LEVER: Learning to Verify Language-to-Code Generation with Execution

Ansong Ni [1 †] **Srini Iyer** [2] **Dragomir Radev** [1] **Ves Stoyanov** [2] **Wen-tau Yih** [2] **Sida I. Wang** [2 *] **Xi Victoria Lin** [2 *]

Abstract

The advent of large language models trained on code (code LLMs) has led to significant progress in language-to-code generation. State-of-the-art approaches in this area combine LLM decoding with sample pruning and reranking using test cases or heuristics based on the execution results. However, it is challenging to obtain test cases for many real-world language-to-code applications, and heuristics cannot well capture the semantic features of the execution results, such as data type and value range, which often indicates the correctness of the program. In this work, we propose LEVER, a simple approach to improve language-to-code generation by learning to verify the generated programs with their execution results. Specifically, we train verifiers to determine whether a program sampled from the LLMs is correct or not based on the natural language input, the program itself and its execution results. The sampled programs are reranked by combining the verification score with the LLM generation probability, and marginalizing over programs with the same execution results. On four datasets across the domains of table QA, math QA and basic Python programming, LEVER consistently improves over the base code LLMs(4.6% to 10.9% with `code-davinci-002`) and achieves new state-of-the-art results on all of them.

1. Introduction

The ability of mapping natural language to executable code is the cornerstone of a variety AI applications such as database interfaces (Pasupat & Liang, 2015; Yu et al., 2018; Shi et al., 2020), robotics control (Zhou et al., 2021; Shridhar et al., 2020) and virtual assistants (Agashe et al., 2019; Lai et al., 2022). Recent advances on large language models (LLMs) (Brown et al., 2020; Wei et al., 2021; Chowdhery et al., 2022), especially those pre-trained on code (code LLMs) (Chen et al., 2021a; Fried et al., 2022; Nijkamp et al., 2022; Li et al., 2022a), have shown great promise in such tasks with in-context few-shot learning (Shi et al., 2022; Chen et al., 2022a; Zhang et al., 2022). Yet their performance is still far from perfect (Chen et al., 2021a). Considering the computation cost to finetune such models, it is appealing to explore ways to improve them without changing their parameters.

A key observation is that while LLMs struggles with precision in the few-shot setting, it often produces the correct output when enough samples are drawn. Previous work have shown that majority voting and filtering by test cases can significantly boost their performance when samples are drawn at scale (Chen et al., 2021a; Austin et al., 2021; Li et al., 2022a). Shen et al. (2021) and Cobbe et al. (2021) further demonstrated the effectiveness of training a verifier and using the verification scores to rerank the candidate solutions for math world problems. Comparing to approaches that solely rely on execution consistency and error pruning, trained verifiers can make use of the rich semantic features in the model solutions, such as data types, value range, and variable attributes, which can be strong indicators of correctness of the programs. While Cobbe et al. (2021) and subsequent work (Li et al., 2022b; Kadavath et al., 2022) focus on verifying natural language solutions by LMs, a natural question is whether the same approach can be applied to program solutions.

In this work, we propose learning to verify (LEVER) language-to-code generation by code LLMs, with the help of execution. More specifically, we train a verifier that learns to distinguish and reject incorrect programs based on the joint representation of the natural language description, the program surface form and its execution result. We further combine the verification probability with the LLM generation probability and marginalize over programs with the same execution results. We use this aggregated probability as the reranking score and output the programs that execute to the most probable result.

We conduct extensive experiments on four different language-to-code benchmarks across domains of text-to-

[†]Majority of the work done during an internship at Meta AI. [*]Equal contribution [1]Yale University [2]Meta AI. Correspondence to: Ansong Ni <ansong.ni@yale.edu>, Xi Victoria Lin <victorialin@meta.com>, Sida I. Wang <sida@meta.com>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

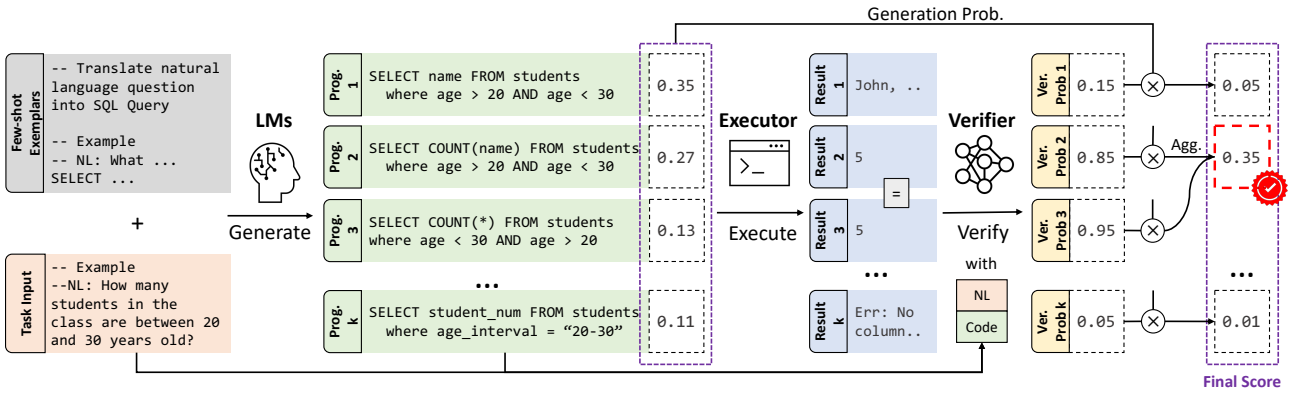


Figure 1: The illustration of LEVER using text-to-SQL as an example. It consists of three steps: 1) *Generation*: sample programs from code LLMs based on the task input and few-shot exemplars; 2) *Execution*: obtain the execution results with program executors; 3) *Verification*: using a learned verifier to output the probability of the program being correct based on the NL, program and execution results.

SQL semantic parsing, table QA, math reasoning and basic Python programming. Experiment results with three different code LLMs show that LEVER consistently improves the execution accuracy of the generated programs. Notably, LEVER coupled with `code-davinci-002` improves over strong baselines that use execution error pruning by 4.6% to 10.9%, and achieves the new state-of-the-art results on all four benchmarks, without using task-specific model architecture or prompting methods. Ablation studies show that execution results are crucial for the verification and LEVER also yields non-trivial improvements in low-resource and weakly-supervised settings.[1]

2. Approach

We now introduce the detailed formulation and training procedures of LEVER. The key components are illustrated in Figure 1.

2.1. Language-to-Code Generation with Code LLMs

The input for a language-to-code task typically consists of the natural language (NL) description and optionally some programming context (*e.g.,* data stores, assertions and so on). We denote such input as x. Given x, a generation model $P(y|x)$ generates a program y which is later executed via an executor $\mathcal{E}(\cdot)$ to obtain the result[2] $\mathcal{E}(y)$. For few-shot learning with large LMs, the generation is also often conditioned on a fixed set of m exemplars, $\{(x_i, y_i)\}_{i<m}$. Thus the few-shot language-to-code generation with code LLMs can be formulated as:

$$P_{\mathbf{LM}}(y|x) = P(y|\,\text{prompt}(x, \{(x_i, y_i)\}_{i<m})), \quad (1)$$

where $\text{prompt}(x, \{(x_i, y_i)\}_{i<m})$ is a string representation of the overall input. Greedy search is typically used to find the program with the (approximately) highest generation probability, *i.e.,* $\hat{y}_{\text{greedy}} \approx \arg\max_y P_{\mathbf{LM}}(y|x)$.

2.2. Reranking of Program Candidates

The key observation motivating our method is that a reasonably large sample set from $P_{\mathbf{LM}}(y|x)$ often includes the correct programs. This suggests that reranking of the program candidates may yield significant result improvement. The idea of discriminative reranking (Shen et al., 2004; Collins & Koo, 2005) is to learn a scoring function $R(x, \hat{y})$ that measures how likely $\hat{y}$ is the best output for input x. Given $R(\cdot)$, the reranker outputs the program with the highest reranking score among the set of candidates S:

$$\hat{y}_{\text{rerank}} = \arg\max_{\hat{y} \in S} R(x, \hat{y}) \quad (2)$$

Next we introduce how we adopt a trained verifier to verify and rerank program candidates sampled from code LLMs such that $\hat{y}_{\text{rerank}}$ is better than $\hat{y}_{\text{greedy}}$.

Program Sampling from Code LLMs. Given input x, instead of performing greedy search, we obtain k programs from $P_{\mathbf{LM}}(y|x)$ with temperature sampling, *i.e.,* $\{\hat{y}_i\}_{i=1}^{k} \sim P_{\mathbf{LM}}(y|x)$. As the same programs may be sampled more than once, we perform deduplication to form a set of n *unique* program candidates $S = \{\hat{y}_i\}_{i=1}^{n}$, where $n \leq k$. We choose to do sampling instead of beam search mainly for two reasons: 1) recent work suggests that beam search for code generation typically results in worse performance due

[1] We open-source our experiment code for reproducibility: `https://github.com/niansong1996/lever`.

[2] Some datasets such as Spider (Yu et al., 2018) require the input values to be generated together with the programs, hence y^* is directly executable. Others require the programs to be executable on separately provided test cases, *e.g.,* MBPP (Austin et al., 2021). We adopt this notation for simplicity.

to degenerated programs (Austin et al., 2021; Zhang et al., 2022); and 2) beam search is not available or efficiently implemented for all LLMs that we test on (*e.g.,* Codex).

Verification with Execution. We use a simple concatenation of the problem description x, candidate program $\hat{y}$ and a representation of its execution results $\mathcal{E}(\hat{y})$ as the input to the reranker. Inspired by recent work (Cobbe et al., 2021; Li et al., 2022b), we parameterize our discriminative reranker as a verification (*i.e.,* binary classification) model $P_\theta(v|x, \hat{y}, \mathcal{E}(\hat{y}))$, where $v \in \{0, 1\}$. In practice, the reranker can be implemented using any binary classification architecture. We report experiments using T5 (Raffel et al., 2020) and RoBERTa (Liu et al., 2019) in §B.2.

Given an input x and a candidate program $\hat{y} \in S$, we obtain the reranking probability as the joint probability of generation and passing the verification:

$$P_R(\hat{y}, v_{=1}|x) = P_{\mathbf{LM}}(\hat{y}|x) \cdot P_\theta(v_{=1}|x, \hat{y}, \mathcal{E}(\hat{y})) \quad (3)$$

Execution Result Aggregation. Since programs with the same semantics may have different surface forms, we further aggregate the reranking probability of the programs in S that executes to the same result. In this way, we relax the dependency on the surface form and focus on the execution results instead. The final scoring function for reranking is therefore:

$$R(x, \hat{y}) = P_R(\mathcal{E}(\hat{y}), v_{=1}|x) = \sum_{y \in S, \mathcal{E}(y)=\mathcal{E}(\hat{y})} P_R(y, v_{=1}|x)$$

Since there might be several programs that share the same execution result of the highest probability, we break tie randomly in this case when outputting the programs.

2.3. Learning the Verifiers

The previous sections described how to use a verifier at inference time. Next we introduce its training process.

Training Data Creation. For language-to-code datasets, each example is typically a triplet of (x, y^*, z^*), where $z^* = \mathcal{E}(y^*)$ is the gold execution result and y^* is the gold program. As annotating the programs requires domain expertise, for some datasets where the final results can be directly obtained, only z^* but no y^* is provided for learning (Artzi & Zettlemoyer, 2013; Cheng & Lapata, 2018; Goldman et al., 2018). This is known as the weakly-supervised setting. To gather training data, we obtain a set of n unique programs candidates $\hat{S} = \{\hat{y}_i\}_{i=1}^n$ for each input x in the training set, by first sampling k programs from $P_{\mathbf{LM}}(\hat{y}|x)$ and then remove all the duplicated programs, similarly as inference time. Then for each program candidate $\hat{y} \in S$, we obtain its binary verification label by comparing the execution result $\hat{z} = \mathcal{E}(\hat{y})$ with the gold[3] execution result z^*, *i.e.,* $v = \mathbb{1}(\hat{z} = z^*)$. For the datasets that contain the gold program y^*, we append $(x, y^*, z^*, v_{=1})$ as an additional verification training example, and we skip this step for the weakly-supervised datasets. This way, we create a set of verification training examples $\{(x, \hat{y}_i, \hat{z}_i, v_i) \mid \hat{y}_i \in S\}$ for each input x.

	Spider	**WikiTQ**	**GSM8k**	**MBPP**
Domain	Table QA	Table QA	Math QA	Basic Coding
Has program	✓	✓*	✗	✓
Target	SQL	SQL	Python	Python
Data Statistics				
# Train	7,000	11,321	5,968	378
# Dev	1,032	2,831	1,448	90
# Test	-	4,336	1,312	500
Few-shot Generation Settings				
Input Format	Schema + NL	Schema + NL	NL	Assertion + NL
# Shots	8‡	8	8	3
# Samples (train / test)	20/50†	50/50	50/100	100/100
Generation Length	128	128	256	256

Table 1: Summary of the datasets used in this work. *: About 80% examples in WikiTableQuestions are annotated with SQL by Shi et al. (2020). †: 50/100 for InCoder and CodeGen for improving the upper-bound. ‡: Only the first 2 of the 8 exemplars are used for InCoder and CodeGen due to limits of context length and hardware.

Learning Objective. Given this set of verification training examples, we formulate the loss for input x with the negative log-likelihood function, normalized by the number of program candidates

$$\mathcal{L}_\theta(x, S) = -\frac{1}{|S|} \cdot \sum_{\hat{y}_i \in S} \log P_\theta(v_i|x, \hat{y}_i, \hat{z}_i) \quad (4)$$

The normalization step is important to prevent an example with a large number of unique program candidates to dominate learning.

3. Experimental Setup

3.1. Datasets

We conduct experiments on four language-to-code datasets across domains of semantic parsing, table QA, math reasoning and basic python programming. The main settings

[3]For datasets that provide multiple test cases, we label a program as correct if and only if its execution results match the ground truth program on all test cases.

of these four datasets are shown in Table 1. More detailed settings for verification are in Table 7 of the Appendix.

▷ **Spider** (Yu et al., 2018) is a semantic parsing dataset on generating SQL queries from natural language questions. With 7k parallel training data, it is also ideal for finetuning generators;
▷ **WikiTableQuestions (WikiTQ)** (Pasupat & Liang, 2015) is a table question answering dataset, for which we attempt to solve by generating and executing SQL queries over the source tables. We use the preprocessed tables from Shi et al. (2020) and adopt their annotated SQL queries for adding gold programs for the originally weakly-supervised dataset;
▷ **GSM8k** (Cobbe et al., 2021) is a benchmark for solving grade-school level math word problems. Following previous work (Chowdhery et al., 2022; Chen et al., 2022b; Gao et al., 2022), we approach this benchmark by generating Python programs from questions in NL, which should produce the correct answer upon execution. The original dataset only has natural language and not program solutions, thus it is weakly-supervised for language-to-code;
▷ **MBPP** (Austin et al., 2021) contains basic Python programming programs stated in natural language. Each example is equipped with 3 test cases to check the correctness of the programs. Following previous work (Shi et al., 2022; Zhang et al., 2022), we use the first test case as part of the prompt for the model to generate correct function signatures and use all three of them for evaluating correctness.

3.2. Code LLMs

We evaluate LEVER with three different code LLMs:
▷ **Codex** (Chen et al., 2021a) is a family of code LLMs of different sizes developed by OpenAI. Specifically, we use the `code-davinci-002` API[4] through its official Python bindings.
▷ **InCoder** (Fried et al., 2022) is a family of code LLMs up to 6B parameters trained on a large corpus of code with permissively licenses. We experiment with InCoder-6B and use it for left-to-right generation.
▷ **CodeGen** (Nijkamp et al., 2022) is a family of code LLMs and we evaluate the CodeGen-16B-multi version. Although SQL files are not included in the training corpus for CodeGen, we found it to still perform reasonably well on SQL generation tasks possibly because the SQL queries were mixed in with source files of other programming languages.

3.3. Baselines and Evaluation Metric

Baselines. We compare LEVER to the following baseline approaches for generating programs using code LLMs.
▷ **Greedy**: Select the most likely token per decoding step.
▷ **Maximum Likelihood (ML)**: From k sampled program candidates, select the program with the highest generation log-probability, *i.e.,* $\log P_{\mathbf{LM}}(\hat{y}|x)$ (or normalized generation log-probability as $\log P_{\mathbf{LM}}(\hat{y}|x)/|\hat{y}|$). We determine empirically using the development set whether to use the normalized probability for each dataset. More details can be found in Appendix A.
▷ **Error Pruning + ML (EP + ML)**: Prune out the candidate programs with execution errors; then select the program with the maximum likelihood;
▷ **Error Pruning + Voting (EP + Voting)**: Take the majority vote on the execution results among the error-free programs, and select the most-voted execution result and its corresponding programs.

We focus on comparing with the EP+ML baseline, as it is a simple reranking method that exploits execution and yields competitive results consistently across different datasets and code LLMs.

Evaluation metric. Following previous work (Xie et al., 2022; Liu et al., 2021; Ni et al., 2022; Zhang et al., 2022), we use *execution accuracy* as the main evaluation metric for all datasets, which measures the percentage of examples that yields the gold execution result or pass all test cases.

3.4. Implementation Details

Verifier training. We create the verification training data by sampling from the LLMs on the training set, using the sampling budget described in Table 1. More statistics of the resulting training data can be found in Table 7 in the Appendix. When learning the verifiers, as shown in Eq. 4, the training loss is computed by averaging over all the program samples for each example. As we batch the program samples for the same examples together, the effective batch size will also be multiplied by the sample size. This could be problematic when sample size gets large (up to 100 in our experiments) as they may not be able to fit into the GPU memory at once. Therefore, we down-sample the programs used for learning per example in each iteration. The random down-sampling happens at the beginning of every epoch of training so the verifiers are able to see different programs each epoch. Detailed batch sizes and downsampling factor can be found in Table 7 in the Appendix.

Execution result representation. The input to the verifier is a concatenation of the task input, the candidate program and its execution results. For Spider and WikiTQ, we use the linearized resulting tables from SQL execution as the execution results. For GSM8k, we use the value of the variable named "`answer`" after executing the program as the execution results. For MBPP, we use the type and value (casted to string) returned by the functions. All execution errors are represented as "`ERROR: [reason]`", such as "`ERROR: Time out`". Examples of these verifier inputs

[4] `https://openai.com/api/`

Methods	Dev
Previous Work without Finetuning	
Rajkumar et al. (2022)	67.0
MBR-Exec (Shi et al., 2022)	75.2
Coder-Reviewer (Zhang et al., 2022)	74.5
Previous Work with Finetuning	
T5-3B (Xie et al., 2022)	71.8
PICARD (Scholak et al., 2021)	75.5
RASAT (Qi et al., 2022)	80.5
This Work with code-davinci-002	
Greedy	75.3
EP + ML	77.3
LEVER	**81.9**$_{\pm 0.1}$

Table 2: Execution accuracy on the Spider dataset. Standard deviation is calculated over three runs with different random seeds (same for the following tables when std is presented).

Methods	Dev	Test
Previous Work without Finetuning		
Codex QA* (Cheng et al., 2022)	50.5	48.7
Codex SQL (Cheng et al., 2022)	60.2	61.1
Codex Binder (Cheng et al., 2022)	**65.0**	64.6
Previous Work with Finetuning		
TaPEX* (Liu et al., 2021)	60.4	59.1
TaCube (Zhou et al., 2022)	61.1	61.3
OmniTab* (Jiang et al., 2022)	-	63.3
This Work with code-davinci-002		
Greedy	49.6	53.0
EP + ML	52.7	54.9
LEVER	64.6$_{\pm 0.2}$	**65.8**$_{\pm 0.2}$

Table 3: Execution accuracy on the WikiTQ dataset. *: modeled as end-to-end QA without generating programs as a medium.

for different datasets can be found in Table 11.

Verifier model selection. We use the development set to choose the best verifier model. We select T5-base for Spider, T5-large for WikiTQ and MBPP, and RoBERTa-large for GSM8k as the base LM for the verifiers to use in the main experiments[5]. The selection process is detailed in § B.2. For the T5 models (Raffel et al., 2020), we train them to output the token "yes/no" for each positive/negative example given the verifier input, and we take the probability of generating "yes" as the verification probability during inference. For RoBERTa (Liu et al., 2019), we add a linear layer on top of the `[CLS]` head, following the standard practice of sequence classification with encoder-only models (Devlin et al., 2019).

The details of LLM sampling, few-shot prompt construction and dataset-specific setups can be found in Appendix A.

4. Main Results

We show the performance of LEVER coupled with Codex-Davinci and compare it with the state-of-the-art finetuning and few-shot performances from previous work for Spider (Table 2), WikiTQ (Table 3), GSM8k (Table 4) and MBPP (Table 5). In addition, we also evaluate LEVER with InCoder and CodeGen models on Spider and GSM8k (Table 6).

4.1. Effectiveness of LEVER.

LEVER consistently improves the performance of all code LLMs on all tasks, yielding improvements of 6.6% (Spider) to 17.3% (WikiTQ) over the greedy decoding baselines for Codex-Davinci. For weaker models such as InCoder and CodeGen, we observe improvements up to 30.0% for Spider and 15.0% for GSM8k. Moreover, LEVER combined with Codex-Davinci also achieves new state-of-the-art results on all four datasets, with improvements ranging from 1.2% (WikiTQ) to 2.0% (MBPP). On the challenging text-to-SQL dataset, Spider, where the previous state-of-the-art is achieved by finetuning a T5-3B model augmented with relational-aware self-attention, we achieved even better results with Codex-Davinci + LEVER, where the verifier is finetuned using a T5-base model. LEVER also improves the previous best results on Spider using InCoder and CodeGen, by 13.2% and 20.6%, respectively.

As LEVER is a simple method that combines few-shot LM generation with learned verifiers, it can potentially benefit more advanced prompting methods (Li et al., 2022b; Cheng et al., 2022) or model architectures (Qi et al., 2022; Wang et al., 2020), which we leave as future work.

Methods	Dev	Test
Previous Work without Finetuning		
PAL (Gao et al., 2022)	-	72.0
Codex + SC† (Wang et al., 2022)	-	78.0
PoT-SC (Chen et al., 2022b)	-	80.0
Previous Work with Finetuning		
Neo-2.7B + SS (Ni et al., 2022)	20.7	19.5
Neo-1.3B + SC (Welleck et al., 2022)	-	24.2
DiVeRSe*† (Li et al., 2022b)	-	83.2
This Work with codex-davinci-002		
Greedy	68.1	67.2
EP + ML	72.1	72.6
LEVER	**84.1**$_{\pm 0.2}$	**84.5**$_{\pm 0.3}$

Table 4: Execution accuracy on the GSM8k dataset. *: finetuned model combined with Codex (similar to LEVER); †: generating natural language solutions instead of programs.

[5]We attempted using the code LLM itself as the verifier in a few-shot manner, but the performance is inferior than EP+ML.

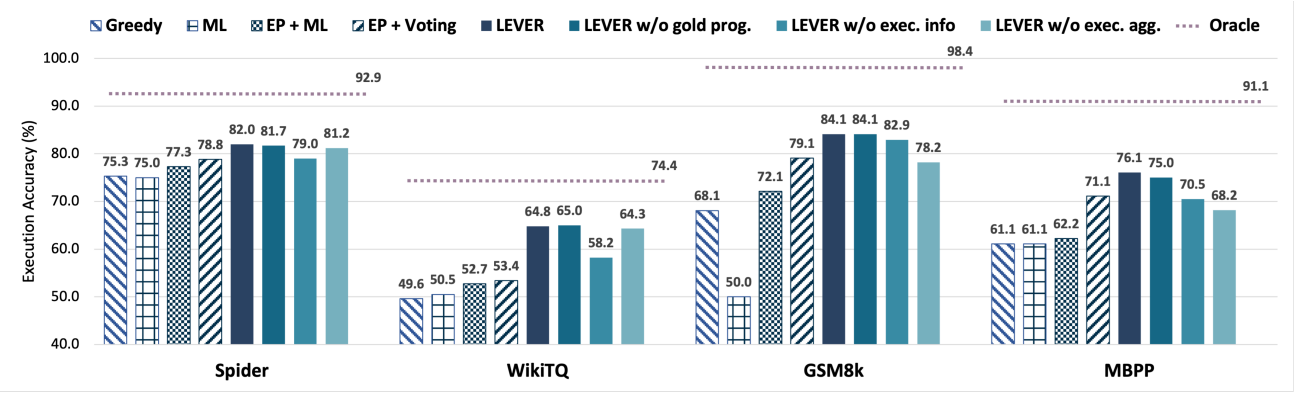


Figure 2: Comparison of LEVER and baselines with Codex-Davinci. LEVER and its ablation results are in solid bars.

Methods	Dev	Test
Previous Work without Finetuning		
MBR-Exec (Shi et al., 2022)	-	63.0
Reviewer (Zhang et al., 2022)	-	66.9
This Work with codex-davinci-002		
Greedy	61.1	62.0
EP + ML	62.2	60.2
LEVER	**75.4**$_{\pm 0.7}$	**68.9**$_{\pm 0.4}$

Table 5: Execution accuracy on the MBPP dataset.

Methods	InCoder-6B		CodeGen-16B	
	Spider	GSM8k	Spider	GSM8k
Previous work:				
MBR-EXEC	38.2	-	30.6	-
Reviewer	41.5	-	31.7	-
Baselines:				
Greedy	24.1	3.1	24.6	7.1
ML	33.7	3.8	31.2	9.6
EP + ML	41.2	4.4	37.7	11.4
EP + Voting	37.4	5.9	37.1	14.2
LEVER	54.1	**11.9**	51.0	**22.1**
– gold prog.	53.4	-	**52.3**	-
– exec. info	48.5	5.6	43.0	13.4
– exec. agg.	**54.7**	10.6	51.6	18.3
Oracle	71.6	48.0	68.6	61.4

Table 6: Results with InCoder and CodeGen as the Code LLMs, evaluated on the dev set with T5-base as the verifier. Previous work results were copied from Zhang et al. (2022).

4.2. Ablations with LEVER

We perform ablation study for LEVER with Codex-Davinci and compare with the baselines mentioned in § 3.3, and the results are shown in Figure 2. The same ablations are conducted for InCoder and CodeGen with results in Table 6. In these results, we include an "**Oracle**" performance which is obtained by always selecting the correct program as long as they appear in the sample set.

Effect of including execution results. According to Figure 2, the performance drops considerably on all four benchmarks when execution result is removed from the verifier input, indicating that the execution outcome is important for verifier training. The effect varies across different datasets. While it causes an absolute performance drop of 6.6% and 5.6% for WikiTQ and MBPP, as the drop is smaller for Spider (3.0%) and GSM8k (1.2%). We found the code samples for WikiTQ and MBPP contain more execution errors, which explains why our approach is more effective on these two datasets. Table 6 shows similar trends for InCoder-6B and CodeGen-16B on Spider and GSM8k. The smaller LMs have worse few-shot performance and removing the execution information from the verifier often results in even greater performance drops. Moreover, we found that LEVER in general outperforms the EP+ML baseline, indicating that the verifiers can make use of clues beyond simple execution errors. More detailed quantitative analysis of when execution information helps is in Figure 6.

Effect of execution result aggregation. Aggregating the programs with the same execution result is a simple and widely used technique (Chen et al., 2022b; Cheng et al., 2022). We find execution aggregation work well with LEVER on datasets with Python output, but only marginally benefit the SQL datasets. A probable reason is that the Python code structure is more flexible than that of the domain-specific languages as SQL. In the database querying domain, it is more likely for an incorrect program to execute to some trivial but wrong results (*e.g.,* "0" or empty table). After aggregation, such incorrect results may accumulate enough probability mass to out-weight the correct one, leading to negative impact on the performance.

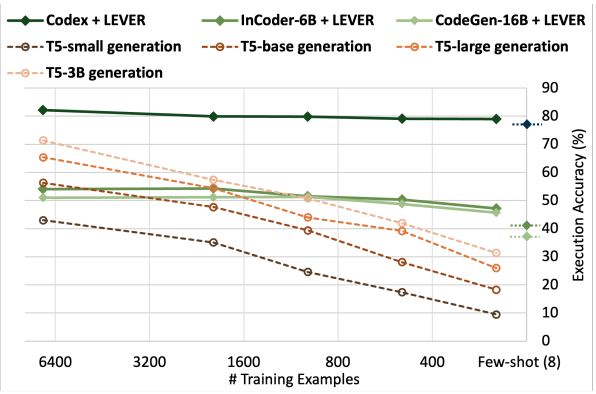
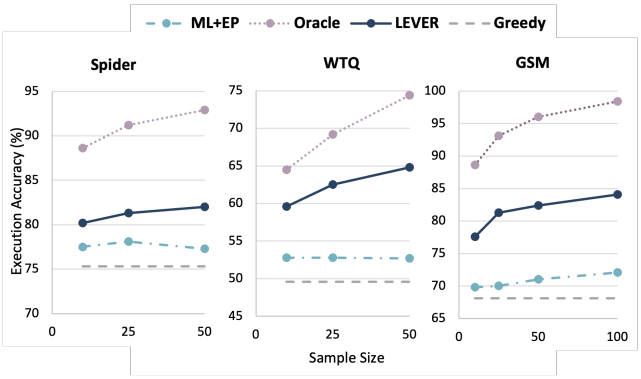
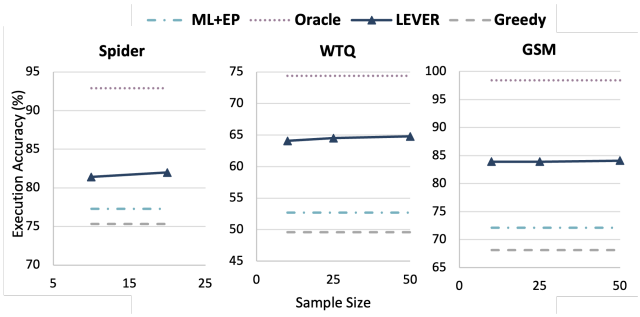


Figure 3: Verification vs. generation performance when decreasing the number of training examples for Spider. Data markers on the y-axis denote the EP+ML baseline, and the x-axis is on the logarithmic scale. T5-base is used as the base model for LEVER. WikiTQ and GSM8k results can be found in Figure 7 in the Appendix.

Weakly-supervised settings. We also compare the performance of LEVER under fully- and weakly-supervised settings. Figure 2 and Table 6 show that the performance of LEVER is largely preserved when the gold programs are not given and the weakly-supervised setting is used (§2.3), with an absolute performance drop up to 1.1%. This suggests that LEVER works well under the weakly-supervised settings, and the program itself is less informative for verification comparing to the execution results.



(a) Ablation on sample size at inference time for LEVER, while sample size at training time is fixed as in Table 1.



(b) Performance with different number of programs to sample per example for training the verifiers. Sample size at inference time is fixed as in Table 1.

Figure 4: How sample size during training and inference time affects the performance, with LEVER + Codex-Davinci.

5. Analysis

5.1. Training Example Scaling

We show how the performance of LEVER changes with fewer training examples in Figure 3, using Spider as an example. More results on WikiTQ and GSM8k are in § B.3. The improvements with LEVER over base LLMs are still consistent even when only 250 examples are given, with improvements ranging from 1.7% to 10.0% over different datasets and LLMs. This suggests that LEVER can work under few-resource settings. Moreover, the trend also varies for different datasets and code LLMs, for example, when using Codex as the LLM, the performance of LEVER drops by 6.4% for WikiTQ and only 3.2% for Spider. However, also on Spider, the performance is lowered by 6.9% and 5.3% for InCoder and CodeGen. This suggests that having more training examples for LEVER has larger effect for harder datasets and weaker LMs.

With Figure 3, we also compare the performance of LEVER with the T5 models being directly finetuned for generation given the same number of training examples. While verification can be learned with only hundreds of examples, the performance of finetuned T5 models drastically drops when less training examples are available. As an example, for 500 examples, a T5-base verifier on InCoder/CodeGen outperforms a finetuned T5-3B generator by $\sim 7\%$.

5.2. Sample Size Scaling

Since drawing samples from LLMs in may be costly computational-wise, here we study the how sample size during training and inference time affects the performance. As we can see from Figure 4a, during inference time, when lowering the sample size from 50 to 10 programs per example, the performance of LEVER drops by 1.8% (Spider) to 5.2% (WikiTQ). This indicates that the LEVER is sensitive to the sample size at inference time, which is expected as it also greatly affects oracle results (*i.e.,* the upper-bound for reranking). In comparison, Figure 4b shows that LEVER is highly insensitive to the sample size for providing training data, with the performance gap all below 1% for the three datasets. Overall, the results show that a higher sampling budget helps more at test time.

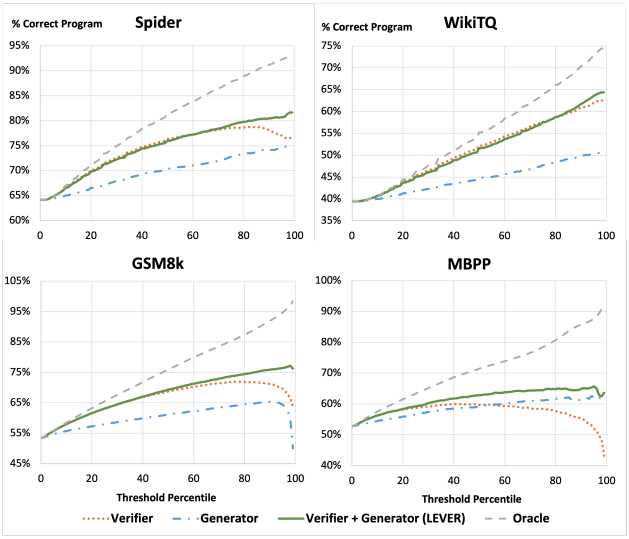
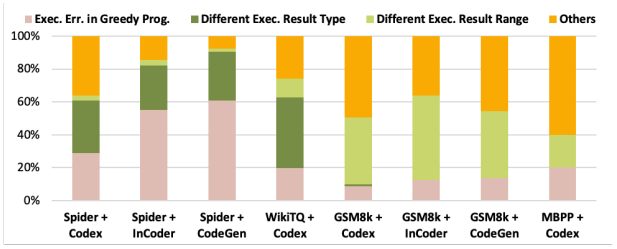


Figure 5: Calibration of the verifier, generator (Codex-Davinci), and their combined probability (used by LEVER). The sampled programs are first ranked by the model probabilities. The x-axis represents the percentage of samples excluded after thresholding, and the y-axis represents the percentage of correct programs in the remaining samples. Execution aggregation is not applied in this group of plots to ensure the scoring of different programs are independent.

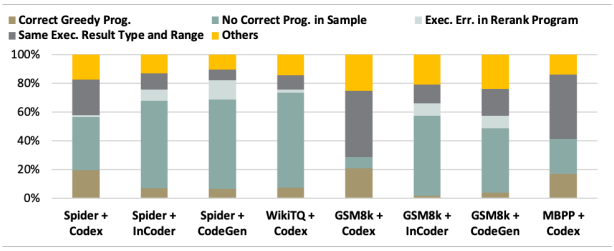
5.3. Verifier and Generator Calibration

We study how well-calibrated are the verifier and generator in identifying correct programs. Ideally, correct program samples shall be given higher probabilities thus we should observe higher percentage of programs being correct when it is closer to the top. To this end, we sort the prediction scores of the verifier, the generator and LEVER (as in Eq. 3), and move the percentile threshold and measuring the percentage of correct programs in the top ranked programs. According to Figure 5, the verifiers are generally better calibrated than the generators, especially when the threshold is in the lower percentiles. This indicates that it is easier for the verifiers to identify obvious mistakes in the programs with execution results as part of their input. Interestingly, when distinguishing between the top-ranked programs, the verifiers are poorly calibrated in three of the four tested datasets[6]. However, the generators are generally better calibrated in this region, and combining the probability of the verifier and the generator yields the best results on all four benchmarks. More specifically, on the GSM8k dataset, where the calibration of both models are quite poor for top-ranking programs, their joint probability is surprisingly well-calibrated, showing that the two models complement each other on this dataset.

[6]Our hypothesis is that the programs ranked at the top have very similar form and execution results (*e.g.,* same type and range), making it hard for a small, though finetuned, model to discriminate.



(a) When LEVER reranks a correct program at the top but the greedy decoding fails.



(b) When LEVER fails to rank a correct program at the top.

Figure 6: Quantitative analysis on when LEVER succeeds and fails to improve code LLMs over greedy decoding.

5.4. Quantitative Analysis

We present a quantitative analysis on why LEVER successfully or failed to improve the performance of LLMs. According to Figure 6, when LEVER reranks a program to replace another with higher generation probability, it is oftentimes because the execution results provide crucial information such as execution errors, variable type and range. This is consistent with our findings in § 4.2 about the importance of execution results for LEVER. It is also worth noticing that there are cases when LEVER is still able to rerank the correct program when the error-free execution results are of the same type and range with the greedy program, *i.e.,* in "others" category. Our hypothesis is that this is when the program itself becomes the main feature for the verifiers to exploit. In addition, when LEVER fails to rank correct programs to the top, the most common reason is that no correct program can be found in the samples (*i.e.,* upper-bound is reached), which is especially the case for weaker LMs. The second most common reason for LEVER to fail is that the execution results of the incorrect program upon reranking has the same type and range as the correct program in the samples. In this case, execution results do not provide rich information for the verifiers thus LEVER fails to improve code LLMs.

6. Related Work

Language-to-Code Generation. Translating natural language to code is a long-standing challenge through all

eras of artificial intelligence, including rule-based systems (Woods, 1973; Templeton & Burger, 1983), structured prediction (Zelle & Mooney, 1996; Zettlemoyer & Collins, 2005; Gulwani & Marron, 2014) and deep learning (Xiao et al., 2016; Dong & Lapata, 2016; Rabinovich et al., 2017; Zhong et al., 2017; Lin et al., 2017). Recently, pre-trained code language models (Chen et al., 2021a; Wang et al., 2021; Fried et al., 2022; Nijkamp et al., 2022; OpenAI, 2022) have demonstrated surprisingly strong performance in this problem across programming languages (Lin et al., 2018; Yu et al., 2018; Austin et al., 2021; Cobbe et al., 2021; Li et al., 2022a). A number of approaches were proposed to refine LLM sample selection, including test case execution (Li et al., 2022a), cross-sample similarity (Chen et al., 2021a; Li et al., 2022a; Shi et al., 2022) and maximum mutual information (Zhang et al., 2022) based filtering. Our work proposes a learnable verification module to judge the sample output of LLMs to further improve their performance.

Code Generation with Execution. Previous code generation work have exploited execution results in different ways. Weakly-supervised learning approaches (Berant et al., 2013; Pasupat & Liang, 2015; Guu et al., 2017) model programs as latent variables and use execution results to derive the supervision signal. Intermediate execution results were used to guide program search at both training (Chen et al., 2019; 2021b) and inference time (Wang et al., 2018). When sampling at scale, majority voting based on the execution results has been shown effective for candidate selection (Li et al., 2022a; Cobbe et al., 2021). Shi et al. (2022) generalizes this principle by selecting samples that have the maximum concensus with other samples in the execution results. We propose to train a verification model to judge the correctness of code generation taking the execution results into account.

Learning to Verify. Previous work have shown the effectiveness of learned verifiers for sample filtering in domains such as math QA (Shen et al., 2021; Cobbe et al., 2021) and commonsense QA (Li et al., 2022b), where the solution is mostly described in natural language. While it is more common to train the verifiers independently from the generator (Cobbe et al., 2021; Li et al., 2022b), Shen et al. (2021) jointly fine-tuned both at the same time. Previous work have also used different base LMs for the verifiers. Cobbe et al. (2021) uses GPT-3 (Brown et al., 2020) while Li et al. (2022b) uses DeBERTa (He et al., 2020). Besides task-specific verifiers, Kadavath et al. (2022) shows that large LMs can self-verify their output in a few-shot setting for a wide range of tasks. Chen et al. (2022a) and other works (Tufano et al., 2020; Li et al., 2022a) use LMs to generate test cases instead of directly judging the correctness of the output programs. In comparison, the setting of LEVER is closer to Li et al. (2022b) as we train the verifier separately and use a much smaller LM for it (approximately 0.5% of the generator parameter size). We report the first set of comprehensive evaluation on language-to-code tasks, making use of the program execution results[7].

Discriminative Reranking. Discriminative reranking approaches have long been used to further improve the performance of sequence generation tasks, including summarization (Wan et al., 2015), machine translation (Shen et al., 2004; Lee et al., 2021), dialogue response generation (Olabiyi et al., 2018) and more rencently, code generation (Yin & Neubig, 2019). LEVER can be viewed as a discriminative reranking framework.

7. Limitations

In this work, we use execution information to verify the programs in LEVER. However, the execution of the programs depends on at least one set of inputs (*e.g.,* arguments for a function) and adequate execution context (*e.g.,* databases), which may not be provided for certain applications. Moreover, we can not always assume that model-generated programs are safe to execute. In addition, we PASS@1 as the main evaluation metric in the experiments. While it is ideal for applications such as text-to-SQL and math reasoning where the users are only looking for answers to their questions, metrics as PASS@k or N@k could provide different perspectives for general programming tasks as MBPP.

8. Conclusion

We propose LEVER, a simple approach for improving code LLMs on language-to-code tasks, by learning separate verification models to judge the correctness of the generated programs, taking their execution results into consideration. We show that it is possible to train verifiers approximately 0.5% the size of the generators using supervised benchmark datasets. Instead of directly perform rejection sampling based on the verifier output, we show it is better to mix the generation and verfication probabilities for sample reranking. LEVER consistently improves the performance of code LLMs on four language-to-code tasks, and achieves new state-of-the-art results on all of them. Further analysis suggest that the program execution results are crucial for verification and the proposed approach is generalizable across different LLMs.

Acknowledgements

The authors would like to thank Xi Ye, Tianyi Zhang, Mengzhou Xia, Luke Zettlemoyer, and the anonymous reviewers for the useful discussion and comments.

[7]While Kadavath et al. (2022) also reported self-verification results on HumanEval, their approach does not leverage execution.

References

Agashe, R., Iyer, S., and Zettlemoyer, L. JuICe: A large scale distantly supervised dataset for open domain context-based code generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 5436–5446, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1546. URL https://aclanthology.org/D19-1546.

Artzi, Y. and Zettlemoyer, L. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 1:49–62, 2013.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Berant, J., Chou, A., Frostig, R., and Liang, P. Semantic parsing on Freebase from question-answer pairs. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2013.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.-G., and Chen, W. Codet: Code generation with generated tests, 2022a. URL https://arxiv.org/abs/2207.10397.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021a.

Chen, W., Ma, X., Wang, X., and Cohen, W. W. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022b.

Chen, X., Liu, C., and Song, D. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=H1gfOiAqYm.

Chen, X., Song, D., and Tian, Y. Latent execution for neural program synthesis beyond domain-specific languages. In Ranzato, M., Beygelzimer, A., Dauphin, Y. N., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pp. 22196–22208, 2021b.

Cheng, J. and Lapata, M. Weakly-supervised neural semantic parsing with a generative ranker. In *Proceedings of the 22nd Conference on Computational Natural Language Learning*, pp. 356–367, 2018.

Cheng, Z., Xie, T., Shi, P., Li, C., Nadkarni, R., Hu, Y., Xiong, C., Radev, D., Ostendorf, M., Zettlemoyer, L., et al. Binding language models in symbolic languages. *arXiv preprint arXiv:2210.02875*, 2022.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

Cobbe, K., Kosaraju, V., Bavarian, M., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Collins, M. and Koo, T. Discriminative reranking for natural language parsing. *Computational Linguistics*, 31(1):25–70, 2005.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. BERT: pre-training of deep bidirectional transformers for language understanding. In Burstein, J., Doran, C., and Solorio, T. (eds.), *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pp. 4171–4186. Association for Computational Linguistics, 2019. doi: 10.18653/v1/n19-1423. URL https://doi.org/10.18653/v1/n19-1423.

Dong, L. and Lapata, M. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016. doi: 10.18653/v1/p16-1004. URL https://doi.org/10.18653/v1/p16-1004.

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.-t., Zettlemoyer, L., and Lewis, M. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., and Neubig, G. Pal: Program-aided language models. *arXiv preprint arXiv:2211.10435*, 2022.

Goldman, O., Latcinnik, V., Nave, E., Globerson, A., and Berant, J. Weakly supervised semantic parsing with abstract examples. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1809–1819, 2018.

Gulwani, S. and Marron, M. Nlyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In Dyreson, C. E., Li, F., and Özsu, M. T. (eds.), *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pp. 803–814. ACM, 2014. doi: 10.1145/2588555.2612177. URL https://doi.org/10.1145/2588555.2612177.

Guu, K., Pasupat, P., Liu, E. Z., and Liang, P. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. In *Association for Computational Linguistics (ACL)*, 2017.

He, P., Liu, X., Gao, J., and Chen, W. Deberta: Decoding-enhanced bert with disentangled attention, 2020. URL https://arxiv.org/abs/2006.03654.

Jiang, Z., Mao, Y., He, P., Neubig, G., and Chen, W. Omnitab: Pretraining with natural and synthetic data for few-shot table-based question answering. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 932–942, 2022.

Kadavath, S., Conerly, T., Askell, A., Henighan, T., Drain, D., Perez, E., Schiefer, N., Hatfield-Dodds, Z., DasSarma, N., Tran-Johnson, E., Johnston, S., El-Showk, S., Jones, A., Elhage, N., Hume, T., Chen, A., Bai, Y., Bowman, S., Fort, S., Ganguli, D., Hernandez, D., Jacobson, J., Kernion, J., Kravec, S., Lovitt, L., Ndousse, K., Olsson, C., Ringer, S., Amodei, D., Brown, T., Clark, J., Joseph, N., Mann, B., McCandlish, S., Olah, C., and Kaplan, J. Language models (mostly) know what they know, 2022. URL https://arxiv.org/abs/2207.05221.

Lai, Y., Li, C., Wang, Y., Zhang, T., Zhong, R., Zettlemoyer, L., Yih, S. W.-t., Fried, D., Wang, S., and Yu, T. Ds-1000: A natural and reliable benchmark for data science code generation. *arXiv preprint arXiv:2211.11501*, 2022.

Lee, A., Auli, M., and Ranzato, M. Discriminative reranking for neural machine translation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 7250–7264, 2021.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022a.

Li, Y., Lin, Z., Zhang, S., Fu, Q., Chen, B., Lou, J.-G., and Chen, W. On the advance of making language models better reasoners. *arXiv preprint arXiv:2206.02336*, 2022b.

Lin, X. V., Wang, C., Pang, D., Vu, K., Zettlemoyer, L., and Ernst, M. D. Program synthesis from natural language using recurrent neural networks. Technical Report UW-CSE-17-03-01, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, March 2017.

Lin, X. V., Wang, C., Zettlemoyer, L., and Ernst, M. D. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation LREC 2018, Miyazaki (Japan), 7-12 May, 2018.*, 2018.

Liu, Q., Chen, B., Guo, J., Ziyadi, M., Lin, Z., Chen, W., and Lou, J.-G. Tapex: Table pre-training via learning a neural sql executor. In *International Conference on Learning Representations*, 2021.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019. URL http://arxiv.org/abs/1907.11692.

Ni, A., Inala, J. P., Wang, C., Polozov, O., Meek, C., Radev, D., and Gao, J. Learning from self-sampled correct and partially-correct programs. *arXiv preprint arXiv:2205.14318*, 2022.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

Olabiyi, O., Salimov, A., Khazane, A., and Mueller, E. T. Multi-turn dialogue response generation in an adversarial learning framework. *CoRR*, abs/1805.11752, 2018. URL http://arxiv.org/abs/1805.11752.

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

Pasupat, P. and Liang, P. Compositional semantic parsing on semi-structured tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 1470–1480, Beijing, China, July 2015. Association for Computational Linguistics. doi: 10.3115/v1/P15-1142. URL https://aclanthology.org/P15-1142.

Qi, J., Tang, J., He, Z., Wan, X., Zhou, C., Wang, X., Zhang, Q., and Lin, Z. Rasat: Integrating relational structures into pretrained seq2seq model for text-to-sql. *arXiv preprint arXiv:2205.06983*, 2022.

Rabinovich, M., Stern, M., and Klein, D. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1139–1149, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1105. URL https://aclanthology.org/P17-1105.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020. URL http://jmlr.org/papers/v21/20-074.html.

Rajkumar, N., Li, R., and Bahdanau, D. Evaluating the text-to-sql capabilities of large language models. *arXiv preprint arXiv:2204.00498*, 2022.

Scholak, T., Schucher, N., and Bahdanau, D. Picard: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 9895–9901, 2021.

Shen, J., Yin, Y., Li, L., Shang, L., Jiang, X., Zhang, M., and Liu, Q. Generate & rank: A multi-task framework for math word problems. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pp. 2269–2279, Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-emnlp.195. URL https://aclanthology.org/2021.findings-emnlp.195.

Shen, L., Sarkar, A., and Och, F. J. Discriminative reranking for machine translation. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics: HLT-NAACL 2004*, pp. 177–184, 2004.

Shi, F., Fried, D., Ghazvininejad, M., Zettlemoyer, L., and Wang, S. I. Natural language to code translation with execution. *arXiv preprint arXiv:2204.11454*, 2022.

Shi, T., Zhao, C., Boyd-Graber, J., Daumé III, H., and Lee, L. On the potential of lexico-logical alignments for semantic parsing to SQL queries. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1849–1864, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.167. URL https://aclanthology.org/2020.findings-emnlp.167.

Shridhar, M., Thomason, J., Gordon, D., Bisk, Y., Han, W., Mottaghi, R., Zettlemoyer, L., and Fox, D. Alfred: A benchmark for interpreting grounded instructions for everyday tasks. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 10737–10746. IEEE, 2020.

Templeton, M. and Burger, J. D. Problems in natural-language interface to DSMS with examples from EUFID. In *1st Applied Natural Language Processing Conference, ANLP 1983, Miramar-Sheraton Hotel, Santa Monica, California, USA, February 1-3, 1983*, pp. 3–16. ACL, 1983. doi: 10.3115/974194.974197. URL https://aclanthology.org/A83-1002/.

Tufano, M., Drain, D., Svyatkovskiy, A., Deng, S. K., and Sundaresan, N. Unit test case generation with transformers. *CoRR*, abs/2009.05617, 2020. URL https://arxiv.org/abs/2009.05617.

Wan, X., Cao, Z., Wei, F., Li, S., and Zhou, M. Multi-document summarization via discriminative summary reranking. *CoRR*, abs/1507.02062, 2015. URL http://arxiv.org/abs/1507.02062.

Wang, B., Shin, R., Liu, X., Polozov, O., and Richardson, M. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 7567–7578, 2020.

Wang, C., Tatwawadi, K., Brockschmidt, M., Huang, P.-S., Mao, Y., Polozov, O., and Singh, R. Robust text-to-sql generation with execution-guided decoding, 2018. URL https://arxiv.org/abs/1807.03100.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., and Zhou, D. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

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

Wei, J., Bosma, M., Zhao, V., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M., and Le, Q. V. Finetuned language models are zero-shot learners. In *International Conference on Learning Representations*, 2021.

Welleck, S., Lu, X., West, P., Brahman, F., Shen, T., Khashabi, D., and Choi, Y. Generating sequences by learning to self-correct. *arXiv preprint arXiv:2211.00053*, 2022.

Woods, W. A. Progress in natural language understanding: an application to lunar geology. In *American Federation of Information Processing Societies: 1973 National Computer Conference, 4-8 June 1973, New York, NY, USA*, volume 42 of *AFIPS Conference Proceedings*, pp. 441–450. AFIPS Press/ACM, 1973. doi: 10.1145/1499586.1499695. URL https://doi.org/10.1145/1499586.1499695.

Xiao, C., Dymetman, M., and Gardent, C. Sequence-based structured prediction for semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1341–1350, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1127. URL https://aclanthology.org/P16-1127.

Xie, T., Wu, C. H., Shi, P., Zhong, R., Scholak, T., Yasunaga, M., Wu, C.-S., Zhong, M., Yin, P., Wang, S. I., et al. Unifiedskg: Unifying and multi-tasking structured knowledge grounding with text-to-text language models. *arXiv preprint arXiv:2201.05966*, 2022.

Yin, P. and Neubig, G. Reranking for neural semantic parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 4553–4559, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1447. URL https://aclanthology.org/P19-1447.

Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., Zhang, Z., and Radev, D. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 3911–3921, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1425. URL https://aclanthology.org/D18-1425.

Zelle, J. M. and Mooney, R. J. Learning to parse database queries using inductive logic programming. In Clancey, W. J. and Weld, D. S. (eds.), *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 2*, pp. 1050–1055. AAAI Press / The MIT Press, 1996. URL http://www.aaai.org/Library/AAAI/1996/aaai96-156.php.

Zettlemoyer, L. S. and Collins, M. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI '05, Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence, Edinburgh, Scotland, July 26-29, 2005*, pp. 658–666. AUAI Press, 2005. URL https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=1209&proceeding_id=21.

Zhang, T., Yu, T., Hashimoto, T. B., Lewis, M., Yih, W.-t., Fried, D., and Wang, S. I. Coder reviewer reranking for code generation. *arXiv preprint arXiv:2211.16490*, 2022.

Zhong, V., Xiong, C., and Socher, R. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017. URL http://arxiv.org/abs/1709.00103.

Zhou, F., Hu, M., Dong, H., Cheng, Z., Han, S., and Zhang, D. Tacube: Pre-computing data cubes for answering numerical-reasoning questions over tabular data. *arXiv preprint arXiv:2205.12682*, 2022.

Zhou, S., Yin, P., and Neubig, G. Hierarchical control of situated agents through natural language. *arXiv preprint arXiv:2109.08214*, 2021.

A. Additional Implementation Details

	Spider	WTQ	GSM8k	MBPP
Verification Settings				
Input Format	NL+ SQL+ Exec.	NL+ SQL+ Exec.	NL+ Prog.+ Exec.	NL+ Prog.+ Exec.
Normalize Gen. Prob.	No	No	Yes	Yes
Batch Size*	8	8	8	8
Downsample†	20	5	5	4
% of Positive Label				
Codex	68.0%	47.6%	61.1%	56.6%
InCoder	9.2%	-	2.2%	-
CodeGen	8.6%	-	4.9%	-
% of Unique Programs				
Codex	30.6%	30.1%	90.3%	93.8%
InCoder	94.2%	-	99.2%	-
CodeGen	94.5%	-	99.3%	-

Table 7: Detailed dataset-specific settings and statistics for the training of the verifiers. *: number of examples per batch during training. †: number of program samples per example in the batch. The percentages are all among the sampled programs per example, and are all measured on the training set.

Code LLM sampling. We use temperature sampling to obtain program candidates given the input formats and sampling hyperparameters as described in Table 7. We set the temperature as $T = 0.6$ for Codex and $T = 0.8$ for InCoder and CodeGen, as the optimal temperatures for the best pass@k by referring to the original papers (Fried et al., 2022; Nijkamp et al., 2022). An ablation study on sampling budget is reported in §5.1.

Few-shot prompt construction. The numbers of few-shot exemplars to include in the prompt for different datasets are shown in Table 1. All exemplars are randomly sampled and ordered from the training set, with the exception of MBPP, where we use the 3 examples provided by the original dataset. Full prompts used for each dataset are shown in Appendix C.

Dataset-specific setups. The detailed experiment setups for specific datasets are shown as Table 7. In particular, we use normalized probability for GSM8k and MBPP datasets as we find these two datasets can benefit from such normalization. We think this is because the Python programs have higher variance in length due to the flexible grammar and being more expressive. Moreover, the percentage of positive labels also denotes the "random" baseline, which is the expected execution accuracy by randomly picking from the sampled programs. This provides the additional perspective of the ability of the code LLMs, as well as the difficult of learning the verifiers for them.

Target LLM & ML+EP Baseline		Source LLM (% Positive Labels) Codex (64.0%)	InCoder (9.2%)	CodeGen (8.6%)
Codex	77.3	82.0 (+4.7)	81.7 (+4.4)	80.8 (+3.5)
InCoder	41.2	46.4 (+5.2)	54.1 (+12.9)	47.6 (+6.4)
CodeGen	37.7	44.7 (+7.0)	48.9 (+11.2)	51.0 (+13.3)

(a) Between the code LLMs transfer results on Spider.

Target LLM & ML+EP Baseline		Source LLM (% Positive Labels) Codex (53.4%)	InCoder (2.3%)	CodeGen (5.0%)
Codex	72.1	83.7 (+11.6)	70.0 (-2.1)	71.9 (-0.2)
InCoder	4.3	8.3 (+4.0)	11.9 (+7.6)	12.3 (+8.0)
CodeGen	9.6	18.4 (+8.8)	20.7 (+11.1)	22.1 (+12.5)

(b) Between the code LLMs transfer results on GSM8k.

Table 8: Execution accuracy of training verifiers on the programs sampled from source code LLM and apply to the target code LLM. The best and second best performance per row is highlighted accordingly.

Verifier Input Examples Here we show examples of the inputs to the verifiers for different datasets in Table 11.

B. Additional Results

B.1. Transfer Learning between Code LLMs

One other way to avoid the cost of sampling from code LLMs is to train verifiers using samples from one LLM and directly apply to the programs sampled from a different LLM, *i.e.,* between LLM transfer, and we show the results of such on Spider and GSM8k in Table 8. From the results, we can first observe that LEVER still non-trivially improves the baseline performance most of the time, with the exception of transferring from InCoder and CodeGen to Codex on the GSM8k dataset. This suggests that the knowledge learned by the verifiers are generalizable to different LLM outputs. Moreover, we can see that the transfer typically works better when the percentage of positive labels are closer, as the transfer is more successful between the InCoder and CodeGen models than that with Codex. These results show between-LLM transfer as an interesting way to reduce the training data need for LEVER.

B.2. Ablation on Base LMs for Verification

In this work, we treat the choice of base models for the verifiers as a hyperparameter and use the best performing model for further experiments. Here we show the performance of all the base models we attempted on the four datasets, with

Base LMs	Spider	WTQ	GSM8k	MBPP
T5-base	**82.0**	64.8	82.4	76.8
T5-large	81.9	**65.0**	82.5	**77.3**
T5-3B	83.1	64.7	**84.4**	-
RoBERTa-large	-	64.3	**84.4**	-

Table 9: Ablations on using different base models for the verifiers. -: base LM not tested on this dataset.

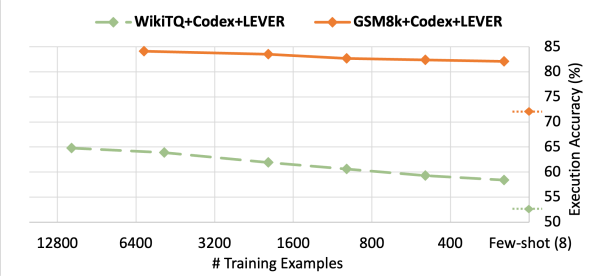


Figure 7: Ablation on number of training examples for Codex+LEVER on the WTQ and GSM8k datasets. Data markers on the y-axis denote the ML+EP performances as baselines. T5-base is used for LEVER.

results in Table 9.

B.3. Training Example Scaling for WTQ and GSM8k

Due to space limit, we are only able to show the ablation in number of training example for Spider. Here in Figure 7, we show the results for WikiTQ and GSM8k as well. From the results, we can see that the learning of LEVER is also very data efficient on those two benchmarks, as non-trivial improvements can be observed even when only 250 training examples are given.

B.4. WikiTQ Results with the Official Evaluator

Following Cheng et al. (2022), we fix the official evaluator of WikiTQ by normalizing units, Boolean values, etc. Here we also report the performance of LEVER with previous work based on the official WikiTQ evaluator in Table 10. From the results, we can see that LEVER still presents the state-of-the-art result under this setting.

B.5. Case Study

Here we give some concrete examples to illustrate how LEVER work and when does it fail in Table 12. In the first example from the Spider dataset, we can see that program candidate $\hat{y}_2$ selects from the wrong table, which results in an execution error. This is easily detected by the verifier thus put a low verification probability on such program. Meanwhile, the execution result $\hat{z}_1$ from program $\hat{y}_1$ seems much more likely to be the answer of the question to the verifier. In the second example from WikiTQ, however, the execution results $\hat{z}_1$ and $\hat{z}_2$ do not provide clear information as they are both county names. In this case, the verifier does not possess much more meaningful information than the generator, thus not able to identify the incorrect program.

Methods	Dev	Test
Previous Work without Finetuning		
Codex QA‡ (Cheng et al., 2022)	49.3	47.6
Codex SQL‡ (Cheng et al., 2022)	57.6	55.1
Codex Binder‡ (Cheng et al., 2022)	**62.6**	61.9
Previous Work with Finetuning		
TaPEX* (Liu et al., 2021)	57.0	57.5
TaCube (Zhou et al., 2022)	59.7	59.6
OmniTab (Jiang et al., 2022)	-	**62.8**
This Work Using code-davinci-002		
Greedy	47.2	50.9
ML	48.3	50.9
EP + ML	50.1	52.5
EP + Voting	50.6	53.6
LEVER	$61.1_{\pm 0.2}$	$\mathbf{62.9}_{\pm 0.2}$
Oracle	70.9	74.6

Table 10: Execution accuracy on the WTQ dataset with the official WTQ executor. ‡: a normalizer to recognize date is added to the official executor.

C. Prompts for Few-shot Generation

Finally, we append the full prompts we used for few-shot prompting the code LLMs for Spider (Table 13, Table 14, Table 15), WikiTQ (Table 16, Table 17), GSM8k (Table 18, Table 19), and MBPP (Table 20).

SPIDER/WIKITQ: question + SQL + linearized result table

Input:
```
-- question:  List the name, born state and age of the heads of departments ordered
by age.|
-- SQL:|select name, born_state, age from head join management on head.head_id =
management.head_id order by age|
-- exec result:|/*| name born_state age| Dudley Hart California 52.0| Jeff Maggert
Delaware 53.0|Franklin Langham Connecticut 67.0| Billy Mayfair California 69.0|
K. J. Choi Alabama 69.0|*/
```
Output: no

GSM8K: question + idiomatic program + answer variable

Input:
```
Carly recently graduated and is looking for work in a field she studied for.  She
sent 200 job applications to companies in her state, and twice that number to
companies in other states.  Calculate the total number of job applications she has
sent so far.  |
n_job_apps_in_state = 200
n_job_apps_out_of_state = n_job_apps_in_state * 2
answer = n_job_apps_in_state + n_job_apps_out_of_state |
'answer':  600
```
Output: yes

MBPP: task description + function + return type & value

Input:
```
# description
Write a function to find the n-th power of individual elements in a list using lambda
function.

# program
def nth_nums(nums,n):
    result_list = list(map(lambda x:  x ** n, nums))
    return (result_list)

# execution
# return:  (list)=[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
# return:  (list)=[1000, 8000, 27000]
# return:  (list)=[248832, 759375]
```
Output: yes

Table 11: Examples of verifier inputs on the datasets. Some newlines are manually inserted for better display.

SPIDER EXAMPLE

Question x: Find the total ranking points for each player and their first name.
Program $\hat{y}_1$ (correct):
```
select first_name, sum(ranking_points) from players join rankings on player.player_id =
  rankings.player_id group by first_name
```
Program $\hat{y}_2$ (incorrect):
```
select first_name, sum(ranking_points) from rankings join players on rankings.player_id
  = players.player_id group by player_id
```
Execution info $\hat{z}_1$:
```
Aastha | 68; Abbi | 304; Abbie | ...
```
Execution info $\hat{z}_2$:
```
ERROR: not column named ...
```

WIKITQ EXAMPLE

Question x: When ranking the counties from first to last in terms of median family income, the first would be?
Program $\hat{y}_1$ (incorrect):
```
select county from main_table order by median_family_income_number limit 1
```
Program $\hat{y}_2$ (correct):
```
select county from main_table order by median_family_income_number desc limit 1
```
Execution info $\hat{z}_1$: `county | jefferson`
Execution info $\hat{z}_2$: `county | sanders`

Table 12: Case study for the WikiTQ and Spider datasets. Program $\hat{y}_1$ is **ranked above** program $\hat{y}_2$ in both examples. The main differences in the SQL programs that lead to error are highlighted.

```
-- Translate natural language questions into SQL queries.

-- Example:

-- Database game_injury:
-- Table stadium: id, name, Home_Games, Average_Attendance, Total_Attendance, Capacity_Percentage
-- Table game: stadium_id, id, Season, Date, Home_team, Away_team, Score, Competition
-- Table injury_accident: game_id, id, Player, Injury, Number_of_matches, Source
-- Question: How many distinct kinds of injuries happened after season 2010?
-- SQL:
SELECT count(DISTINCT T1.Injury) FROM injury_accident AS T1 JOIN game AS T2 ON T1.game_id = T2.id WHERE T2.
    Season > 2010

-- Example:

-- Database farm:
-- Table city: City_ID, Official_Name, Status, Area_km_2, Population, Census_Ranking
-- Table farm: Farm_ID, Year, Total_Horses, Working_Horses, Total_Cattle, Oxen, Bulls, Cows, Pigs,
    Sheep_and_Goats
-- Table farm_competition: Competition_ID, Year, Theme, Host_city_ID, Hosts
-- Table competition_record: Competition_ID, Farm_ID, Rank
-- Question: Return the hosts of competitions for which the theme is not Aliens?
-- SQL:
SELECT Hosts FROM farm_competition WHERE Theme != 'Aliens'
```

Table 13: The prompt we use for the Spider dataset for few-shot generation with code LLMs. Only the first 2 exemplars are shown here, which is also the only two used for InCoder/CodeGen due to limits of model length and computation. 8 total exemplars are used for Codex, and the rest are shown in Table 14 and Table 15.

```
-- Example:

-- Database school_finance:
-- Table School: School_id, School_name, Location, Mascot, Enrollment, IHSAA_Class, IHSAA_Football_Class,
     County
-- Table budget: School_id, Year, Budgeted, total_budget_percent_budgeted, Invested,
     total_budget_percent_invested, Budget_invested_percent
-- Table endowment: endowment_id, School_id, donator_name, amount
-- Question: Show the average, maximum, minimum enrollment of all schools.
-- SQL:
SELECT avg(Enrollment) , max(Enrollment) , min(Enrollment) FROM School

-- Example:

-- Database cre_Docs_and_Epenses:
-- Table Ref_Document_Types: Document_Type_Code, Document_Type_Name, Document_Type_Description
-- Table Ref_Budget_Codes: Budget_Type_Code, Budget_Type_Description
-- Table Projects: Project_ID, Project_Details
-- Table Documents: Document_ID, Document_Type_Code, Project_ID, Document_Date, Document_Name,
     Document_Description, Other_Details
-- Table Statements: Statement_ID, Statement_Details
-- Table Documents_with_Expenses: Document_ID, Budget_Type_Code, Document_Details
-- Table Accounts: Account_ID, Statement_ID, Account_Details
-- Question: Return the ids and details corresponding to projects for which there are more than two documents.
-- SQL:
SELECT T1.Project_ID , T1.Project_Details FROM Projects AS T1 JOIN Documents AS T2 ON T1.Project_ID = T2.
     Project_ID GROUP BY T1.Project_ID HAVING count(*) > 2

-- Example:

-- Database local_govt_in_alabama:
-- Table Services: Service_ID, Service_Type_Code
-- Table Participants: Participant_ID, Participant_Type_Code, Participant_Details
-- Table Events: Event_ID, Service_ID, Event_Details
-- Table Participants_in_Events: Event_ID, Participant_ID
-- Question: List the type of the services in alphabetical order.
-- SQL:
SELECT Service_Type_Code FROM Services ORDER BY Service_Type_Code

-- Example:

-- Database cre_Theme_park:
-- Table Ref_Hotel_Star_Ratings: star_rating_code, star_rating_description
-- Table Locations: Location_ID, Location_Name, Address, Other_Details
-- Table Ref_Attraction_Types: Attraction_Type_Code, Attraction_Type_Description
-- Table Visitors: Tourist_ID, Tourist_Details
-- Table Features: Feature_ID, Feature_Details
-- Table Hotels: hotel_id, star_rating_code, pets_allowed_yn, price_range, other_hotel_details
-- Table Tourist_Attractions: Tourist_Attraction_ID, Attraction_Type_Code, Location_ID, How_to_Get_There, Name
     , Description, Opening_Hours, Other_Details
-- Table Street_Markets: Market_ID, Market_Details
-- Table Shops: Shop_ID, Shop_Details
-- Table Museums: Museum_ID, Museum_Details
-- Table Royal_Family: Royal_Family_ID, Royal_Family_Details
-- Table Theme_Parks: Theme_Park_ID, Theme_Park_Details
-- Table Visits: Visit_ID, Tourist_Attraction_ID, Tourist_ID, Visit_Date, Visit_Details
-- Table Photos: Photo_ID, Tourist_Attraction_ID, Name, Description, Filename, Other_Details
-- Table Staff: Staff_ID, Tourist_Attraction_ID, Name, Other_Details
-- Table Tourist_Attraction_Features: Tourist_Attraction_ID, Feature_ID
-- Question: Show the average price range of hotels that have 5 star ratings and allow pets.
-- SQL:
SELECT avg(price_range) FROM Hotels WHERE star_rating_code = "5" AND pets_allowed_yn = 1
```

Table 14: The prompt we use for the Spider dataset for few-shot generation with code LLMs (Part 2), continued from Table 13.

```
-- Example:

-- Database insurance_fnol:
-- Table Customers: Customer_ID, Customer_name
-- Table Services: Service_ID, Service_name
-- Table Available_Policies: Policy_ID, policy_type_code, Customer_Phone
-- Table Customers_Policies: Customer_ID, Policy_ID, Date_Opened, Date_Closed
-- Table First_Notification_of_Loss: FNOL_ID, Customer_ID, Policy_ID, Service_ID
-- Table Claims: Claim_ID, FNOL_ID, Effective_Date
-- Table Settlements: Settlement_ID, Claim_ID, Effective_Date, Settlement_Amount
-- Question: Find all the phone numbers.
-- SQL:
SELECT Customer_Phone FROM available_policies

-- Example:

-- Database cre_Theme_park:
-- Table Ref_Hotel_Star_Ratings: star_rating_code, star_rating_description
-- Table Locations: Location_ID, Location_Name, Address, Other_Details
-- Table Ref_Attraction_Types: Attraction_Type_Code, Attraction_Type_Description
-- Table Visitors: Tourist_ID, Tourist_Details
-- Table Features: Feature_ID, Feature_Details
-- Table Hotels: hotel_id, star_rating_code, pets_allowed_yn, price_range, other_hotel_details
-- Table Tourist_Attractions: Tourist_Attraction_ID, Attraction_Type_Code, Location_ID, How_to_Get_There, Name
     , Description, Opening_Hours, Other_Details
-- Table Street_Markets: Market_ID, Market_Details
-- Table Shops: Shop_ID, Shop_Details
-- Table Museums: Museum_ID, Museum_Details
-- Table Royal_Family: Royal_Family_ID, Royal_Family_Details
-- Table Theme_Parks: Theme_Park_ID, Theme_Park_Details
-- Table Visits: Visit_ID, Tourist_Attraction_ID, Tourist_ID, Visit_Date, Visit_Details
-- Table Photos: Photo_ID, Tourist_Attraction_ID, Name, Description, Filename, Other_Details
-- Table Staff: Staff_ID, Tourist_Attraction_ID, Name, Other_Details
-- Table Tourist_Attraction_Features: Tourist_Attraction_ID, Feature_ID
-- Question: Which transportation method is used the most often to get to tourist attractions?
-- SQL:
SELECT How_to_Get_There FROM Tourist_Attractions GROUP BY How_to_Get_There ORDER BY COUNT(*) DESC LIMIT 1
```

Table 15: The prompt we use for the Spider dataset for few-shot generation with code LLMs (Part 3), continued from Table 13 and Table 14.

```
-- Translate natural language questions into SQL queries.

-- Example:

-- Database 204_126:
-- Table main_table: id (1), agg (0), place (t1), place_number (1.0), player (larry nelson), country (united
     states), score (70-72-73-72=287), score_result (287), score_number (70), score_number1 (70),
     score_number2 (72), score_number3 (73), score_number4 (72), to_par (-1), to_par_number (-1.0),
     money_lrb_rrb (playoff), money_lrb_rrb_number (58750.0)
-- Question: what was first place 's difference to par ?
-- SQL:
select to_par from main_table where place_number = 1

-- Example:

-- Database 204_522:
-- Table main_table: id (1), agg (0), boat_count (4911), boat_count_number (4911), boat_count_minimum (4951),
     boat_count_maximum (4955), name (ha-201), builder (sasebo naval arsenal), laid_down (01-03-1945),
     laid_down_number (1), laid_down_parsed (1945-01-03), laid_down_year (1945), laid_down_month (1),
     laid_down_day (3), launched (23-04-1945), launched_number (23), launched_parsed (1945-04-23),
     launched_year (1945), launched_month (4), launched_day (23), completed (31-05-1945), completed_number
     (31), completed_parsed (1945-05-31), completed_year (1945), completed_month (5), completed_day (31), fate
      (decommissioned 30-11-1945. scuttled off goto islands 01-04-1946)
-- Question: when was a boat launched immediately before ha-206 ?
-- SQL:
select name from main_table where launched_parsed < ( select launched_parsed from main_table where name = 'ha
     -206' ) order by launched_parsed desc limit 1

-- Example:

-- Database 204_877:
-- Table main_table: id (1), agg (0), place (1), place_number (1.0), position (mf), number (4), number_number
     (4.0), name (ryan hall), league_two (10), league_two_number (10.0), fa_cup (1), fa_cup_number (1.0),
     league_cup (0), league_cup_number (0.0), fl_trophy (3), fl_trophy_number (3.0), total (14), total_number
     (14.0)
-- Question: who scored more , grant or benyon ?
-- SQL:
select name from main_table where name in ( 'anthony grant' , 'elliot benyon' ) order by total_number desc
     limit 1

-- Example:

-- Database 204_400:
-- Table main_table: id (1), agg (0), district (1), district_number (1.0), senator (kenneth p. lavalle), party
      (republican), caucus (republican), first_elected (1976), first_elected_number (1976),
     counties_represented (suffolk), counties_represented_length (1)
-- Table t_counties_represented_list: m_id (1), counties_represented_list (suffolk)
-- Question: how many republicans were elected after 2000 ?
-- SQL:
select count ( * ) from main_table where party = 'republican' and first_elected_number > 2000
```

Table 16: The prompt we use for the WTQ dataset for few-shot generation with code LLMs (Part 1).

```
-- Example:

-- Database 203_208:
-- Table main_table: id (1), agg (0), team (dinamo minsk), location (minsk), venue (dinamo, minsk), capacity
     (41040), capacity_number (41040.0), position_in_1993_94 (1), position_in_1993_94_number (1.0)
-- Table t_venue_address: m_id (1), venue_address (dinamo)
-- Question: what is the number of teams located in bobruisk ?
-- SQL:
select count ( team ) from main_table where location = 'bobruisk'

-- Example:

-- Database 203_60:
-- Table main_table: id (1), agg (0), outcome (winner), no (1), no_number (1.0), date (20 july 1981),
     date_number (20), date_parsed (1981-07-20), date_year (1981), date_month (7), date_day (20), championship
      (bastad, sweden), surface (clay), opponent_in_the_final (anders jarryd), score_in_the_final (6-2, 6-3),
     score_in_the_final_length (2)
-- Table t_championship_address: m_id (1), championship_address (bastad)
-- Table t_score_in_the_final_list: m_id (1), score_in_the_final_list (6-2)
-- Table t_score_in_the_final_list_first: m_id (1), score_in_the_final_list_first (6-2)
-- Table t_score_in_the_final_list_second: m_id (7), score_in_the_final_list_second (1-7)
-- Table t_score_in_the_final_list_first_number: m_id (1), score_in_the_final_list_first_number (6)
-- Table t_score_in_the_final_list_first_number1: m_id (1), score_in_the_final_list_first_number1 (6)
-- Table t_score_in_the_final_list_first_number2: m_id (1), score_in_the_final_list_first_number2 (2)
-- Table t_score_in_the_final_list_second_number: m_id (7), score_in_the_final_list_second_number (1)
-- Table t_score_in_the_final_list_second_number1: m_id (7), score_in_the_final_list_second_number1 (1)
-- Table t_score_in_the_final_list_second_number2: m_id (7), score_in_the_final_list_second_number2 (7)
-- Question: which month were the most championships played ?
-- SQL:
select date_month from main_table group by date_month order by count ( * ) desc limit 1

-- Example:

-- Database 203_462:
-- Table main_table: id (1), agg (0), year (2006), year_number (2006), division (4), division_number (4.0),
     league (usl pdl), regular_season (4th, heartland), regular_season_length (2), playoffs (did not qualify),
      open_cup (did not qualify)
-- Table t_regular_season_list: m_id (1), regular_season_list (4th)
-- Question: what year was more successful , 2012 or 2007 ?
-- SQL:
select year_number from main_table where year_number in ( 2012 , 2007 ) order by regular_season limit 1

-- Example:

-- Database 204_139:
-- Question: are their any other airports that are type `` military/public '' besides eagle farm airport ?
-- Table main_table: id (1), agg (0), community (antil plains), airport_name (antil plains aerodrome), type (
     military), coordinates (19 26'36''s 146 49'29''e/19.44333 s 146.82472 e)
-- SQL:
select ( select count ( airport_name ) from main_table where type = 'military/public' and airport_name != '
     eagle farm airport' ) > 0
```

Table 17: The prompt we use for the WTQ dataset for few-shot generation with code LLMs (Part 2).

```
## Cristina, John, Clarissa and Sarah want to give their mother a photo album for her birthday. Cristina
     brings 7 photos, John brings 10 photos and Sarah brings 9 photos. If the photo album has 40 slots
     available, how many photos does Clarissa need to bring in order to complete the photo album?
n_photo_cristina = 7
n_photo_john = 10
n_photo_sarah = 9
n_photo_total = n_photo_cristina + n_photo_john + n_photo_sarah
n_slots = 40
n_slots_left = n_slots - n_photo_total
answer = n_slots_left

## Katy, Wendi, and Carrie went to a bread-making party. Katy brought three 5-pound bags of flour. Wendi
     brought twice as much flour as Katy, but Carrie brought 5 pounds less than the amount of flour Wendi
     brought. How much more flour, in ounces, did Carrie bring than Katy?
pound_flour_katy = 3 * 5
pound_flour_wendi = pound_flour_katy * 2
pound_flour_carrie = pound_flour_wendi - 5
pound_diff_carrie_katy = pound_flour_carrie - pound_flour_katy
ounce_diff_carrie_katy = pound_diff_carrie_katy * 16
answer = ounce_diff_carrie_katy

## James takes 2 Tylenol tablets that are 375 mg each, every 6 hours. How many mg does he take a day?
mg_tylenol_per_tablet = 375
mg_tylenol_taken_each_time = 2 * mg_tylenol_per_tablet
hours_per_day = 24
times_per_day = hours_per_day / 6
mg_each_day = mg_tylenol_taken_each_time * times_per_day
answer = mg_each_day

## Kyle bakes 60 cookies and 32 brownies. Kyle eats 2 cookies and 2 brownies. Kyle's mom eats 1 cookie and 2
     brownies. If Kyle sells a cookie for $1 and a brownie for $1.50, how much money will Kyle make if he
     sells all of his baked goods?
n_cookies = 60
n_brownies = 32
n_cookies_left_after_kyle = n_cookies - 2
n_brownies_left_after_kyle = n_brownies - 2
n_cookies_left_after_kyle_mom = n_cookies_left_after_kyle - 1
n_brownies_left_after_kyle_mom = n_brownies_left_after_kyle - 2
money_earned_kyle = n_cookies_left_after_kyle_mom * 1 + n_brownies_left_after_kyle_mom * 1.5
answer = money_earned_kyle

## There were 63 Easter eggs in the yard. Hannah found twice as many as Helen. How many Easter eggs did Hannah
      find?
n_easter_eggs = 63
unit_times = 2
total_units = unit_times + 1
n_easter_eggs_per_unit = n_easter_eggs / total_units
n_easter_eggs_helen = n_easter_eggs_per_unit * 1
n_easter_eggs_hannah = n_easter_eggs_per_unit * 2
answer = n_easter_eggs_hannah

## Ethan is reading a sci-fi book that has 360 pages. He read 40 pages on Saturday morning and another 10
     pages at night. The next day he read twice the total pages as on Saturday. How many pages does he have
     left to read?
n_pages = 360
total_page_saturday = 40 + 10
total_page_next_day = total_page_saturday * 2
total_pages_read = total_page_saturday + total_page_next_day
n_pages_left = n_pages - total_pages_read
answer = n_pages_left
```

Table 18: The prompt we use for the GSM8k dataset for few-shot generation with code LLMs (Part 1).

```
## A library has a number of books. 35% of them are intended for children and 104 of them are for adults. How
     many books are in the library?
percent_books_for_children = 0.35
percent_books_for_adults = 1.0 - percent_books_for_children
n_books_for_adults = 104
n_books_in_total = n_books_for_adults / percent_books_for_adults
answer = n_books_in_total

## Tyler has 21 CDs. He gives away a third of his CDs to his friend. Then he goes to the music store and buys
     8 brand new CDs. How many CDs does Tyler have now?
n_cds_tyler = 21
percent_cds_given_away = 1.0 / 3.0
n_cds_left_after_giving_away = n_cds_tyler - n_cds_tyler * percent_cds_given_away
n_new_cds_purchased = 8
n_cds_now = n_cds_left_after_giving_away + n_new_cds_purchased
answer = n_cds_now
```

Table 19: The prompt we use for the GSM8k dataset for few-shot generation with code LLMs (Part 2).

```
# Write Python function to complete the task and pass the assertion tests.

### Task Start ###
# These are the assertions for your function:
assert similar_elements((3, 4, 5, 6),(5, 7, 4, 10)) == (4, 5)

""" Write a function to find the similar elements from the given two tuple lists. """
def similar_elements(test_tup1, test_tup2):
   res = tuple(set(test_tup1) & set(test_tup2))
   return (res)
### Task End ###

### Task Start ###
# These are the assertions for your function:
assert is_not_prime(2) == False

""" Write a python function to identify non-prime numbers. """
import math
def is_not_prime(n):
   result = False
   for i in range(2,int(math.sqrt(n)) + 1):
      if n % i == 0:
         result = True
   return result
### Task End ###

### Task Start ###
# These are the assertions for your function:
assert heap_queue_largest( [25, 35, 22, 85, 14, 65, 75, 22, 58],3)==[85, 75, 65]

""" Write a function to find the largest integers from a given list of numbers using heap queue algorithm. """
import heapq as hq
def heap_queue_largest(nums,n):
   largest_nums = hq.nlargest(n, nums)
   return largest_nums
### Task End ###
```

Table 20: The prompt we use for the MBPP dataset for few-shot generation with code LLMs.