-Turbo-0125 and CodeQwen15-7B-chat.

Input:

Please write a hard-coded unit test script according to the provided task description and the solution to be tested, following the format in the demonstration.

Test script framework demonstration:

```
def test_code_name(x, y): # The unit test script needs to contain the code to be tested
    return x + y

def test_script(): # The entry point of the test script needs to be called test_script(), and unittest, pytest and other libraries are not used in the test script.
 # use hard code to descripe test case, each test_case is stored in the list in the form of a tuple,
    input_output = [ ((1,2),(3)),
                     ((2,-4),(-2)),
                    ((-100,99),(-1)) ]
    for id, (input, output) in enumerate(input_output):
        try: # avoid program interruption
            result = test_code_name(*input) == output # The test result is stored as a boolean type
            print(f'Test case {{id+1}}: {{result}}') # print in specified format
        except:
            pass

if __name__ == '__main__': test_script()
```

Task Description:

Can you provide a comprehensive analysis on the estimation of the complexity of the selection sort algorithm, encompassing both the time and space aspects and their implications on system performance? Could you please generate a code snippet for the topic we're discussing?

```
# Python program for implementation of Selection Sort
def selection_sort(arr):
    for i in range(len(arr)):
        # Find the minimum element in remaining unsorted array
        min_index = i
        for j in range(i+1, len(arr)):
            if arr[min_index] > arr[j]:
                min_index = j
        # Swap the found minimum element with the first element of unsorted array
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr
# Testing the algorithm with an array
arr = [64, 25, 12, 22, 11]
print("Sorted array is:", selection_sort(arr))
```

1.Give a complete unit test script, which needs to include the code snippet to be tested, referring to the test_code_name function in the test script framework demonstration.
2. If the code to be tested needs standard input and output, please simulate the standard input and output environment.
3. It needs to be strictly generated according to the test script framework, do not use unittest and other libraries.
4. Each unit test script includes 3 test cases.
5. Test cases need to use a hard-coded approach.
6. Test cases need to follow the same format as the demonstration.

Output:

```
# Python program for implementation of Selection Sort
def selection_sort(arr):
    for i in range(len(arr)):
        # Find the minimum element in remaining unsorted array
        min_index = i
        for j in range(i+1, len(arr)):
            if arr[min_index] > arr[j]:
                min_index = j
        # Swap the found minimum element with the first element of unsorted array
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr
# Testing the algorithm with an array
arr = [64, 25, 12, 22, 11]
print("Sorted array is:", selection_sort(arr))
```

Figure 8: Input and output case of unit test model.