A Road for LLM SQL Bug-Fixing Enhancing

Anonymous ACL submission

Abstract

Code Large Language Models (Code LLMs), such as Code llama and DeepSeek-Coder, have demonstrated exceptional performance in the code generation tasks. However, most existing models focus on the abilities of generating correct code, but weak in bug code repair. In this paper, we introduce a suit of methods which enhance LLM's bug-fixing abilities on SQL code, which are mainly consisted of two parts: A Progressive Dataset Construction (PDC) from scratch and Dynamic Mask Supervised Fine-tuning (DM-SFT). PDC proposes two data expansion methods from the perspectives of breadth first and depth first respectively. DM-SFT introduces an efficient bug-fixing supervised learning approach, which effectively reduce the total training steps and mitigate the "mental disorientation" in SQL code bug-fixing training. In our evaluation, the code LLM models trained on these two methods have exceeds all current best performing model which size is much larger.

1 Introduction

Recently, large language models (LLMs) trained on diverse Internet scale datasets and code repositories have achieved remarkable success. Meanwhile, Large Language Models for Code (Code LLMs) have rapidly emerged as powerful assistants for writing code. However, most of code LLMs are focus on the capabilities of "writing" code. Such as text2code (Given a natural language text, then model write the expected code), code completion. Code bug-fixing receives less attention especially compared with the above. Moreover, we discovered those open source pretrained code LLMs, like DeepSeek-Coder (Guo et al., 2024), WizardCoder (Luo et al., 2023) and Code llama (Roziere et al., 2023), are very limited in bug-fixing capability.

In this paper, we especially focus on SQL code bug-fixing task. Due to the complex nested query structure, SQL code bugs are more difficult to solve compared with other programing languages. Furthermore, SQL code is less dependent on third-party packages, which mitigates the incidence of unsolvable bugs that arise due to insufficient information from third-party toolkits. We formulate the SQL code bug-fixing task as Equation 1.

$$SQL_{correct} = f(Schema, SQL_{bug}, R) \quad (1)$$

Where the f represents your bug-fixing model. Schema means the related tables schemas in your bug SQL code. SQL_{bug} denote the SQL code which contains some bugs need to be fixed. R is the return massage by your SQL execution system when you run your bug SQL code. $SQL_{correct}$ is the bug-fixing model's output, which is expected the right SQL code.

We propose a set of methods to enhance the bug-fixing capabilities of Large Language Models (LLMs). This includes a method for mining and collecting supervised data, termed Progressive Dataset Construction (PDC), and an efficient training method based on dynamic masking, known as Dynamic Mask-SFT (DM-SFT). Additionally, we discuss an effective approach to reduce hallucination outputs when applying open-source code LLMs to specific domains—continue pre-train with domain-specific data. Experiments show that training with data collected via the PDC method generally improved the SQL bug-fixing capabilities of open-source code LLMs by nearly $+50\%$. The Dynamic Mask-SFT training method further enhanced model performance by approximately $+10\%$ relative to the default generative SFT. Continue pretrain effectively reduced the occurrence of hallucinatory answers.

2 Related Work

Code bug fixing with deep learning has gained increasing attention as the capabilities of pre-trained

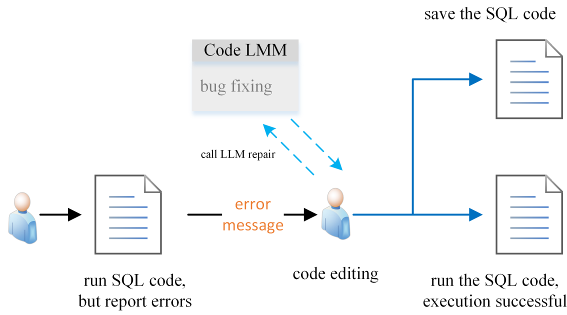

Figure 1: The initial training data collection via user behavior logs mining.

language models continue to improve. However, the code bug fixing ability of these models often falls short due to the lack of large-scale annotated data. Several methods have been proposed to train a model to generate bugs from correct code, thereby obtaining annotated data for the transition from bug code to correct code.

Some recent studies have focused on training a model to transform correct code into bug code, thereby generating annotated data for code bug fixing learning. BUGLAB (Allamanis et al., 2021) propose a self-supervised approach which trains robust bug detectors by co-training a bug selector that learns to create hard-to-detect bugs. Break-It-Fix-It (Yasunaga and Liang, 2021) is a similar approach that involves the collaborative training of both bug fixers and bug generators. To the best of our knowledge, these methods have scarcely ventured into the realm of SQL language, and in our practice, it has proven challenging to train a model capable of generating bug SQL with diffusion characteristics (not only diversity but also close to the distribution of human bugs). This may be attributed to the inherent differences between SQL code and most object-oriented programming languages.

Additionally, some approaches attempt to address the code bug repair problem from the perspective of an agent. A typical example is RepairAgent (Bouzenia et al., 2024) and SELF-DEBUGGING (Chen et al., 2023), which treats the LLM as an agent capable of autonomously planning and executing actions to fix bugs by invoking suitable tools. However, with SQL code, executing a large query can be time-consuming. Therefore, repeatedly debugging and running the code to resolve bugs is clearly impractical.

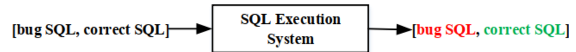

Figure 2: Execution filter for data quality.

3 Progressive Dataset Construction

In this section, we introduce a set of data collection methods called Progressive Dataset Construction (PDC). The methods mainly include two parts: diverse collecting from online system (breadth first) and oriented generation of offline mining (depth first). The diverse collecting through automated methods ensures the diversity coverage and sustainable scalability of the training datasets, thereby maintaining a consistent alignment between the distribution of training data and the behaviors of online users. The oriented generation method is primarily used for data augmentation in cases where the model performs poorly in evaluation and online serving. This approach requires the assistance of mature code LLM (Large Language Model) and some SQL corpora recall methods.

3.1 Diverse Collecting

To complete the collection of initial training data, we designed a set of rules to mine online user behavior logs. As shown in Figure 1, when user encounter execution error while running SQL code, the system will report and log a snapshot of the bug code and error message. Subsequently, users would typically edit and correct the code until it successfully runs in the next attempt. Therefore, we can extract an extensive array of $(bugSQL, correctSQL)$ pairs which generated by users based on this behavior.

Additionally, we observed that since the SQL execution environment integrates certain syntax checking capabilities, some users, when encountering syntax errors, tend to modify their code until the highlighted syntax error prompts disappear and then save their code, rather than executing it again for confirmation. Consequently, we also consider the most recent operation of 'save code' after a code execution error as a behavioral signal for mining correct SQL. As illustrated in Figure 1.

After acquiring data samples from online user behavior logs, we apply an execution filter as Figure 2. This process retains $(bugSQL, correctSQL)$ pairs where the bug SQL triggers an error, and the correct SQL executes successfully (Red font

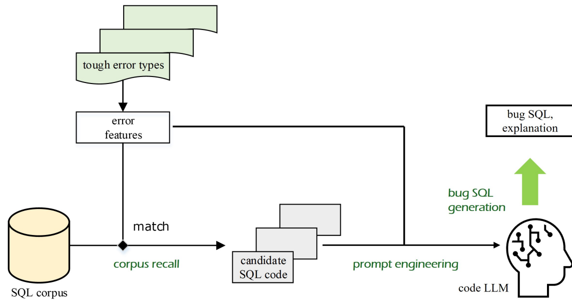

Figure 3: Overview of oriented generation method.

indicates execution failure, green font signifies successful execution). Moreover, to ensure data quality, we also removed samples where the disparity between the bug SQL code and the correct SQL code was excessively large.

Lastly, we conduct a manual sampling inspection of the filtered data. If the [bug SQL, correct SQL] pairs achieve a pass rate of over 85%, they meet our quality standards and are deemed suitable for model training.

Diverse collecting samples for bug SQL repair directly from online user behavior ensures an excellent coverage of diversity. It aligns with the natural data distribution in real service scenarios, which is crucial for model training. Once our SQL code bug-fixing model is deployed, we pay more attention on cases where users disregard the model's suggested modifications and proceed with manual edits. This behavior often implies that the model's proposed changes did not meet the user's expectations. Diverse collection will persist as a crucial method of sample gathering throughout the continuous iteration process of the model.

3.2 Oriented Generation

The oriented generation method is primarily used for data augmentation aimed at difficult cases. These types of cases typically include unique syntax features of internal systems and relatively rare long-tail error types, among others. This method comprises two processes: the recall of appropriate SQL corpora and the generation of bug SQL, which relies on the code LLM.

The original SQL corpus is derived from a vast accumulation of correctly executable SQL code, which is written by historical users in the data query system. As illustrated in Figure 3, we perform oriented data augmentation for certain high-frequency bug error types and some bug types that are difficult for the model to resolve. The oriented generation data augmentation approach primarily encompasses the following steps:

(1) **Identify the bug types that need data augmentation via oriented generation.** For instance, during the cold-start phase, we primarily target those long-tail bug types that occur infrequently. After the model is deployed as a serving, we mainly focus on those types where the model's correction accuracy is not high.

(2) **Create an "error feature" for each bug type.** The error feature is primarily related to the corpus recall algorithm you use. For example, you can use syntax keywords for recall, such as using the keyword "group by" to match SQL code suitable for generating "group by" errors.

(3) **Recall the candidate SQL code for every bug type.** We employ appropriate rule based matching algorithm to pair a rich corpus of SQL code with each bug type via "error feature". The accuracy of matching varies across different bug types. We select different matching algorithms for different bug types based on the certain circumstances.

(4) **Generate a rich set of bug SQL samples for each bug type.** This step requires the assistance of a robust code LLM for the generation of bug SQL code. In our practice, the quality of generated bug SQL is highly correlated with the prompt. We provide a reference prompt for generating bug SQL in Appendix A.1 used in our internal code fundamental LLM for bug SQL generation.

The diverse collecting and oriented generation methods respectively accomplish the supervised dataset construction for the SQL bug fixing task from the perspectives of breadth-first and depth-first approaches. As you can see, these two data construction methods remain effective even after the model is deployed as an SQL bug fixing tool. The diverse collecting method, based on user behavior, can collect unsatisfactory samples which modified by the users in a crowdsourcing-like manner. Meanwhile, oriented generation can specifically enhance the types of bugs where the model's performance is subpar. The collected data can be utilized to improve the model's performance. The enhancement of model performance, in turn, affects the distribution of the data collecting. Therefore, this is a progressive dataset construction method.

4 Dynamic Mask Supervised Fine-tuning

In this section, we present a detailed introduction to an efficient training method for LLM SQL code bug fixing, which we refer to as dynamic mask supervised fine-tuning (DM-SFT). The Figure 4 provides a detailed comparison of dynamic mask SFT with the default generative SFT in terms of training methodology and loss calculation. As outlined in the introduction section, in our task formulation, the model input consists of a bugfix prompt composed of three pieces of information: [tables DDL, bug SQL, report error]. The model output is the complete, corrected SQL code. In most cases, most of the code lines in the correct SQL and the bug SQL are identical, with only a few lines typically requiring modification.

In the training data we collected, the distribution of the number of code lines that need to be modified in the correct SQL compared to the corresponding bug SQL (we name it diff lines) is shown in Appendix A.2 Figure 9 . As shown in the figure, cases where the number of diff lines is less than 5 account for more than 92% of the instances. Therefore, most of the correct code that the model needs to predict as output has already appeared in the model input prompt (bug SQL). In the default generative supervised fine-tuning training process, all tokens in the output answer are equally important in the calculation of the final loss. This can lead to a series of issues such as slow convergence and unstable training results. We will discuss these problems in detail in experimental section.

To address these issues, we propose a code bug repair training method called dynamic mask SFT. During the model training process, we divide the correct SQL code that the model is expected to predict post bug-fixing into two categories in line-by-line basis:

(i) **Consistent lines** representing the lines of code that remain unchanged when compared to the original bug-infested code.

(ii) **Diff lines** representing the lines of code that require modifications when compared to the original bug-infested code.

Given a bug SQL code, related tables schema, report error and corresponding correct SQL code, we use $(l_0, l_1, l_2, \cdots, d_0, \cdots, d_m, \cdots l_n), m \leq n$ denoting the correct code lines. The $l_i, i \in [0, n]$ represents the consistent lines and $d_j, j \in [0, m]$ represents the diff lines. We use u to denote tokens of consistent lines, and v to denote tokens of diff lines. Equation 2 shows the loss function of dynamic mask SFT.

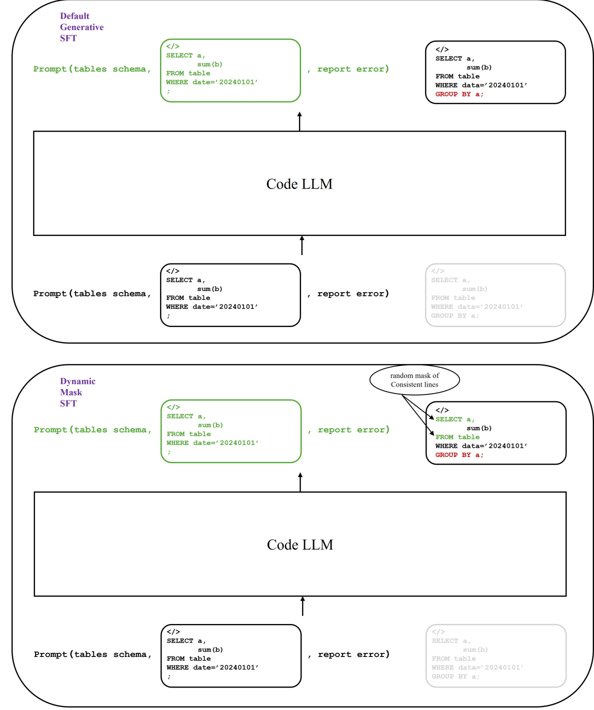

Figure 4: A comparison between default generative SFT (upper part) and dynamic mask SFT (lower part) of code bug fixing task.

$$L_1 = -\sum \log P(u_{k+1} \mid u_k, u_{k-1}, \ldots, u_0) * a(l(u_{k+1})) \quad (2)$$

$$a(l_i) = \begin{cases} 0 & \text{p} \\ 1 & \text{(1-p)} \end{cases} \quad (3)$$

Where $a(l_i)$ is the mask weight of line l_i as Equation 3, and mask weight of all tokens in line l_i are the same. The p is random mask ratio factor, used to control the proportion of masked code lines. $l(u_{k+1})$ represents the line number of code where token u_{k+1} is located. In Equation 2, L_1 represents the language model loss of the consistent lines (after dynamic masking). In Equation 4, L_2 represents the language model loss of the diff lines.

$$L_2 = -\sum logP(v_{k+1} \mid v_k, v_{k-1}, \ldots, v_0) \quad (4)$$

$$L = L_1 + L_2 \quad (5)$$

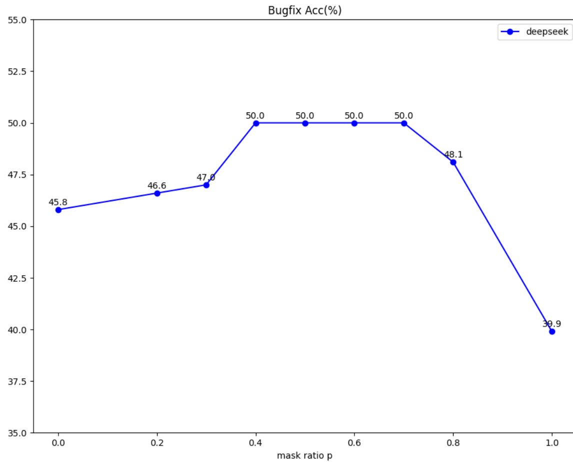

Figure 5: Bug fixing evaluation results with different value of random mask ratio factor p.

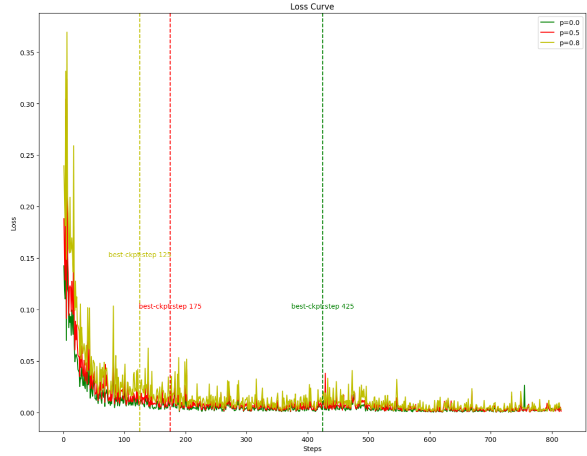

Figure 6: Loss reduction curves and best bug fixing performance steps across typical random mask ratio factors p during model training.

The final total loss L, as shown in Equation 5, is composed of L_1 and L_2.

Figure 4 clearly illustrates the similarities and differences between the dynamic mask SFT process and the default bug fixing SFT process. The correct SQL code that the model needs to predict, and output is set in grey. In the entire model output label, the parts that do not need to calculate loss are highlighted in green (input prompt and masked code lines randomly selected with probability p).

5 Experiments and Results

In this section, we will provide a detailed overview of our experiments setup and the related results. This section is divided into two main parts: first, we will discuss the ablation experiments related to the effectiveness of PDC and DM-SFT; subsequently, we will briefly introduce the issue of hallucinatory modifications found in external open-source models during the model evaluation process, as well as the mitigation for hallucination phenomena through Continue Pre-train (CPT) (Ke et al., 2023) using internal data. In addition, we made some analyses for the experiment results and processes, providing several analytical perspectives and conclusions.

5.1 PDC and SFT Experiments

We demonstrate the efficacy of the suit of methods (PDC & DM-SFT) through a series of experiments. We collected 3k diverse samples through the diverse collecting method and 300+ oriented enhancement samples based on code LLM through the oriented generation method. Based on these 3.3k data, we conducted a series of ablation experiments to verify the effectiveness of DM-SFT and the impact of various parameter settings on the model training results.

We use DeepSeek-Coder-instruct (6.7b) as the fundamental model and carry out the training experiments on a cluster of 32 × NVIDIA A800 80GB GPUs using the DeepSpeed (Rajbhandari et al., 2020) framework stage 3. In terms of hyperparameters setting, we used batch size = 32, learning rate = 1.2e-5, and AdamW optimizer (Loshchilov and Hutter, 2017) with $adam_beta1 = 0.9$ and $adam_beta2 = 0.95$ (more detailed experimental parameter configurations, please refer to the information of code release in the final parts of this section).

We have an evaluation dataset of size 268, the data of which comes from bug SQL code genuinely submitted by online users. The ground truth in this dataset is precisely annotated by experienced SQL engineers. During the model development stage, we used machine automatic evaluation (a method based on AST semantic comparison) results to select the approximate best training steps of the model. Since there may be many different ways to fix a bug, the final model's bug fixing accuracy was determined through manual evaluation of experienced SQL engineers.

In the evaluation, we first assessed the bug-fixing capabilities of the currently best open code LLMs, as well as our powerful internal code LLM that have not yet been opened, without any bug-fixing SFT enhancement. This serves as a baseline for comparing and evaluating the effectiveness of our PDC data collection methods. On the other hand, through ablation experiments, we compared the im-

Method	Model	Size	Acc
Pretrain	gemma	7B	20.8%
	StarCoderBase	7B	27.9%
	StarCoder2	7B	28.3%
	CodeQwen1.5-Chat	7B	29.8%
	DeepSeek-Coder-instruct	6.7B	30.2%
	DeepSeek-Coder-instruct	33B	30.9%
	WizardCoder-V1.1	33 B	30.9%
	internal code LLM	*	41.4%
SFT	gemma	7B	30.9%
	StarCoderBase	7B	35.4%
	CodeQwen1.5-Chat	7B	44.4%
	DeepSeek-Coder-instruct	6.7B	45.8%
DM-SFT	CodeQwen1.5-Chat	7B	48.1%
	DeepSeek-Coder-instruct	6.7B	50.0%

Table 1: Accuracy of different models and training methods.

pact of dynamic mask SFT and default generative SFT on training, as well as the effect of the value of random mask ratio factor p on model training.

We conducted independent tests on various models, and all output results from these models were subjected to blind manual evaluation (evaluators were unaware of which model each answer came from, and each bug-fixing sample was cross reviewed by three individuals). The final fixing accuracy of each model on the 268-sample evaluation dataset are shown in Table 1.

It is evident that among the models with around the 7B parameters, DeepSeek-Coder-6.7B-instruct achieves the highest fixing accuracy. Additionally, we observe that the larger 33B model does not exhibit significant improvement compared to the 7B model. Using DeepSeek-Coder-6.7B-instruct as the foundational model, we conducted both default generative dynamic mask SFT training on the 3.3k training dataset collected through the PDC method.

As observed in Table 1, the 3.3k data samples collected through the PDC method (Diverse collecting & Oriented generation) significantly enhanced the accuracy of the DeepSeek-Coder model in the bug-fixing task. The accuracy improved from 30.2% in the original model to 45.8%, representing a relative increase of 51.6% in the capability to fix bugs in SQL code. We also conducted SFT experiments on other models with parameter sizes around 7B, and the findings were consistent.

Furthermore, we employed dynamic mask SFT to train models on DeepSeek-Coder-6.7B-instruct and CodeQwen1.5-7B-Chat, which are among the best-performing models with parameter sizes around 7B. Results from manual evaluations indicate that dynamic mask SFT can enhance the model's bug fixing capability by approximately 10% compared to the default generative SFT training (DeepSeek-Coder-6.7B-instruct: 45.8%→50.0%, CodeQwen1.5-7B-Chat: 44.4%→48.1%).

Taking the best-performing DeepSeek-Coder-6.7B-instruct model as the foundation model, we trained the model under different values of p and evaluated its optimal bug-fixing capability, with the results presented in Figure 5. After that, we compared the impact of different random mask ratio factors p on per-token loss reduction process, as illustrated in Figure 6. From Figure 5 and Figure 6, we can draw the following three conclusions:

(i) In the early stages of training (less than 400 steps), the higher value of p, come up with the greater the per-token loss. In the later stages (after 500 steps), the per-token loss converges regardless of the value of p. This phenomenon is intuitive as the mask ratio factor effectively amplifies the weight of the diff code tokens loss with pre-trained LLM, the loss of diff code is greater than the loss of consistent code that has appeared in the prompt. As the model gradually converges, the difference in per-token loss between the two diminishes.

(ii) Generally, the higher value of p, the fewer training steps are required to reach the check-

```sql
37   COUNT(1) as pay_freq,
38   COUNT(
39   distinct DATE_FORMAT(FROM_UNIXTIME(pay_finish_time), 'yyyyMMdd')
40   ) as pay_days
41   from ****.************_df
42   where date between '${date}' and '${date+3}'
43   and DATE_FORMAT(FROM_UNIXTIME(pay_finish_time), 'yyyyMMdd') <= '${date+3}'
44   and DATE_FORMAT(FROM_UNIXTIME(pay_finish_time), 'yyyyMMdd') >= '${date}'
45   group by
46   user_id,
47   mix_id
48   )t3
49 on t1.user_id=t3.user_id
50  and CAST(t1.mix_id AS STRING)=CAST(t3.mix_id AS STRING)
51 limit 90000000;
```

```sql
37   COUNT(1) as pay_freq,
38   COUNT(
39   distinct DATE_FORMAT(FROM_UNIXTIME(pay_finish_time), 'yyyyMMdd')
40   ) as pay_days
41   from ****.************_df
42   where date between '${date}' and '${date+3}'
43   and DATE_FORMAT(FROM_UNIXTIME(pay_finish_time), 'yyyyMMdd') <= '${date+3}'
44   and DATE_FORMAT(FROM_UNIXTIME(pay_finish_time), 'yyyyMMdd') >= '${date}'
45   group by
46   user_id,
47   mix_id
48   )t3
49 on t1.user_id=t3.user_id
50  and CAST(t1.mix_id AS STRING)=CAST(t3.mix_id AS STRING)
51 limit 900000000;
```

Figure 7: Hallucination modification by DeepSeek-Coder. Left: Output from internal code LLM (limit value consistent with original code). Right: Output from DeepSeek-Coder-Bugfix (limit value erroneously increased by an additional 0 character).

point with the best bug-fixing capability. This is one of the most valuable features of the dynamic mask SFT method, in addition to its ability to enhance the model's bug-fixing capabilities. This implies that we can achieve better model performance with lower computational costs and reduced energy consumption.

(iii) From Figure 5, we can clearly see that when the value of p is between $[0.4, 0.7]$, all the trained models achieve optimal performance. When the value of p is 1 (completely ignoring the loss of identical code lines), the performance of the model is worse than those using the default generative SFT (where p is 0).

The manual evaluation results of the ablation experiments shown in Table 1 have adequately demonstrated the effectiveness and applicability of the Progressive Dataset Construction (PDC) data collection method and the Dynamic Mask SFT (DM-SFT) training approach in enhancing the LLM's capability for SQL code bug fixing. It is noteworthy that by appropriately setting the parameter p, the dynamic mask SFT method can enhance the model's bug fixing capability while significantly reducing the training time. This allows the model to achieve optimal performance at earlier training steps. Such a feature is particularly appealing to the model developers in the era of LLMs, where computational resources are highly pricey.

5.2 Continue Pre-train

Throughout the model development phase, we compared the bug fixing capabilities of DeepSeek-Coder-6.7B-instruct and our internal code LLM on a case-by-case basis after fine-tuning them on the same dataset. We found that compared to the internal code LLM, DeepSeek-Coder is more prone

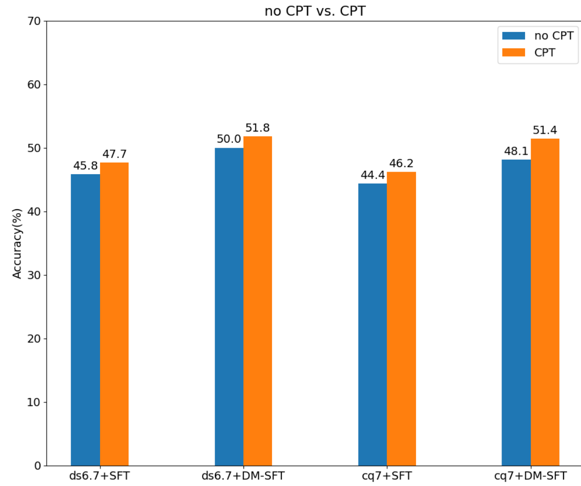

Figure 8: Performance differences of models with and without continued Pre-train on domain-specific corpus (pre-train before SFT/DM-SFT), where 'ds6.7' represents DeepSeek-Coder-6.7B-instruct and 'cq7' denotes CodeQwen1.5-7B-Chat.

to producing hallucination outputs when generating correct SQL code. Figure 7 presents a typical example, where the left side shows the correct code snippet predicted by the internal code LLM (have been fine-tuned), and the right side shows the correct code snippet predicted by the DeepSeek-Coder-Bugfix (have been fine-tuned) model. The constant value 90000000 of the original code was erroneously increased by an additional 0 in DeepSeek-Coder-Bugfix model's prediction.

Through the analysis, we discovered that the differences in performance between the two foundation models which have been fine-tuned with the same supervised data may stem from their familiarity for the domain-specific SQL code style and distribution (the internal model's pre-train corpus includes domain-specific code data). To validate this hypothesis, we have mined, cleaned, and deduplicated a dataset from internal scenarios to obtain a SQL code corpus with size of 53k. To ensure the rigor of the experiment, we carefully inspected these entries to guarantee that there would be no overlap with the 268 samples in evaluation dataset.

We conduct continue pre-train (CPT) (Ke et al., 2023) on the 53k domain-specific corpus which we have cleaned and use DeepSeek-Coder-6.7B-instruct and CodeQwen1.5-7B-Chat as foundation models. We then compared the capabilities of the models before and after Continued Pre-training, as illustrated in Figure 8. We made some adjustments to the learning rate, setting it to $1.5e-5$ for con-

tinue pre-train and later adjusting it to $1.0e-5$ for subsequent SFT/DM-SFT. Through comparison, it is evident that after continue pre-train with domain-specific data, the four combinations of models and training methods achieved a bug-fixing accuracy improvement range of $1.8\% \sim 3.4\%$. Additionally, the number of bad cases which involved with hallucination modification has decreased across all models.

There's worth mentioning that when using different models for Continue Pre-train, we adhered to the same input formats as their original pre-train. Additionally, we compared two training methods: training all parameters versus training only the parameters outside of the embedding layer during Continue Pre-train. Although the parameters of the embedding layer constitute only a small portion of the total parameters in most LLMs (for example, in DeepSeek-Coder 6.7b, the embedding layer accounts for approximately 1.96% of whole parameters), training with the embedding layer parameters frozen has proven challenging to achieve the expected results in our practice. In Appendix A.2 Figure 10, we have documented the training loss decline curves for both full parameter Continue Pre-train and Continue Pre-train with only non-embedding layer parameters updated. It is evident that training with only non-embedding layer parameters updated struggles to converge, whereas full parameter update in Continue Pre-train demonstrates good convergence.

Finally, all source code related to our experiments will be made publicly available in the corresponding GitHub repository[1]. All training and evaluation data used will be released later after being anonymized by data engineers.

6 Conclusion

In this paper, we innovatively propose a set of methods to enhance the LLM's capability for SQL bug fixing, encompassing both data construction and model training aspects. In terms of data construction, we propose two approaches: a breadth-first diverse collecting method and a depth-first oriented generation method. The diverse collecting method employs a rigorous strategy to mine from online users' behavior, obtaining bug fixing annotated data that aligns with real-world scenario distributions. The oriented generation method is primarily used for targeted data augmentation to address specific weaknesses in the model's capabilities. Both methods require minimal manual labor, making them semi-automated and sustainable approaches for data construction and iteration. Therefore, we have named this suit of data construction methods Progressive Dataset Construction (PDC). In terms of training methodology, we propose the dynamic mask SFT training method, which is generally applicable to generative code bug fixing tasks. Compared to the default generative SFT method, this approach can enhance the model's bug fixing capability by nearly 10% under the same training data. Additionally, it significantly reduces the training time required to achieve optimal model performance.

[1] https://github.com/*/*

Limitations

Only generate the modification code lines We attempted a highly efficient and intuitively appealing approach that involves generating only the correct code for the diff sections. Specifically, our approach required the model to output the lines of code that needed modification and the corrected code after changes. This definition could handle all code rewriting operations, including additions (where a single line of original code is replaced by multiple lines), deletions (where multiple lines of original code are replaced by an empty string), and modifications (where multiple lines of original code are replaced by multiple lines of new code). Unfortunately, this method resulted in impaired model performance due to the lack of context in the outputs, making it challenging to achieve the accuracy of generating complete code, both in prompt engineering experiments on GPT-4 (Achiam et al., 2023) and in SFT training on open-source code LLMs.

Token level dynamic mask SFT A pertinent question arises as to why consistent lines cannot use token-level dynamic masking and must instead be masked by code lines. Indeed, in our earliest practices, we masked at the token level. However, perplexingly, models masked at the token level struggled to converge, and during evaluations, a portion of the samples consistently failed to generate complete and usable code. This remains a puzzle we have not fully resolved. We hypothesize that for programming languages, a line may correspond to a more complete semantic module, and token-level masking disrupts this contextual integrity. Research on this aspect will continue in subsequent studies.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-supervised bug detection and repair. *Advances in Neural Information Processing Systems*, 34:27865–27876.

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134*.

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

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Zixuan Ke, Yijia Shao, Haowei Lin, Tatsuya Konishi, Gyuhak Kim, and Bing Liu. 2023. Continual pre-training of language models. *arXiv preprint arXiv:2302.03241*.

Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

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

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

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

662 # A Appendix

663 ## A.1 Bug SQL generation prompt of oriented generation method

Prompt

Based on the SCHEMAS and TARGET SQL, help to generate the error `sql` which are related to SCHEMAS and similar to TARGET SQL. The generated error `sql` should contain error related to ERROR INFO. You should obey the following RULES.
RULES

1. If the SCHEMAS are empty, it means the TARGET SPARK SQL is not related to any schemas.
2. ERROR INFO should not be appeared in explanation.
3. Except for error part of code, other parts of code should be same between correct `sql` and error `sql`.
4. Comments and indents in generated error `sql` and correct `sql` should be the same.
5. If it is hard to generate error `sql` which is similar to the TARGET SQL related to ERROR INFO, please return no in suitable field, otherwise it should be yes.

Below is a brief example which you can refer to (if the slots of example is empty please ignore Example section):
[EXAMPLE]
target `sql`:
TARGET_SQL_EXAMPLE_PLACEHOLDER
error info:
ERROR_INFO_EXAMPLE_PLACEHOLDER
error `sql`:
ERROR_SQL_EXAMPLE_PLACEHOLDER
Now give you the tables schema, corresponding target SQL and error type information as below. Please write a error SQL that match the error type information.
[SCHEMAS]
SCHEMAS_PLACEHOLDER
[TARGET SPARK SQL]
TARGET_SPARK_SQL_PLACEHOLDER
[ERROR INFO]
ERROR_INFO_PLACEHOLDER
RESPONSE REQUIREMENT
Return `json` str which can be parsed by `json.loads()` of python3 as following:
`{"error sql": "", "correct sql": "", "reason": "", "suitable": ""}`

664

665 ## A.2 Figures

666 **Figure 9** illustrates the distribution of the number of diff code lines in our collected training data. It can
667 be observed that over 50% of the bug SQL code require only a single line modification to be transformed
668 into correct SQL code.
669 **Figure 10** clearly demonstrates the differences in loss reduction when performing continued pre-train on
670 in-domain SQL code corpus, comparing full-parameter training and training with frozen embedding layer
671 parameters. Despite the embedding layer parameters constituting less than 2% of the total parameters in
672 DeepSeek-Coder6.7b, the loss reduction during continue pre-train with frozen embedding layer parameters
673 is highly unstable. Moreover, the final converged loss value shows a significant disparity compared to
674 full-parameter continue pre-train.

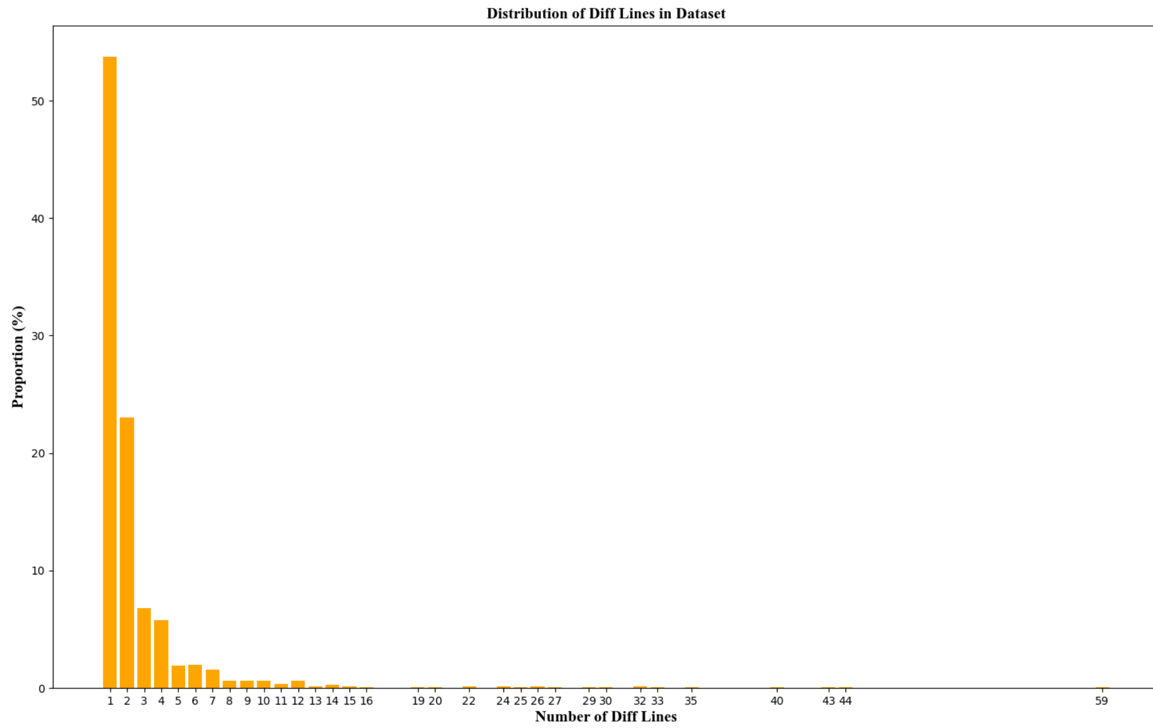

Figure 9: Distribution of diff lines proportion in SQL code

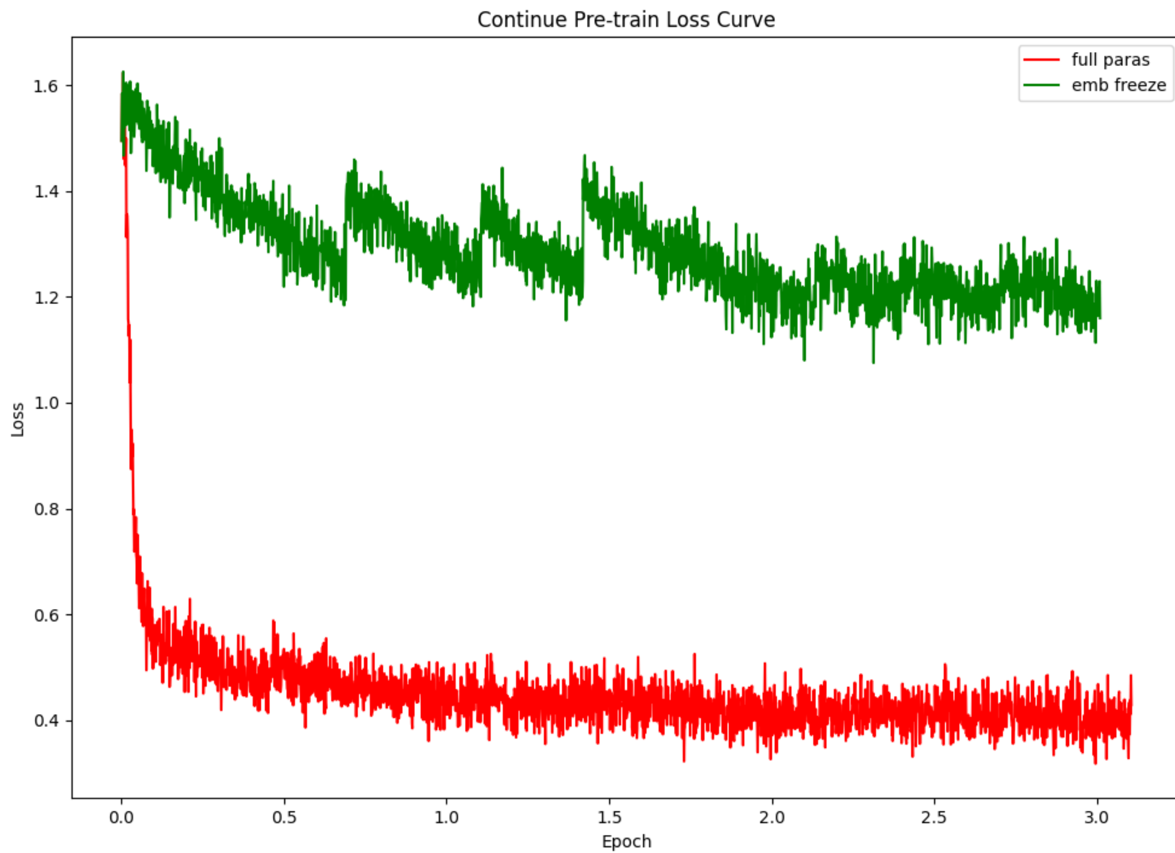

Figure 10: Training Loss Curve for Two Continue Pre-train Methods