ARCTIC-SNOWCODER: DEMYSTIFYING HIGH-QUALITY DATA IN CODE PRETRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent studies have been increasingly demonstrating that high-quality data is crucial for effective pretraining of language models. However, the precise definition of "high-quality" remains underexplored. Focusing on the code domain, we introduce Arctic-SnowCoder-1.3B, a data-efficient base code model pretrained on 555B tokens through three phases of progressively refined data: (1) *general pretraining* with 500B standard-quality code tokens, preprocessed through basic filtering, deduplication, and decontamination, (2) *continued pretraining* with 50B high-quality tokens, selected from phase one by a BERT-style quality annotator trained to distinguish good code from random data, using positive examples drawn from high-quality code files, along with instruction data from Magicoder and StarCoder2-Instruct, and (3) *enhanced pretraining* with 5B synthetic data created by Llama-3.1-70B using phase two data as seeds, adapting the Magicoder approach for pretraining. Despite being trained on a limited dataset, Arctic-SnowCoder achieves state-of-the-art performance on BigCodeBench, a coding benchmark focusing on practical and challenging programming tasks, compared to similarly sized models trained on no more than 1T tokens, outperforming Phi-1.5-1.3B by 36%. Across all evaluated benchmarks, Arctic-SnowCoder-1.3B beats StarCoderBase-3B pretrained on 1T tokens. Additionally, it matches the performance of leading small base code models trained on trillions of tokens. For example, Arctic-SnowCoder-1.3B surpasses StarCoder2-3B, pretrained on over 3.3T tokens, on HumanEval+, a benchmark that evaluates function-level code generation, and remains competitive on BigCodeBench. Our evaluation presents a comprehensive analysis justifying various design choices for Arctic-SnowCoder. Most importantly, we find that the key to high-quality data is its consistency with the distribution of downstream applications.

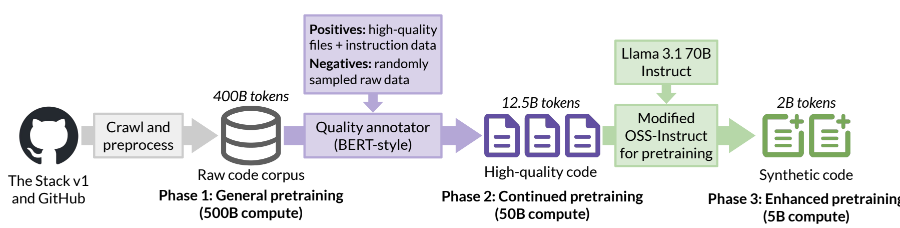

Figure 1: The three-phase pretraining of Arctic-SnowCoder-1.3B using progressively higher-quality data, sourced from the same raw code corpus.

1 INTRODUCTION

Pretraining large language models (LLMs) has generally relied on vast quantities of data. This emphasis on data volume is especially true in specialized domains like code, where researchers obtain massive code pretraining datasets by crawling platforms like GitHub (Li et al., 2023a; Rozière et al., 2024; Guo et al., 2024; Lozhkov et al., 2024; Mishra et al., 2024; DeepSeek-AI et al., 2024b). Recent studies, however, have increasingly showed that high-quality data is crucial for effective pretraining (DeepSeek-AI et al., 2024a; Penedo et al., 2024; Li et al., 2024; Abdin et al., 2024), including the code domain (Gunasekar et al., 2023; Li et al., 2023b; DeepSeek-AI et al., 2024b).

In the general domain, researchers have explored various techniques to curate high-quality pretraining data for language models. FineWeb-Edu (Penedo et al., 2024) uses a linear regressor built on `Snowflake-arctic-embed-m` (Merrick et al., 2024) embeddings to assess the educational value of web pages and select high-quality content, while the DCLM (Li et al., 2024) approach employs a `fastText`-based (Bojanowski et al., 2017) filter trained on positive examples from high-quality online sources (Wei, 2024) and instruction data (Wei et al., 2024b), and random negative web pages to identify high-quality text. These model-based quality filters have been shown to significantly enhance language model performance on downstream tasks, compared to using unfiltered, large-scale datasets. Similarly, researchers have recognized the importance of high-quality code data for pretraining, with Phi-1 (Gunasekar et al., 2023) using a random forest classifier on CodeGen (Nijkamp et al., 2023) embeddings to select educational code samples, and DeepSeek-Coder-V2 (DeepSeek-AI et al., 2024a) employing a multi-stage `fastText`-based (Bojanowski et al., 2017) pipeline to recall web-related code data and high-quality code from GitHub, achieving state-of-the-art coding performance.

In this paper, we introduce Arctic-SnowCoder-1.3B, a high-performing small code model created by a novel three-step training methodology focused on progressive improvements in data quality. As a result of this methodology, Arctic-SnowCoder-1.3B outperforms StarCoderBase-3B (Li et al., 2023a) across all evaluated benchmarks and exceeds Phi-1.5-1.3B (Li et al., 2023b) by 36% on the complex and practical BigCodeBench benchmark (Zhuo et al., 2024), a benchmark that truly matters for real-world programming. As shown in Figure 1, Arctic-SnowCoder is developed through a three-stage, data-efficient pretraining process that progressively refines the quality of the data used. The first stage involves general pretraining for a 500B token horizon using 400B unique raw code data, which have been preprocessed through basic filtering, deduplication, and decontamination. The 400B raw corpus is primarily derived from the coding data used to train Snowflake Arctic (Snowflake AI Research, 2024), combining cleaned The Stack v1 (Li et al., 2023a) and GitHub crawls. This is followed by continued pretraining on 50B tokens, utilizing a smaller, high-quality subset of 12.5B code files, repeated four times. The high-quality tokens are selected from phase one by a BERT-based (Devlin et al., 2019) quality annotator trained to distinguish good code from random data, using positive examples drawn from publicly available high-quality code files (Wei, 2024), along with instruction data from Magicoder (Wei et al., 2024b) and StarCoder2-Instruct (Wei et al., 2024a). Finally, the model undergoes an enhanced pretraining phase for 5B tokens, leveraging roughly 2B synthetic data generated by Llama-3.1-70B (Dubey et al., 2024). This process uses the phase two data as seeds and adapts the OSS-Instruct methodology from Magicoder (Wei et al., 2024b) by transforming lower-quality seed code into high-quality code documents. Notably, all training phases of Arctic-SnowCoder derive data from the same raw pretraining corpus, ensuring that minimal new knowledge is introduced.

Arctic-SnowCoder-1.3B achieves state-of-the-art results on BigCodeBench (Zhuo et al., 2024), a coding benchmark focusing on practical and challenging programming tasks, among models of similar size trained with $\leq$ 1T tokens. Particularly, it outperforming Phi-1.5-1.3B (Li et al., 2023b) by 36%. Despite being trained on 555B tokens, compared to other state-of-the-art small code models trained on trillions of tokens, Arctic-SnowCoder matches or surpasses the performance of these models on several benchmarks. For instance, Arctic-SnowCoder-1.3B beats StarCoderBase-3B (Li et al., 2023a), trained on over 1T tokens, across all evaluated benchmarks. Arctic-SnowCoder-1.3B outperforms StarCoder2-3B (Lozhkov et al., 2024), trained on over 3T tokens, on HumanEval+ (Chen et al., 2021; Liu et al., 2023) (28.0 vs. 27.4), a benchmark evaluating function-level code generation, while remaining competitive on BigCodeBench (19.4 vs. 21.4). We conduct comprehensive ablation studies to validate the design decisions behind training Arctic-SnowCoder:

- First, our findings indicate that, in general pretraining, organizing file-level data into repositories after partitioning by programming language significantly outperforms the approach of grouping data solely by repository names.
- Additionally, we determine the optimal learning rate schedule, which involves a re-warmup phase followed by linear decay, as well as the ideal repetition of high-quality data during continued pretraining, which we find to be four times.
- More importantly, our comparisons of model-based quality annotators, trained on various data combinations, highlight that the consistency of pretraining data distribution and downstream tasks is crucial for achieving superior performance.

In summary, we make the following contributions:

- We introduce Arctic-SnowCoder-1.3B, a high-performing small code model trained on 555B tokens that benefits from progressive improvements in data quality.
- We demonstrate that high-quality data and synthetic data can significantly improve the model performance despite being seeded from the same raw corpus.
- For the first time, we demystify the notion of data quality in code pretraining by systematically comparing model-based quality annotators trained on different data combinations.
- We provide practical insights into optimal design choices for repo-level grouping in general pretraining, and optimal learning rate schedules and repetitions of high-quality data during continued pretraining, providing practical guidelines for future model development.

2 ARCTIC-SNOWCODER

In this section, we provide a detailed explanation of the training methodology used for Arctic-SnowCoder-1.3B, as illustrated in Figure 1. We begin by discussing the composition of the raw training data in §2.1, followed by an overview of the general pretraining phase in §2.2. Next, we describe the continued pretraining process using high-quality data in §2.3, and finally, we elaborate on the enhanced pretraining with synthetic data in §2.4. The model architecture is based on Llama-2 (Touvron et al., 2023), with specific details provided in Table 1.

Table 1: Model architecture details of Arctic-SnowCoder.

Parameter	Arctic-SnowCoder-1.3B
`hidden_dim`	2048
`ffn_hidden_dim`	5632
`num_heads`	16
`num_kv_heads`	16
`num_layers`	24
`vocab_size`	64000
`seq_len`	8192
`positional_encodings`	RoPE (Su et al., 2023)
`tie_embeddings_and_output_weights`	True

2.1 RAW DATA

The raw pretraining data used to train Arctic-SnowCoder-1.3B consists exclusively of code, primarily derived from the coding data used to train Snowflake Arctic (Snowflake AI Research, 2024). This data combines cleaned versions of The Stack v1 (Li et al., 2023a) and GitHub crawls. From this data, we select 18 popular programming languages for training, similar to StarCoder2-3B (Lozhkov et al., 2024). These languages include Python, Java, C++, C, JavaScript, PHP, C#, Go, TypeScript, SQL, Ruby, Rust, Jupyter Notebook, Scala, Kotlin, Shell, Dart, Swift, amounting to a total of 400B unique tokens.

2.2 GENERAL PRETRAINING

In general pretraining, the model is trained for 500B tokens with a sequence length of 8,192 and a batch size of 512 using Adam (Kingma & Ba, 2017). The learning rate follows a cosine decay after a linear warmup of 600 iterations. We set the maximum learning rate to 5.3×10^{-4} and the minimum to 5.3×10^{-5}, following DeepSeek-Coder (Guo et al., 2024). In this phase, we use the entire 400B raw data without applying additional quality filtering. We start by partitioning code files by programming language, grouping them by repository, and then concatenating them in random order, similar to the StarCoder2 (Lozhkov et al., 2024) approach. In §3.3, we show the advantage of first partitioning code files by programming language. We name the model produced by this phase as Arctic-SnowCoder-alpha.

2.3 Continued pretraining with high-quality data

After general pretraining, we continue pretraining Arctic-SnowCoder-alpha with 50B high-quality tokens sourced from the same raw pretraining corpus. The 50B high-quality tokens are formed by repeating 12.5B top-percentile code file tokens for 4 times scored by our code quality annotator. Inspired by FineWeb-Edu (Penedo et al., 2024) and DCLM (Li et al., 2024), we train a linear classification head on top of `Snowflake-arctic-embed-m` (Merrick et al., 2024), a state-of-the-art embedding model based on BERT (Devlin et al., 2019). The training data comprises 300k positive examples, sampled from a blend of 220k high-quality open-source code files (Wei, 2024), 80k high-quality instruction data from Magicoder (Wei et al., 2024b) and StarCoder2-Instruct (Wei et al., 2024a), and 300 randomly selected code documents from the pretraining corpus. Prior research on code quality, such as Phi-1 (Gunasekar et al., 2023), often overemphasizes the "educational value" of code, skewing models towards simpler benchmarks like HumanEval+ (Chen et al., 2021; Liu et al., 2023). In §3.2, we show that our annotation leads to a more balanced enhancement of model capabilities. Furthermore, given that these code documents typically exceed 1000 tokens, surpassing the BERT context window size of 512, we improve over FineWeb-Edu's pipeline to calculate the score for each file by averaging the scores from the top, middle, and bottom sections as produced by the quality annotator. In this phase, we rewarmup the learning rate for 1000 iterations from 0 to 5.3×10^{-4}, the maximum pretraining learning rate, followed by a linear decay to 0. The model produced in this phase is referred to as Arctic-SnowCoder-beta. In §3.4, we perform a comprehensive analysis that validates all of our design choices.

2.4 Enhanced pretraining with synthetic data

In the enhanced pretraining stage, we generate even higher-quality data than in continued pretraining leveraging Llama-3.1-70B-Instruct (Dubey et al., 2024) and increase the Python mix ratio to approximately 50% while keeping the proportions of the other languages unchanged. Phi-1 (Gunasekar et al., 2023) demonstrates that synthetic, textbook-like pretraining data can significantly enhance model performance. However, overemphasis on such data risks skewing the model's distribution, potentially impairing its effectiveness in real-world coding tasks. For example, we show in §3.2 that Phi-1.5 excels in HumanEval+ (Chen et al., 2021; Liu et al., 2023) and MBPP+ (Austin et al., 2021; Liu et al., 2023), which resemble textbook exercises, but performs less effectively on the more complex and practical coding tasks in BigCodeBench (Zhuo et al., 2024). To address this, we adapt the OSS-Instruct method from Magicoder (Wei et al., 2024b) for pretraining purposes. Originally, OSS-Instruct was originally designed to generate realistic instruction-tuning data by prompting a model to create question-answer pairs inspired by open-source code snippets. In contrast, we produce high-quality synthetic pretraining data by using Llama-3.1-70B-Instruct to generate high-quality and problem-solving oriented code files, seeded with code documents scored in the top percentile during the continued pretraining phase. In §3.2, we conduct an extensive evaluation to demonstrate that each pretraining phase significantly outperforms the previous one, highlighting the effectiveness of progressively enhancing data quality.

3 Experiments

In this section, we compare Arctic-SnowCoder with state-of-the-art small language models and show performance boost over each pretraining stage (§3.2), evaluate two strategies of forming repo-level data in general pretraining (§3.3), and perform detailed ablation to justify our design choices in continued pretraining (§3.4).

3.1 Experimental setup

We consider the following four diverse programming benchmarks to comprehensively evaluate the code generation capability of different code models:

HumanEval+ and MBPP+ (Liu et al., 2023). HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) are the two most widely-used benchmarks for function-level code generation. We adopt their augmented version powered by EvalPlus (Liu et al., 2023), with 80×/35×

more test cases for rigorous evaluation. HumanEval+ and MBPP+ include 164 and 378 coding problems, respectively.

EvoEval (Xia et al., 2024) is a program synthesis benchmark suite created by evolving existing benchmarks into different targeted domains. We employ its five default transformation categories, namely `difficult`, `creative`, `subtle`, `combine` and `tool_use`, totaling 500 tasks.

BigCodeBench (Zhuo et al., 2024) evaluates LLMs with practical and challenging programming tasks. It has 1140 programming tasks, where each task in BigCodeBench is created through human-LLM collaboration, where the task quality is ensured by human experts.

We incorporate HumanEval+, MBPP+, EvoEval, and BigCodeBench for baseline comparison in §3.2. For the subsequent ablation studies in §3.3 and §3.4, we include the base versions of HumanEval and MBPP while omitting BigCodeBench for faster evaluation. Throughout the experiments, we report the pass@1 metric (Chen et al., 2021) using greedy decoding.

3.2 Baseline comparison and effectiveness of three-stage pretraining

Table 2: Comparing Arctic-SnowCoder with state-of-the-art small language models (< 3B), divided by whether training compute > 1T tokens. Arctic-SnowCoder-alpha and Arctic-SnowCoder-beta are checkpoints after general pretraining and continued pretraining with high-quality data, respectively. Arctic-SnowCoder is the final checkpoint after enhanced pretraining with synthetic data.

Model	Training compute	HumanEval+	MBPP+	EvoEval	BigCodeBench
StableCode-3B (Pinnaparaju et al., 2024)	1.3T	26.2	43.9	18.6	25.9
StarCoder2-3B (Lozhkov et al., 2024)	3.3T to 4.3T	27.4	**49.2**	19.0	21.4
Granite-Code-Base-3B (Mishra et al., 2024)	4.5T	29.3	45.8	**19.8**	20.0
CodeGemma-2B-v1.0 (Team et al., 2024)	3T + 1T	18.3	46.3	15.4	23.9
CodeGemma-2B-v1.1 (Team et al., 2024)	3T + 500B	**32.3**	48.9	**19.8**	**28.0**
Qwen1.5-1.8B (Yang et al., 2024a)	3T	19.5	28.3	5.0	6.3
Qwen2-1.5B (Yang et al., 2024a)	7T	31.1	38.4	17.2	16.5
DeepSeek-Coder-1.3B (Guo et al., 2024)	2T	28.7	48.1	19.2	22.2
StarCoderBase-3B (Li et al., 2023a)	1T	17.7	36.8	11.6	5.9
SmolLM-1.7B (Allal et al., 2024)	1T	15.9	34.7	10.0	2.5
Phi-1.5-1.3B (Li et al., 2023b)	150B	**31.7**	**43.7**	**20.6**	**14.3**
Arctic-SnowCoder-alpha-1.3B	500B	14.0	27.8	7.4	10.3
Arctic-SnowCoder-beta-1.3B	500B + 50B	21.3	34.7	12.8	12.3
Arctic-SnowCoder-1.3B	550B + 5B	**28.0**	**42.9**	**18.0**	**19.4**

Table 2 presents a comprehensive comparison of various small language models (less than 3B parameters) across multiple coding benchmarks, categorized by whether their training compute exceeds 1T tokens. Notably, Arctic-SnowCoder demonstrates exceptional performance, particularly given its limited training data. Arctic-SnowCoder-1.3B achieves state-of-the-art performance on BigCodeBench compared to similarly sized models trained on no more than 1T token, significantly outperforming StarCoderBase-3B, SmolLM-1.7B, and Phi-1.5-1.3B. Particularly, although Phi-1.5-1.3B has an advantage in "textbook-like" benchmarks such as HumanEval+, MBPP+, and EvoEval, Arctic-SnowCoder-1.3B outperforms Phi-1.5-1.3B by 36% on the more complex and practical BigCodeBench. Also, Arctic-SnowCoder-1.3B beats StarCoderBase-3B, the predecessor of StarCoder2-3B trained on 1T tokens, across all evaluated benchmarks. Despite being trained on only 555B tokens, on HumanEval+, Arctic-SnowCoder-1.3B rivals and even surpasses models that have undergone significantly more extensive training, such as StarCoder2-3B, StableCode-3B, CodeGemma-2B-v1.0, and Qwen1.5-1.8B. On EvoEval and BigCodeBench, Arctic-SnowCoder remains competitive. Additionally, the table highlights the consistent improvement of Arctic-SnowCoder across its training phases: Arctic-SnowCoder-alpha, Arctic-SnowCoder-beta, and the final Arctic-SnowCoder. Each phase builds on the previous one, with Arctic-SnowCoder achieving the highest scores in all benchmarks. This steady enhancement emphasizes the crucial role of high-quality and synthetic data in the final phase. Despite starting with the same data, each iteration of Arctic-SnowCoder narrows the gap with state-of-the-art models, demonstrating the efficacy of the overall training approach.

3.3 Repo-level data in general pretraining

In the general pretraining phase, we adopt StarCoder2's approach to group file-level data randomly into repositories through a random concatenation of file contents (Lozhkov et al., 2024). In Table 3, we study two methods: (1) grouping files just by repository names, meaning that each training document can be a mix of multi-lingual code files if the repository is written in different languages, and (2) partitioning files into different programming languages before grouping them into repositories, meaning that each training document only focuses on one single language.

Table 3: Comparison of two methods for grouping repo-level data for pretraining. (1) "Group by repo" treats each repository as a single training unit with possibly mixed languages, and (2) "Group by language and repo" partitions data by programming language before grouping by repository.

Setting	HumanEval (+)	MBPP (+)	EvoEval
Group by repo	12.8 (10.4)	30.7 (25.9)	7.0
Group by language and repo	**17.1 (15.9)**	**33.9 (27.8)**	**7.4**

We can observe that the second approach, which we finally adopt in general pretraining, performs significantly better than the first one. The primary reason for enhanced performance when grouping by language before the repository is that grouping by repositories can result in training instances containing mixed file types, such as configuration files and programming files. During training, we align the compute, meaning that the "grouping by repositories" approach processes fewer tokens specifically from programming files. Additionally, since files are randomly ordered, code files from different languages are often unrelated. Consequently, each training example may include two entirely unrelated files, which can negatively affect learning.

A promising hybrid approach could involve grouping files by language within each repository. This method ensures that training examples can include multiple programming language files while maintaining the cohesion of files in the same language within each group.

3.4 Design choices in continued pretraining

In continued pretraining, we source high-quality tokens from our pretraining corpus and train an improved base model. To obtain high-quality tokens, a model-based quality annotator is employed. In this section, we experiment with various design choices of our approach, including the training data used for the annotator, the learning rate used in continued pretraining, and the optimal repetitions of high-quality tokens.

3.4.1 Model-based quality annotator

Similar to FineWeb-Edu (Penedo et al., 2024), we train a linear head on top of the `Snowflake-arctic-embed-m` (Merrick et al., 2024) embedding model to score each code file. In Table 4, we experiment with 4 variants:

- Ann-Edu: We prompt Mixtral-8x7B-Instruct (Jiang et al., 2024) to annotate the educational value of each code file (1 to 5). 400k annotations are used to train a linear regression head. For the following variants, similar to DCLM (Li et al., 2024), we sample negative documents randomly and change the positive parts only. We equip the embedding model with a linear classification head.
- Ann-Ins: Positives are a mix of 100k educational data (3.5+) bootstrapped from Ann-Edu and 100k high-quality instruction data from Magicoder (Wei et al., 2024b) and StarCoder2-Instruct (Wei et al., 2024a).
- Ann-HQ: Positives are 220k open-source, synthetic, high-quality code files (Wei, 2024).
- Ann-HQIns: Positives are a mix of 220k Ann-HQ training data and 80k instruction data from Magicoder (Wei et al., 2024b) and StarCoder2-Instruct (Wei et al., 2024a).

Table 4: Comparison of downstream performance by applying model-based quality annotators trained with different recipes to 10B continued pretraining.

Annotator	Training data	HumanEval (+)	MBPP (+)	EvoEval
Pretrained model (no continued pretraining)		17.1 (15.9)	33.9 (27.8)	7.4
Continued pretraining on random 10B tokens		15.9 (12.8)	30.7 (23.3)	8.0
Ann-Edu	400k Mixtral annotations for educational scores (0–5)	19.5 (16.5)	27.8 (22.2)	10.4
Ann-Ins	100k high Ann-Edu + 100k instruction data from Magicoder (Wei et al., 2024b) and StarCoder2-Instruct (Wei et al., 2024a)	21.3 (18.3)	37.3 (29.9)	10.4
Ann-HQ	220k open-source, synthetic high-quality code files (Wei, 2024)	19.5 (16.5)	33.9 (26.7)	9.2
Ann-HQIns	220k Ann-HQ data mixed with 80k instruction data	**22.0 (18.3)**	**40.2 (33.1)**	**11.6**

After training the annotators, we first apply each annotator to the entire pretraining corpus to obtain a score for each file. Unlike FineWeb-Edu, which only scans the top 2k characters, we scan the top, middle, and bottom parts of a code file and average the scores. We then rank the code files per language based on these scores and select the top percentile of documents until we reach approximately 10 billion tokens. We maintain the same mix ratio as used in pretraining. The table shows that Ann-HQIns, which combines both high-quality files and instruction data, achieves the best downstream performance.

To understand the underlying factor that causes the performance difference, we conduct an additional analysis in Figure 2. For each annotator, we create a validation dataset with positives from code solution benchmarks and negatives from random pretraining data not seen during training. We use the ROC-AUC (Bradley, 1997) (Area Under the Receiver Operating Characteristic Curve) score to evaluate how well the annotator ranks benchmark data. The figure illustrates the correlation between per-benchmark ROC-AUC scores and benchmark pass rates. There is an almost consistent trend: higher ROC-AUC scores lead to better benchmark performance. A good ROC-AUC score indicates that the annotator effectively shapes the distribution of downstream tasks. Thus, the key to high-quality data is essentially the alignment with downstream application distributions.

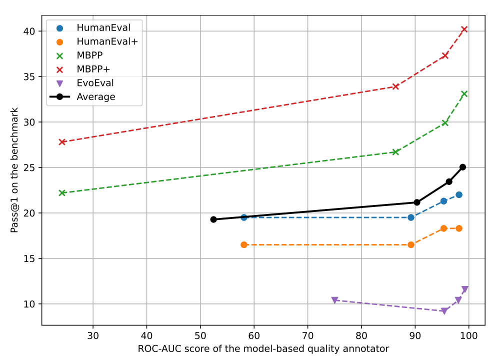

Figure 2: Correlation between annotator ROC-AUC score and benchmark pass@1.

3.4.2 Learning rate schedule

We also study the effect of different learning rate schedules for continued pretraining in Table 5, including (1) a linear annealing starting from the minimum pretraining learning rate to zero, (2) a constant schedule using the minimum pretraining learning rate, and (3) a re-warmup to the maximum pretraining learning rate followed by a linear decay to zero. Empirically, we find that the re-warmup approach performs the best and use it consistently in all the other experiments with respect to continued pretraining.

Table 5: Comparison of different learning rate schedules in 10B continued pretraining using ANN-HQINS. Here $\texttt{MIN\_LR} = 5.3 \times 10^{-5}$ and $\texttt{MAX\_LR} = 5.3 \times 10^{-4}$.

Setting	Schedule	HumanEval (+)	MBPP (+)	EvoEval
Pretraining	$0 \to \texttt{MAX\_LR} \to \texttt{MIN\_LR}$	17.1 (15.9)	33.9 (27.8)	7.4
Linear	$\texttt{MIN\_LR} \to 0$	18.3 (16.5)	37.0 (30.4)	9.8
Constant	$\texttt{MIN\_LR} \to \texttt{MIN\_LR}$	20.7 (18.3)	39.4 (31.7)	9.4
Re-warmup	$0 \to \texttt{MAX\_LR} \to 0$	**22.0 (18.3)**	**40.2 (33.1)**	**11.6**

3.4.3 REPETITIONS OF HIGH-QUALITY DATA

Finally, we scale up the token horizon from 10 billion to 50 billion in continued pretraining. One remaining question to address is determining the optimal repetitions for high-quality tokens. We experiment with repetitions ranging from 1 to 5, as shown in Table 6, by selecting the top percentile tokens ranked by ANN-HQINS. In this context, the top percentile tokens are the highest quality tokens available. For example, $1 \times$ 50B indicates one repetition of the top 50B tokens, while $4 \times$ 12.5B denotes four repetitions of the top 12.5B tokens, ensuring that the selected tokens are of the best quality. Based on the results in the table, repeating the high-quality tokens four times ($4 \times$ 12.5B) yields the best overall downstream performance across multiple evaluation metrics, showing the highest scores for HumanEval and EvoEval. Two repetitions ($2 \times$ 25.0B) and three repetitions ($3 \times$ 16.7B) also demonstrate strong performance, particularly in `mbpp`. Five repetitions ($5 \times$ 10.0B) achieve the highest MBPP score but do not surpass the four repetitions in overall metrics. A single repetition ($1 \times$ 50.0B) shows the least improvement compared to multiple repetitions.

Table 6: Downstream performance with varying repetitions of high-quality data in 50B continued pretraining using ANN-HQINS.

Repetition pattern	HumanEval (+)	MBPP (+)	EvoEval
Pretrained	17.1 (15.9)	33.9 (27.8)	7.4
$1 \times$ 10.0B	22.0 (18.3)	40.2 (33.1)	11.6
$1 \times$ 50.0B	17.4 (14.0)	41.5 (33.6)	9.6
$2 \times$ 25.0B	23.2 (19.5)	42.1 (34.7)	9.2
$3 \times$ 16.7B	23.8 (18.9)	42.3 (34.4)	11.2
$4 \times$ 12.5B	**26.2 (21.3)**	40.2 (32.5)	**12.8**
$5 \times$ 10.0B	20.1 (17.7)	**43.9 (36.0)**	10.4

4 RELATED WORK

4.1 CODE PRETRAINING CORPUS FOR LANGUAGE MODELS

Code data is essential to improving the reasoning capabilities of large language models (LLMs) (Aryabumi et al., 2024; Madaan et al., 2022; MA et al., 2024; Yang et al., 2024b; DeepSeek-AI et al., 2024b). Typically, researchers obtain massive code pretraining data by crawling from public platforms hosting code repositories such as GitHub (Li et al., 2023a; Rozière et al., 2024; Guo et al., 2024; Lozhkov et al., 2024; Mishra et al., 2024; DeepSeek-AI et al., 2024b). For example The Stack v1 (Kocetkov et al., 2022) is a 3.1 TB dataset consisting of permissively licensed source code mined from GitHub in 30 programming languages. Its successor The Stack v2 (Lozhkov et al., 2024), built on the Software Heritage archive (Cosmo & Zacchiroli, 2017), is an order of magnitude larger, with a raw dataset of 67.5 TB spanning 619 programming languages. However, directly using these massive unfiltered code for pretraining is suboptimal, because the code documents may contain undesired contents or duplicates. Therefore, further preprocessing steps are needed to downscale the raw corpus, which can include deduplication (Li et al., 2023a; Rozière et al., 2024; Guo et al., 2024; Lozhkov et al., 2024; Mishra et al., 2024; DeepSeek-AI et al., 2024b; Team et al., 2024), PII (Personally Identifiable Information) redaction (Li et al., 2023a; Lozhkov et al., 2024; Mishra et al., 2024), benchmark decontamination (Li et al., 2023a; Lozhkov et al., 2024; Guo et al., 2024; DeepSeek-AI et al., 2024b), and model-based filtering (DeepSeek-AI et al., 2024b). As an example, StarCoder2 (Lozhkov et al., 2024) selects only 3 TB of data for pretraining from the 67.5 TB total data available in The Stack v2. The code pretraining corpus of Arctic-SnowCoder follows a similar preprocessing pipeline, comprising approximately 400B unique tokens from a mix of filtered The Stack v1 and GitHub crawls.

4.2 Model-based quality filtering

In addition to common preprocessing steps like deduplication and heuristic filtering, a recent trend is using model-based quality filters to select high-quality pretraining data. Phi-1 (Gunasekar et al., 2023) employs a random forest classifier trained on top of the CodeGen (Nijkamp et al., 2023) embedding layer on GPT-4 annotations, to assess the educational value of files. This filter selects high-quality The Stack v1 and StackOverflow content, significantly enhancing coding performance. FineWeb-Edu (Penedo et al., 2024) employs a linear regressor built on `Snowflake-arctic-embed-m` (Merrick et al., 2024), an advanced embedding model based on BERT (Devlin et al., 2019). This regressor, trained on 400k Llama-3 (Dubey et al., 2024) annotations rating the educational value (0-5) of FineWeb dataset documents, significantly enhances STEM performance. DCLM-Baseline (Li et al., 2024) uses a `fastText` (Bojanowski et al., 2017) filter trained on positives from OpenHermes 2.5 (Teknium, 2023), high-scoring posts from `r/ExplainLikeImFive`, and random negatives. It outperforms FineWeb-Edu in top-10% selection. DeepSeek-Coder-V2 (DeepSeek-AI et al., 2024b) follows DeepSeek-Math (Shao et al., 2024) by leveraging a multi-stage `fastText`-based pipeline to recall high-quality code and math contents. Llama-3 (Dubey et al., 2024) uses `fastText` for recognizing text referenced by Wikipedia (Wikipedia contributors, 2004) and Roberta-based (Liu et al., 2019) classifiers trained on Llama-2 (Touvron et al., 2023) predictions. While prior work focuses on initial pretraining, Arctic-SnowCoder demonstrates that high-quality data from the pretraining corpus can significantly enhance model performance during continued pretraining. We are also the first to uncover the secret of data quality, revealing the importance of matching data distribution with downstream tasks.

4.3 High-quality code data for pretraining

Phi-1 (Gunasekar et al., 2023) is one of the first to study the impact of high-quality code data. It first uses a random forest classifier to filter out high-quality code data from The Stack v1 and StackOverflow, and then creates synthetic textbook-like data and exercises using GPT-3.5 (OpenAI, 2022), showing significant coding performance with only 50B+ training tokens. DeepSeek-Coder-V2 (DeepSeek-AI et al., 2024b), pretrained for around 14T tokens in total, achieves state-of-the-art coding performance, with a multi-stage `fastText`-based (Bojanowski et al., 2017) pipeline to recall web-related code data as well as high-quality GitHub code. Arctic-SnowCoder utilizes a high-quality code annotator to extract high-quality code from pretraining datasets. It then generates synthetic files seeded from this high-quality data, adapting Magicoder OSS-Instruct (Wei et al., 2024b) into pretraining.

5 Conclusion

We introduce Arctic-SnowCoder-1.3B, a high-performing code model that underscores the critical importance of data quality in the pretraining process. Trained on 555B tokens, Arctic-SnowCoder-1.3B achieves competitive results with state-of-the-art small code models while using significantly fewer tokens. Our three-stage pretraining process begins with 500B tokens of general pretraining on a raw code corpus, followed by 50B high-quality tokens scored by a quality annotator, and concludes with 5B tokens of synthetic data for further enhancement. This work demystifies the notion of high-quality data in code pretraining by demonstrating the key to high-quality data is its alignment with the distribution of downstream applications. Additionally, the paper offers practical guidelines for repo-level data grouping, learning rate scheduling, and the repetition of high-quality data, paving the way for more efficient and effective code model development.

References

Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Qin Cai, Martin Cai, Caio César Teodoro Mendes, Weizhu Chen, Vishrav Chaudhary, Dong Chen, Dongdong Chen, Yen-Chun Chen, Yi-Ling Chen, Parul Chopra, Xiyang Dai, Allie Del Giorno, Gustavo de Rosa, Matthew Dixon, Ronen Eldan, Victor Fragoso, Dan Iter, Mei Gao, Min Gao, Jianfeng Gao, Amit Garg, Abhishek Goswami, Suriya Gunasekar, Emman Haider, Junheng Hao, Russell J. Hewett, Jamie Huynh, Mojan Javaheripi, Xin Jin, Piero Kauffmann, Nikos Karampatziakis, Dongwoo Kim, Mahoud Khademi, Lev Kurilenko, James R. Lee, Yin Tat Lee, Yuanzhi Li, Yunsheng Li, Chen Liang, Lars Liden, Ce Liu, Mengchen Liu, Weishung Liu, Eric Lin, Zeqi Lin, Chong Luo, Piyush Madan, Matt Mazzola, Arindam Mitra, Hardik Modi, Anh Nguyen, Brandon Norick, Barun Patra, Daniel Perez-Becker, Thomas Portet, Reid Pryzant, Heyang Qin, Marko Radmilac, Corby Rosset, Sambudha Roy, Olatunji Ruwase, Olli Saarikivi, Amin Saied, Adil Salim, Michael Santacroce, Shital Shah, Ning Shang, Hiteshi Sharma, Swadheen Shukla, Xia Song, Masahiro Tanaka, Andrea Tupini, Xin Wang, Lijuan Wang, Chunyu Wang, Yu Wang, Rachel Ward, Guanhua Wang, Philipp Witte, Haiping Wu, Michael Wyatt, Bin Xiao, Can Xu, Jiahang Xu, Weijian Xu, Sonali Yadav, Fan Yang, Jianwei Yang, Ziyi Yang, Yifan Yang, Donghan Yu, Lu Yuan, Chengruidong Zhang, Cyril Zhang, Jianwen Zhang, Li Lyna Zhang, Yi Zhang, Yue Zhang, Yunan Zhang, and Xiren Zhou. Phi-3 technical report: A highly capable language model locally on your phone, 2024. URL https://arxiv.org/abs/2404.14219.

Loubna Ben Allal, Anton Lozhkov, and Elie Bakouch. Smollm - blazingly fast and remarkably powerful. https://huggingface.co/blog/smollm, 2024.

Viraat Aryabumi, Yixuan Su, Raymond Ma, Adrien Morisot, Ivan Zhang, Acyr Locatelli, Marzieh Fadaee, Ahmet Üstün, and Sara Hooker. To code, or not to code? exploring impact of code in pre-training, 2024. URL https://arxiv.org/abs/2408.10914.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information, 2017. URL https://arxiv.org/abs/1607.04606.

Andrew P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30(7):1145–1159, 1997. ISSN 0031-3203. doi: https://doi.org/10.1016/S0031-3203(96)00142-2. URL https://www.sciencedirect.com/science/article/pii/S0031320396001422.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri 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, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Roberto Di Cosmo and Stefano Zacchiroli. Software heritage: Why and how to preserve software source code. In *iPRES 2017: 14th International Conference on Digital Preservation*, Kyoto, Japan, 2017. URL https://www.softwareheritage.org/wp-content/uploads/2020/01/ipres-2017-swh.pdf. https://hal.archives-ouvertes.fr/hal-01590958.

DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao

Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Liu, Xin Xie, Xingkai Yu, Xinnan Song, Xinyi Zhou, Xinyu Yang, Xuan Lu, Xuecheng Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Zheng, Yichao Zhang, Yiliang Xiong, Yilong Zhao, Ying He, Ying Tang, Yishi Piao, Yixin Dong, Yixuan Tan, Yiyuan Liu, Yongji Wang, Yongqiang Guo, Yuchen Zhu, Yuduan Wang, Yuheng Zou, Yukun Zha, Yunxian Ma, Yuting Yan, Yuxiang You, Yuxuan Liu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhewen Hao, Zhihong Shao, Zhiniu Wen, Zhipeng Xu, Zhongyu Zhang, Zhuoshu Li, Zihan Wang, Zihui Gu, Zilin Li, and Ziwei Xie. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024a. URL https://arxiv.org/abs/2405.04434.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence, 2024b. URL https://arxiv.org/abs/2406.11931.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio (eds.), *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Olivier Duchenne, Onur

Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaoqing Ellen Tan, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aaron Grattafiori, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alex Vaughan, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Franco, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, Danny Wyatt, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Firat Ozgenel, Francesco Caggioni, Francisco Guzmán, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Govind Thattai, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Igor Molybog, Igor Tufanov, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Karthik Prasad, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kun Huang, Kunal Chawla, Kushal Lakhotia, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Maria Tsimpoukelli, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikolay Pavlovich Laptev, Ning Dong, Ning Zhang, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Rohan Maheswari, Russ Howes, Ruty Rinott, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Kohler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaofang Wang, Xiaojian Wu, Xiaolan Wang, Xide Xia, Xilun Wu, Xinbo Gao, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yuchen Hao, Yundi Qian, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, and Zhiwei Zhao. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks are all you need, 2023. URL https://arxiv.org/abs/2306.11644.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024. URL https://arxiv.org/abs/2401.14196.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024. URL https://arxiv.org/abs/2401.04088.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. URL https://arxiv.org/abs/1412.6980.

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. The stack: 3 tb of permissively licensed source code, 2022. URL https://arxiv.org/abs/2211.15533.

Jeffrey Li, Alex Fang, Georgios Smyrnis, Maor Ivgi, Matt Jordan, Samir Gadre, Hritik Bansal, Etash Guha, Sedrick Keh, Kushal Arora, Saurabh Garg, Rui Xin, Niklas Muennighoff, Reinhard Heckel, Jean Mercat, Mayee Chen, Suchin Gururangan, Mitchell Wortsman, Alon Albalak, Yonatan Bitton, Marianna Nezhurina, Amro Abbas, Cheng-Yu Hsieh, Dhruba Ghosh, Josh Gardner, Maciej Kilian, Hanlin Zhang, Rulin Shao, Sarah Pratt, Sunny Sanyal, Gabriel Ilharco, Giannis Daras, Kalyani Marathe, Aaron Gokaslan, Jieyu Zhang, Khyathi Chandu, Thao Nguyen, Igor Vasiljevic, Sham Kakade, Shuran Song, Sujay Sanghavi, Fartash Faghri, Sewoong Oh, Luke Zettlemoyer, Kyle Lo, Alaaeldin El-Nouby, Hadi Pouransari, Alexander Toshev, Stephanie Wang, Dirk Groeneveld, Luca Soldaini, Pang Wei Koh, Jenia Jitsev, Thomas Kollar, Alexandros G. Dimakis, Yair Carmon, Achal Dave, Ludwig Schmidt, and Vaishaal Shankar. Datacomp-lm: In search of the next generation of training sets for language models, 2024. URL https://arxiv.org/abs/2406.11794.

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, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023a.

Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. Textbooks are all you need ii: phi-1.5 technical report, 2023b. URL https://arxiv.org/abs/2309.05463.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 21558–21572. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019. URL https://arxiv.org/abs/1907.11692.

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. Starcoder 2 and the stack v2: The next generation, 2024. URL https://arxiv.org/abs/2402.19173.

YINGWEI MA, Yue Liu, Yue Yu, Yuanliang Zhang, Yu Jiang, Changjian Wang, and Shanshan Li. At which training stage does code data help LLMs reasoning? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=KIPJKST4gw.

Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. Language models of code are few-shot commonsense learners. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (eds.), *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 1384–1403, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.90. URL https://aclanthology.org/2022.emnlp-main.90.

Luke Merrick, Danmei Xu, Gaurav Nuti, and Daniel Campos. Arctic-embed: Scalable, efficient, and accurate text embedding models, 2024. URL https://arxiv.org/abs/2405.05374.

Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, Manish Sethi, Xuan-Hong Dang, Pengyuan Li, Kun-Lung Wu, Syed Zawad, Andrew Coleman, Matthew White, Mark Lewis, Raju Pavuluri, Yan Koyfman, Boris Lublinsky, Maximilien de Bayser, Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Yi Zhou, Chris Johnson, Aanchal Goyal, Hima Patel, Yousaf Shah, Petros Zerfos, Heiko Ludwig, Asim Munawar, Maxwell Crouse, Pavan Kapanipathi, Shweta Salaria, Bob Calio, Sophia Wen, Seetharami Seelam, Brian Belgodere, Carlos Fonseca, Amith Singhee, Nirmit Desai, David D. Cox, Ruchir Puri, and Rameswar Panda. Granite code models: A family of open foundation models for code intelligence, 2024. URL https://arxiv.org/abs/2405.04324.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=iaYcJKpY2B_.

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

Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The fineweb datasets: Decanting the web for the finest text data at scale, 2024. URL https://arxiv.org/abs/2406.17557.

Nikhil Pinnaparaju, Reshinth Adithyan, Duy Phung, Jonathan Tow, James Baicoianu, Ashish Datta, Maksym Zhuravinskyi, Dakota Mahan, Marco Bellagente, Carlos Riquelme, and Nathan Cooper. Stable code technical report, 2024. URL https://arxiv.org/abs/2404.01226.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Snowflake AI Research. Snowflake arctic: The best llm for enterprise ai — efficiently intelligent, truly open, 2024. URL https://www.snowflake.com/en/blog/arctic-open-efficient-foundation-language-models-snowflake/.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023. URL https://arxiv.org/abs/2104.09864.

CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A. Choquette-Choo, Jingyue Shen, Joe Kelley, Kshitij Bansal, Luke Vilnis, Mateo Wirth, Paul Michel, Peter Choy, Pratik Joshi, Ravin Kumar, Sarmad Hashmi, Shubham Agrawal, Zhitao Gong, Jane Fine, Tris Warkentin, Ale Jakse Hartman, Bin Ni, Kathy Korevec, Kelly Schaefer, and Scott Huffman. Codegemma: Open code models based on gemma, 2024. URL https://arxiv.org/abs/2406.11409.

Teknium. Openhermes 2.5: An open dataset of synthetic data for generalist llm assistants. https://huggingface.co/datasets/teknium/OpenHermes2.5, 2023.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023. URL https://arxiv.org/abs/2307.09288.

Yuxiang Wei. hqcode. https://huggingface.co/datasets/yuxiang630/hqcode, 2024.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Harm de Vries, Leandro von Werra, Arjun Guha, and Lingming Zhang. Starcoder2-instruct: Fully transparent and permissive self-alignment for code generation. https://huggingface.co/blog/sc2-instruct, 2024a.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with OSS-instruct. In Ruslan Salakhutdinov, Zico Kolter, Katherine

Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 52632–52657. PMLR, 21–27 Jul 2024b. URL https://proceedings.mlr.press/v235/wei24h.html.

Wikipedia contributors. Plagiarism — Wikipedia, the free encyclopedia, 2004. URL https://en.wikipedia.org/w/index.php?title=Plagiarism&oldid=5139350. [Online; accessed 22-July-2004].

Chunqiu Steven Xia, Yinlin Deng, and Lingming Zhang. Top leaderboard ranking = top coding proficiency, always? evoeval: Evolving coding benchmarks via llm, 2024. URL https://arxiv.org/abs/2403.19114.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jianxin Yang, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Xuejing Liu, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, Zhifang Guo, and Zhihao Fan. Qwen2 technical report, 2024a. URL https://arxiv.org/abs/2407.10671.

Ke Yang, Jiateng Liu, John Wu, Chaoqi Yang, Yi Fung, Sha Li, Zixuan Huang, Xu Cao, Xingyao Wang, Heng Ji, and ChengXiang Zhai. If LLM is the wizard, then code is the wand: A survey on how code empowers large language models to serve as intelligent agents. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*, 2024b. URL https://openreview.net/forum?id=8dmNOD9hbq.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions, 2024. URL https://arxiv.org/abs/2406.15877.

A Examples of Synthetic Data

In this section, we present qualitative examples illustrating the disparities between the synthetic data generated during the enhanced pretraining stage and the high-quality data obtained through continued pretraining. The gray background code represents a high-quality code snippet annotated by the quality annotator, while the purple background code signifies the synthetic data of higher quality. A notable pattern of synthetic data is that it contains more documentation, natural language explanations, and comprehensive functionality. These features make the data more informative and boost the effectiveness of model learning.

A.1 Wait-Until Method

```
import time
from robot.utils import timestr_to_secs


class Wait:
    @staticmethod
    def until_true(condition, timeout, error_msg):
        """Helper to wait until given condition is met."""
        timeout = timestr_to_secs(timeout)
        max_wait = time.time() + timeout
        while True:
            if condition():
                break
            if time.time() > max_wait:
                raise AssertionError(error_msg)
            time.sleep(0.1)
```

```
"""
Module: wait_until

This module provides a set of functions to wait until a given condition
    is met.
It is designed to help developers write more robust and reliable code by
    providing
a simple way to wait for certain conditions to be met before proceeding
    with the
execution of their code.

Author: [Your Name]
Date: [Today's Date]
"""

import time
from typing import Callable, Optional

def timestr_to_secs(timestr: str) -> float:
    """
    Convert a time string to seconds.

    Args:
    timestr (str): A string representing a time duration (e.g., "10s", "1
    m", "1h").

    Returns:
    float: The time duration in seconds.

    Raises:
    ValueError: If the input string is not in the correct format.

    Examples:
    >>> timestr_to_secs("10s")
```

```
    10.0
    >>> timestr_to_secs("1m")
    60.0
    >>> timestr_to_secs("1h")
    3600.0
    """
    if timestr.endswith("s"):
        return float(timestr[:-1])
    elif timestr.endswith("m"):
        return float(timestr[:-1]) * 60
    elif timestr.endswith("h"):
        return float(timestr[:-1]) * 3600
    else:
        raise ValueError("Invalid time string format")


def wait_until_true(
    condition: Callable[[], bool],
    timeout: str,
    error_msg: Optional[str] = None,
    poll_interval: float = 0.1,
) -> None:
    """
    Wait until the given condition is met.

    Args:
    condition (Callable[[], bool]): A function that returns a boolean
    value.
    timeout (str): A string representing the maximum time to wait (e.g.,
    "10s", "1m", "1h").
    error_msg (Optional[str]): An optional error message to raise if the
    condition is not met.
    poll_interval (float): The interval to wait between checks (default:
    0.1 seconds).

    Raises:
    AssertionError: If the condition is not met within the specified
    timeout.

    Examples:
    >>> def is_true():
    ...     return True
    >>> wait_until_true(is_true, "10s")
    >>> def is_false():
    ...     return False
    >>> try:
    ...     wait_until_true(is_false, "10s", "Condition not met")
    ... except AssertionError as e:
    ...     print(e)
    Condition not met
    """
    timeout_in_secs = timestr_to_secs(timeout)
    max_wait = time.time() + timeout_in_secs
    while True:
        if condition():
            break
        if time.time() > max_wait:
            if error_msg:
                raise AssertionError(error_msg)
            else:
                raise AssertionError("Condition not met")
        time.sleep(poll_interval)


def wait_until_false(
```

```
    condition: Callable[[], bool],
    timeout: str,
    error_msg: Optional[str] = None,
    poll_interval: float = 0.1,
) -> None:
    """
    Wait until the given condition is false.

    Args:
    condition (Callable[[], bool]): A function that returns a boolean
    value.
    timeout (str): A string representing the maximum time to wait (e.g.,
    "10s", "1m", "1h").
    error_msg (Optional[str]): An optional error message to raise if the
    condition is not met.
    poll_interval (float): The interval to wait between checks (default:
    0.1 seconds).

    Raises:
    AssertionError: If the condition is not met within the specified
    timeout.

    Examples:
    >>> def is_false():
    ...     return False
    >>> wait_until_false(is_false, "10s")
    >>> def is_true():
    ...     return True
    >>> try:
    ...     wait_until_false(is_true, "10s", "Condition not met")
    ... except AssertionError as e:
    ...     print(e)
    Condition not met
    """
    wait_until_true(lambda: not condition(), timeout, error_msg,
    poll_interval)


if __name__ == "__main__":
    # Example usage:
    def is_true():
        return True

    def is_false():
        return False

    wait_until_true(is_true, "10s")
    try:
        wait_until_true(is_false, "10s", "Condition not met")
    except AssertionError as e:
        print(e)

    wait_until_false(is_false, "10s")
    try:
        wait_until_false(is_true, "10s", "Condition not met")
    except AssertionError as e:
        print(e)
```

A.2 QUICK SORT ALGORITHM

```
Array.prototype.quick_sort = function () {
    if (this.length < 2) { return this; }
```

```
    var pivot = this[Math.round(this.length / 2)];

    return this.filter(x => x <  pivot)
               .quick_sort()
               .concat(this.filter(x => x == pivot))
               .concat(this.filter(x => x >  pivot).quick_sort());
};
```

```
/**
 * Quick Sort Algorithm Implementation
 * =================================
 *
 * This module provides a high-quality implementation of the quick sort
   algorithm.
 * It includes a stable and efficient sorting function, as well as a
   utility function
 * for generating random arrays for testing.
 */

/**
 * Generates a random array of integers for testing.
 *
 * @param {number} length - The length of the array.
 * @param {number} min - The minimum value in the array.
 * @param {number} max - The maximum value in the array.
 * @returns {number[]} A random array of integers.
 */
function generateRandomArray(length, min, max) {
  return Array.from({ length }, () => Math.floor(Math.random() * (max -
    min + 1)) + min);
}

/**
 * Swaps two elements in an array.
 *
 * @param {any[]} arr - The array.
 * @param {number} i - The index of the first element.
 * @param {number} j - The index of the second element.
 */
function swap(arr, i, j) {
  [arr[i], arr[j]] = [arr[j], arr[i]];
}

/**
 * Partitions the array around a pivot element.
 *
 * @param {number[]} arr - The array.
 * @param {number} low - The low index.
 * @param {number} high - The high index.
 * @returns {number} The index of the pivot element.
 */
function partition(arr, low, high) {
  const pivot = arr[high];
  let i = low - 1;
  for (let j = low; j < high; j++) {
    if (arr[j] < pivot) {
      i++;
      swap(arr, i, j);
    }
  }
  swap(arr, i + 1, high);
  return i + 1;
}
```

```
/**
 * Recursively sorts the array using the quick sort algorithm.
 *
 * @param {number[]} arr - The array.
 * @param {number} low - The low index.
 * @param {number} high - The high index.
 */
function quickSort(arr, low, high) {
  if (low < high) {
    const pivotIndex = partition(arr, low, high);
    quickSort(arr, low, pivotIndex - 1);
    quickSort(arr, pivotIndex + 1, high);
  }
}

/**
 * Sorts the array using the quick sort algorithm.
 *
 * @param {number[]} arr - The array.
 * @returns {number[]} The sorted array.
 */
function sort(arr) {
  quickSort(arr, 0, arr.length - 1);
  return arr;
}
```

A.3 REVERSE WORDS

```
# https://leetcode.com/problems/reverse-words-in-a-string-iii/

class Solution:
    def reverseWords(self, s: str) -> str:
        # return self.solution1(s)
        # return self.solution3(s)
        return self.solution2(s)

    # Time O(n), Space O(n)
    def solution2(self, s):
        res = []
        word = []

        for char in s:
            if char == ' ':
                res.extend(reversed(word))
                res.append(' ')
                word = []
            else:
                word.append(char)

        res.extend(reversed(word))

        return ''.join(res)


    # Time O(n), Space O(n)
    def solution1(self, s):
        if len(s) <= 1:
            return s

        counter = 0
        res = []

        def extract_word(from_index, count):
```

```
            for prev in range(count):
                res.append(s[from_index - prev])

        for idx, char in enumerate(s):
            if char == ' ':
                extract_word(idx - 1, counter)
                res.append(' ')
                counter = 0
            else:
                counter += 1

        extract_word(len(s) - 1, counter)

        return ''.join(res)

    def solution3(self, s):
        return ' '.join([v[::-1] for v in s.split()])
```

```
# Reverse Words in a String III

class Solution:
    def reverse_words(self, s: str) -> str:
        """
        Reverses the order of characters in each word within a given
    string.

        Args:
            s (str): The input string.

        Returns:
            str: The string with each word's characters reversed.

        Example:
            >>> solution = Solution()
            >>> solution.reverse_words("Let's take LeetCode contest")
            "s'teL ekat edoCteeL tsetnoc"
        """
        return ' '.join(word[::-1] for word in s.split())

    def reverse_words_iterative(self, s: str) -> str:
        """
        Reverses the order of characters in each word within a given
    string using iteration.

        Args:
            s (str): The input string.

        Returns:
            str: The string with each word's characters reversed.

        Example:
            >>> solution = Solution()
            >>> solution.reverse_words_iterative("Let's take LeetCode
    contest")
            "s'teL ekat edoCteeL tsetnoc"
        """
        result = []
        word = []
        for char in s:
            if char == ' ':
                result.extend(reversed(word))
                result.append(' ')
                word = []
            else:
```

```
                word.append(char)
        result.extend(reversed(word))
        return ''.join(result)

    def reverse_words_recursive(self, s: str) -> str:
        """
        Reverses the order of characters in each word within a given
    string using recursion.

        Args:
            s (str): The input string.

        Returns:
            str: The string with each word's characters reversed.

        Example:
            >>> solution = Solution()
            >>> solution.reverse_words_recursive("Let's take LeetCode
    contest")
            "s'teL ekat edoCteeL tsetnoc"
        """
        if not s:
            return s
        if s[0] == ' ':
            return ' ' + self.reverse_words_recursive(s[1:])
        if ' ' not in s:
            return s[::-1]
        space_index = s.index(' ')
        return s[:space_index][::-1] + ' ' + self.reverse_words_recursive
    (s[space_index + 1:])
```

A.4 SKLEARN USAGE WITH JOBLIB

```
import joblib

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB


class Model:
    def __init__(self, vocab_size=10_000):
        self.vocab_size = vocab_size
        self.clf = None
        self.vectorizer = None

    def train(self, X_train, y_train):
        self.vectorizer = TfidfVectorizer(max_features=self.vocab_size)
        X_train = self.vectorizer.fit_transform(X_train)

        self.clf = MultinomialNB()
        self.clf.fit(X_train, y_train)

    def predict(self, X):
        X = self.vectorizer.transform(X)

        self.y_pred = self.clf.predict(X)
        return self.y_pred

    def predict_proba(self, X):
        X = self.vectorizer.transform(X)

        self.y_proba = self.clf.predict_proba(X)
        return self.y_proba
```

```
    def save_model(self):
        joblib.dump(self.vocab_size, "models/vocab.pkl")
        joblib.dump(self.vectorizer, "models/vecorizer.pkl")
        joblib.dump(self.clf, "models/model.pkl")

    @staticmethod
    def load_model():
        model = Model()
        model.vocab_size = joblib.load("models/vocab.pkl")
        model.vectorizer = joblib.load("models/vecorizer.pkl")
        model.clf = joblib.load("models/model.pkl")
        return model
```

```
import joblib
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report

class TextClassifier:
    def __init__(self, vocab_size=10_000):
        self.vocab_size = vocab_size
        self.vectorizer = TfidfVectorizer(max_features=vocab_size)
        self.clf = MultinomialNB()

    def train(self, X, y):
        """Train the model on the given data.

        Args:
            X (list): List of text samples.
            y (list): List of corresponding labels.

        Returns:
            tuple: Training accuracy and classification report.
        """
        X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2, random_state=42)
        X_train = self.vectorizer.fit_transform(X_train)
        self.clf.fit(X_train, y_train)

        X_test = self.vectorizer.transform(X_test)
        y_pred = self.clf.predict(X_test)
        accuracy = accuracy_score(y_test, y_pred)
        report = classification_report(y_test, y_pred)
        return accuracy, report

    def predict(self, X):
        """Make predictions on the given data.

        Args:
            X (list): List of text samples.

        Returns:
            list: List of predicted labels.
        """
        X = self.vectorizer.transform(X)
        return self.clf.predict(X)

    def save_model(self, path="models/"):
        joblib.dump(self.vectorizer, path + "vectorizer.pkl")
        joblib.dump(self.clf, path + "model.pkl")

    @staticmethod
```

```
    def load_model(path="models/"):
        model = TextClassifier()
        model.vectorizer = joblib.load(path + "vectorizer.pkl")
        model.clf = joblib.load(path + "model.pkl")
        return model
```

A.5 JAVA TEXT VIEW BINDING

```
package android.databinding.testapp.vo;

import android.databinding.Bindable;

public class AutoCompleteTextViewBindingObject extends
    BindingAdapterBindingObject {
    @Bindable
    private int mPopupBackground;

    @Bindable
    private int mCompletionThreshold = 1;

    public int getCompletionThreshold() {
        return mCompletionThreshold;
    }

    public int getPopupBackground() {
        return mPopupBackground;
    }

    public void changeValues() {
        mPopupBackground = 0xFF23456;
        mCompletionThreshold = 5;
        notifyChange();
    }
}
```

```
package com.example.databinding;

import android.databinding.Bindable;
import android.databinding.Observable;

/**
 * Represents a binding object for AutoCompleteTextView.
 */
public class AutoCompleteTextViewBindingObject extends Observable {

    // Default values
    private static final int DEFAULT_POPUP_BACKGROUND = 0xFFFFFFFF;
    private static final int DEFAULT_COMPLETION_THRESHOLD = 1;

    @Bindable
    private int mPopupBackground;

    @Bindable
    private int mCompletionThreshold;

    /**
     * Constructs a new AutoCompleteTextViewBindingObject with default
    values.
     */
    public AutoCompleteTextViewBindingObject() {
        this(DEFAULT_POPUP_BACKGROUND, DEFAULT_COMPLETION_THRESHOLD);
    }
```

```
    /**
     * Constructs a new AutoCompleteTextViewBindingObject with the
    specified values.
     *
     * @param popupBackground The background color of the popup.
     * @param completionThreshold The minimum number of characters to
    show the popup.
     */
    public AutoCompleteTextViewBindingObject(int popupBackground, int
    completionThreshold) {
        this.mPopupBackground = popupBackground;
        this.mCompletionThreshold = completionThreshold;
    }

    /**
     * Gets the background color of the popup.
     *
     * @return The background color of the popup.
     */
    public int getPopupBackground() {
        return mPopupBackground;
    }

    /**
     * Sets the background color of the popup.
     *
     * @param popupBackground The new background color of the popup.
     */
    public void setPopupBackground(int popupBackground) {
        if (this.mPopupBackground != popupBackground) {
            this.mPopupBackground = popupBackground;
            notifyPropertyChanged(BR.popupBackground);
        }
    }

    /**
     * Gets the minimum number of characters to show the popup.
     *
     * @return The minimum number of characters to show the popup.
     */
    public int getCompletionThreshold() {
        return mCompletionThreshold;
    }

    /**
     * Sets the minimum number of characters to show the popup.
     *
     * @param completionThreshold The new minimum number of characters to
     show the popup.
     */
    public void setCompletionThreshold(int completionThreshold) {
        if (this.mCompletionThreshold != completionThreshold) {
            this.mCompletionThreshold = completionThreshold;
            notifyPropertyChanged(BR.completionThreshold);
        }
    }

    /**
     * Changes the values of the binding object.
     *
     * @param popupBackground The new background color of the popup.
     * @param completionThreshold The new minimum number of characters to
     show the popup.
     */
```

```
    public void changeValues(int popupBackground, int completionThreshold
    ) {
        setPopupBackground(popupBackground);
        setCompletionThreshold(completionThreshold);
    }
}
```