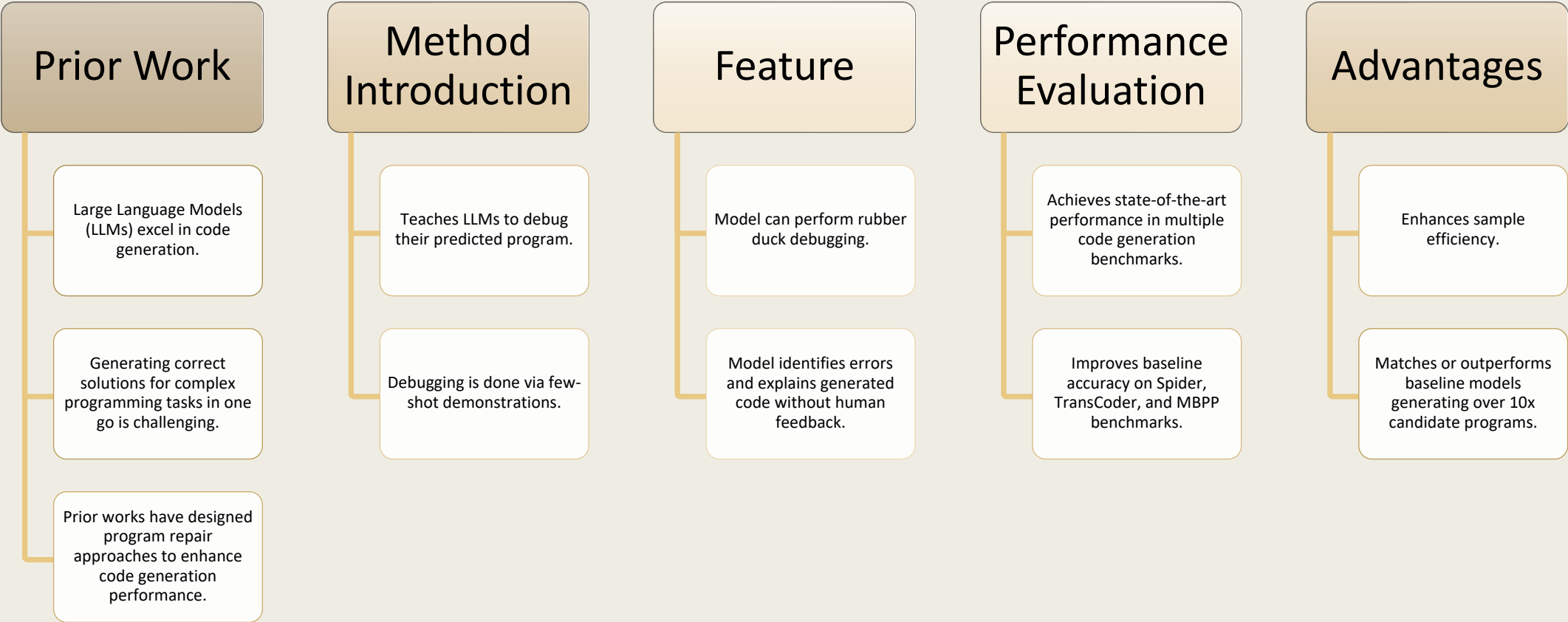


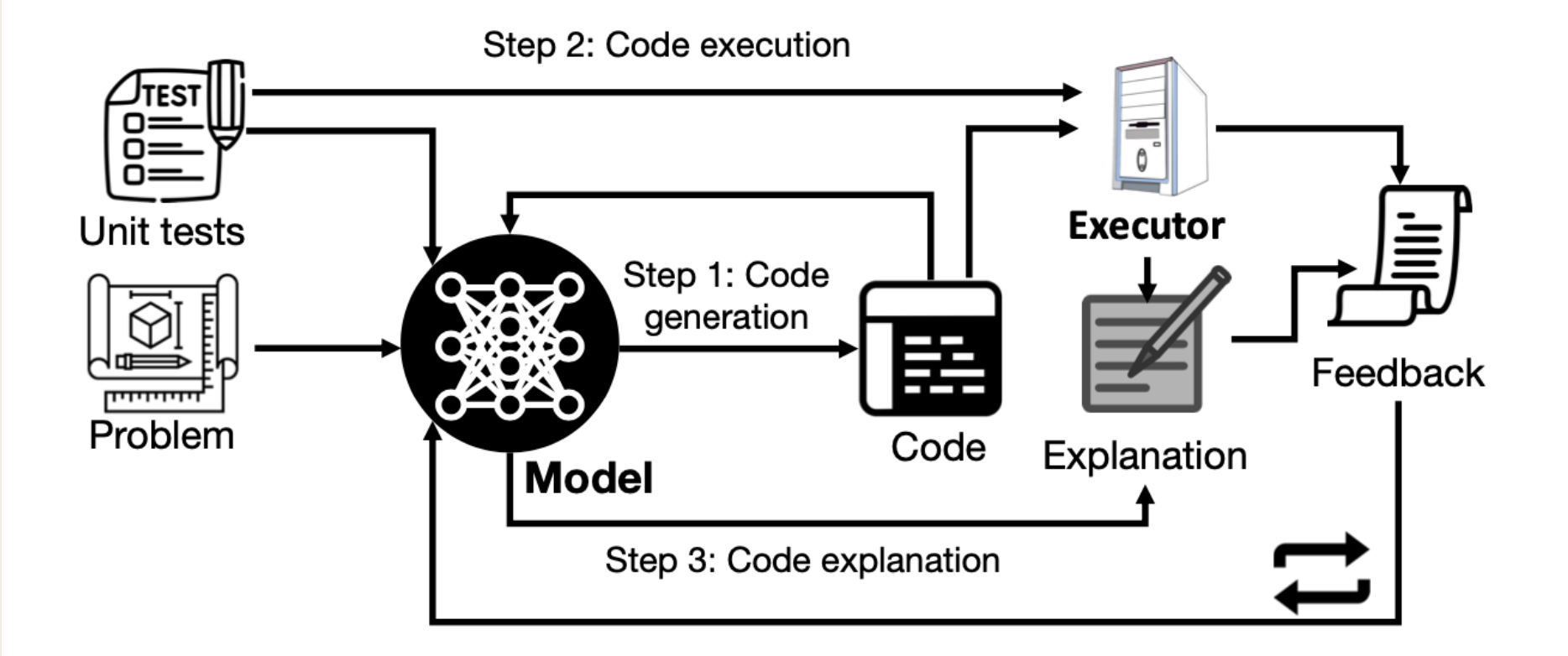
TEACHING LARGE LANGUAGE MODELS TO SELF-DEBUG

Presenter: Yuqi Yan

Introduction



Introduction



Code Generation

Few-shot prompting

1. An approach that uses multiple input-output examples to guide a language model in solving a task.
2. (Problem, SQL) A list of pairs.
3. Prompts can also contain instructions that provide advanced task descriptions.

Which customers have both "On Road" and "Shipped" as order status? List the customer names.

[Question Explanation]

"List the customer names" returns 1 column. The question returns the customer names who have both "On Road" and "Shipped" as order status. So the question returns 1 column.

Step 2: Code explanation

Summarize the return type of the execution.

[SQL]

```
SELECT customers.customer_name FROM customers JOIN orders ON customers.customer_id = orders.customer_id  
WHERE orders.order_status = "On Road" OR orders.order_status = "Shipped"
```

Execution:

| George |

[SQL Explanation]

The execution of the SQL query above would return a table with 1 column.

The first column, "customers.customer_name" would contain the customer names.

With "customers JOIN orders", the table would contain the data about customers with orders.

With "WHERE orders.order_status = 'On Road' OR orders.order_status = 'Shipped'", the table filters the records to only include customers who have order status "On Road" or "Shipped".

So the SQL query returns a table with 1 column, the customer names who have the order status "On Road" or "Shipped".

Step 3: Self-Debugging with explanation

[Question]

[Explanation summary]

As in your explanation, the SQL query returns a table with 1 column, the customer names who have the order status "On Road" or "Shipped". The question returns 1 column, the customer names who have both "On Road" and "Shipped" as order status.

Problem description

```
CREATE TABLE customers (  
  customer_id number ,  
  customer_name text ,  
  customer_details text ,  
  primary key ( customer_id )  
)
```

```
insert into customers (customer_id, customer_name, customer_details) values  
(1, 'Savannah', 'rerum');
```

```
CREATE TABLE orders (  
  order_id number ,  
  customer_id number ,  
  order_status text ,  
  date_order_placed time ,  
  order_details text ,  
  primary key ( order_id ),  
  foreign key ( customer_id ) references customers ( customer_id )  
)
```

```
insert into orders (order_id, customer_id, order_status, date_order_placed,  
order_details) values (1, 2, 'Shipped', '2009-02-21 15:26:19', NULL);  
...
```

Question: Which customers have both "On Road" and "Shipped" as order status? List the customer names.

Output SQL

```
SQL: SELECT customers.customer_name FROM customers JOIN orders ON customer  
customer_id = orders.customer_id  
WHERE orders.order_status = "On Road"  
INTERSECT SELECT customers.customer_name FROM customers JOIN orders ON  
customers.customer_id = orders.customer_id  
WHERE orders.order_status = "Shipped"
```

Code Generation

Execution-based code selection

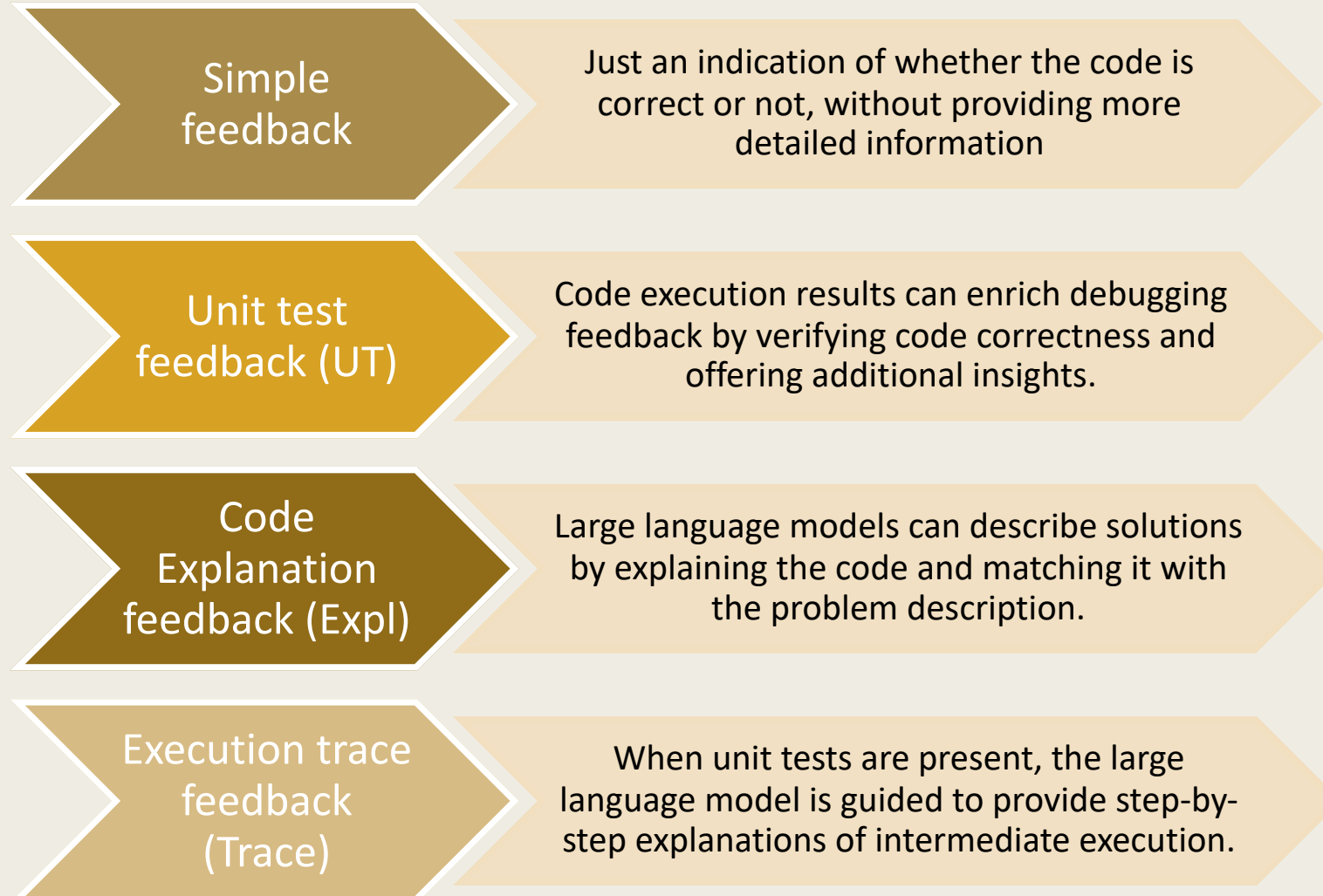
1. This approach is used to improve the performance of large language models.
2. Use majority voting of execution results to select the final prediction code
3. Some code generation tasks come with unit tests that specify the execution behavior of the program

For input data X, the execution result of A is "Result 1", the execution result of B is "Result 2", and the execution result of C is "Result 1";

For input data Y, the execution result of A is "Result 2", the execution result of B is "Result 2", and the execution result of C is "Result 2";

For input data Z, the execution result of A is "Result 1", the execution result of B is "Result 1", and the execution result of C is "Result 1".

Self-Debugging Framework



Applications

-- Text-to-SQL Generation (Text to SQL generation represents a situation where no unit tests are available)

Evaluating SELF-DEBUGGING on the development set of the Spider benchmark

- Step 1

Summarize the question and infer the return type

- Step 2

Execute the SQL query and add the returned table for code explanation

- Step 3

Compare inferred SQL explanation with question description and predict correctness

Applications

-- Text-to-Python Generation

Evaluation on the MBPP test set

- Each problem contains 3 unit tests.
- The first unit test is included in the prompt as part of the problem description and the remaining 2 unit tests are reserved for full evaluation.
- Even if the predicted code passes the given unit tests, the model still needs to infer the correctness of the code.

Experiment

- SELF-DEBUGGING was evaluated on several models with 155 billion parameters, including code- davinci-002, gpt-3.5-turbo, gpt-4, StarCoder.

Comparing SELF-DEBUGGING with two types of code reranking baselines

Models trained for a given task

- Comparing SELF-DEBUGGING with T5-3B + N-best Reranking
- Compared with LEVER

Prompting-based approaches

- Comparing SELF-DEBUGGING with MBR-Exec and Coder-Reviewer

Experiment

--Main Result

Table 1: Comparing SELF-DEBUGGING to prior ranking techniques.

(a) Results on the Spider development set.		(b) Results on MBPP dataset.	
Spider (Dev)		n samples	
<i>w/ training</i>		Prior work	
T5-3B + N-best Reranking	80.6	MBR-Exec	63.0 ($n = 25$)
LEVER (Ni et al., 2023)	81.9	Reviewer	66.9 ($n = 25$)
<i>Prompting only w/o debugging</i>		LEVER	68.9 ($n = 100$)
Coder-Reviewer	74.5	SELF-DEBUGGING (this work)	
MBR-Exec	75.2	Codex	72.2 ($n = 10$)
SELF-DEBUGGING (this work)		Simple	73.6
Codex	81.3	UT	75.2
+ Expl.	84.1	UT + Expl.	75.6

Comparing SELF-DEBUGGING to prior code reranking approaches in Table 1, where both SELF-DEBUGGING and prior prompting-based approaches use Codex.

We demonstrate that SELF-DEBUGGING consistently improves performance.

Experiment

--Main Result

Table 2: Results of SELF-DEBUGGING with different feedback formats.

(a) Results on the Spider development set.

Spider	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	81.3	71.1	73.2	64.7
Simple	81.3	72.2	73.4	64.9
+Expl.	84.1	72.2	73.6	64.9

(b) Results on TransCoder.

TransCoder	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	80.4	89.1	77.3	70.0
Simple	89.3	91.6	80.9	72.9
UT	91.6	92.7	88.8	76.4
+ Expl.	92.5	92.7	90.4	76.6
+ Trace.	87.9	92.3	89.5	73.6

(c) Results on MBPP.

MBPP	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	61.4	67.6	72.8	47.2
Simple	68.2	70.8	78.8	50.6
UT	69.4	72.2	80.6	52.2
+ Expl.	69.8	74.2	80.4	52.2
+ Trace.	70.8	72.8	80.2	53.2

By comparing the feedback format of SELF-DEBUGGING on the Spider benchmark, we see that simple feedback is of very limited use in the absence of unit tests.

Experiment

--Main Result

Table 2: Results of SELF-DEBUGGING with different feedback formats.

(a) Results on the Spider development set.

Spider	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	81.3	71.1	73.2	64.7
Simple	81.3	72.2	73.4	64.9
+Expl.	84.1	72.2	73.6	64.9

(b) Results on TransCoder.

TransCoder	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	80.4	89.1	77.3	70.0
Simple	89.3	91.6	80.9	72.9
UT	91.6	92.7	88.8	76.4
+ Expl.	92.5	92.7	90.4	76.6
+ Trace.	87.9	92.3	89.5	73.6

(c) Results on MBPP.

MBPP	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	61.4	67.6	72.8	47.2
Simple	68.2	70.8	78.8	50.6
UT	69.4	72.2	80.6	52.2
+ Expl.	69.8	74.2	80.4	52.2
+ Trace.	70.8	72.8	80.2	53.2

1. On the Spider dataset, GPT-4 outperforms Codex in both initial SQL generation and self debugging.
2. On the MBPP dataset, GPT-4 outperforms Codex and GPT-3.5 in initial Python code generation.

Experiment

--Ablation Studies (Self-debugging improves the sample efficiency)

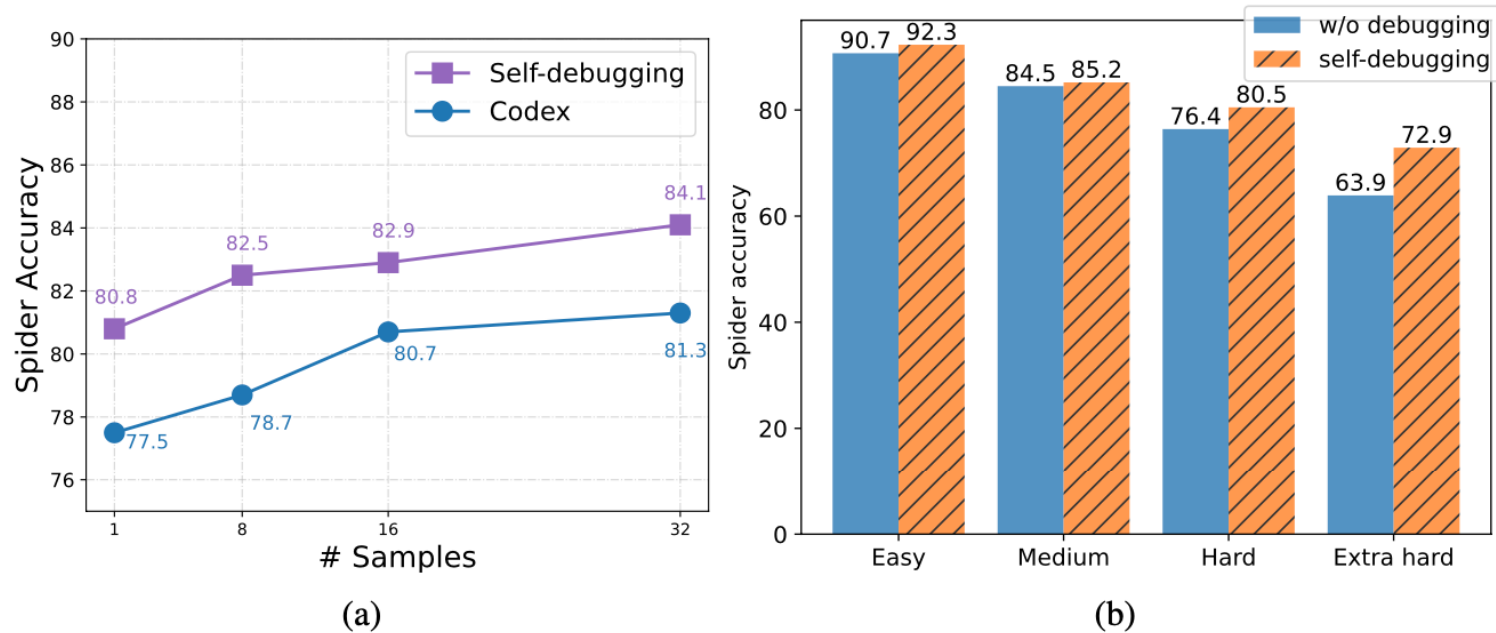


Figure 6: Ablation studies on the Spider development set with Codex. (a) Accuracies with different numbers of initial samples. (b) Breakdown accuracies on problems with different hardness levels.

Experiment

--Ablation Studies (Importance of code execution)

Table 3: Results of SELF-DEBUGGING without unit test execution.

(a) Results on Transcoder.				(b) Results on MBPP			
TransCoder	Codex	GPT-3.5	GPT-4	MBPP	Codex	GPT-3.5	GPT-4
Baseline	80.4	89.1	77.3	Baseline	61.4	67.6	72.8
Simple	83.4	89.1	78.2	Simple	57.6	68.2	76.0
+ Expl.	83.9	89.1	78.0	+ Expl.	64.4	68.2	76.0
+ Trace.	83.9	89.1	78.4	+ Trace.	66.2	69.2	76.4

- With Codex, SELF-DEBUGGING still improves the performance by up to 5%, and the execution trace feedback consistently improves over the simple feedback performance.
- GPT-4 without unit test execution improves the MBPP accuracy by 3.6%, and the improvement on other benchmarks is up to around 1%.
- Compared to Codex, GPT-3.5 and GPT-4 don't benefit much from few-shot prompting in SELF-DEBUGGING, relying solely on their internal code knowledge. Without unit test execution, they tend to be overconfident, with GPT-4 outperforming GPT-3.5 in Python generation.

Experiment

--Ablation Studies (Error Types Fixed by Self-Debugging)

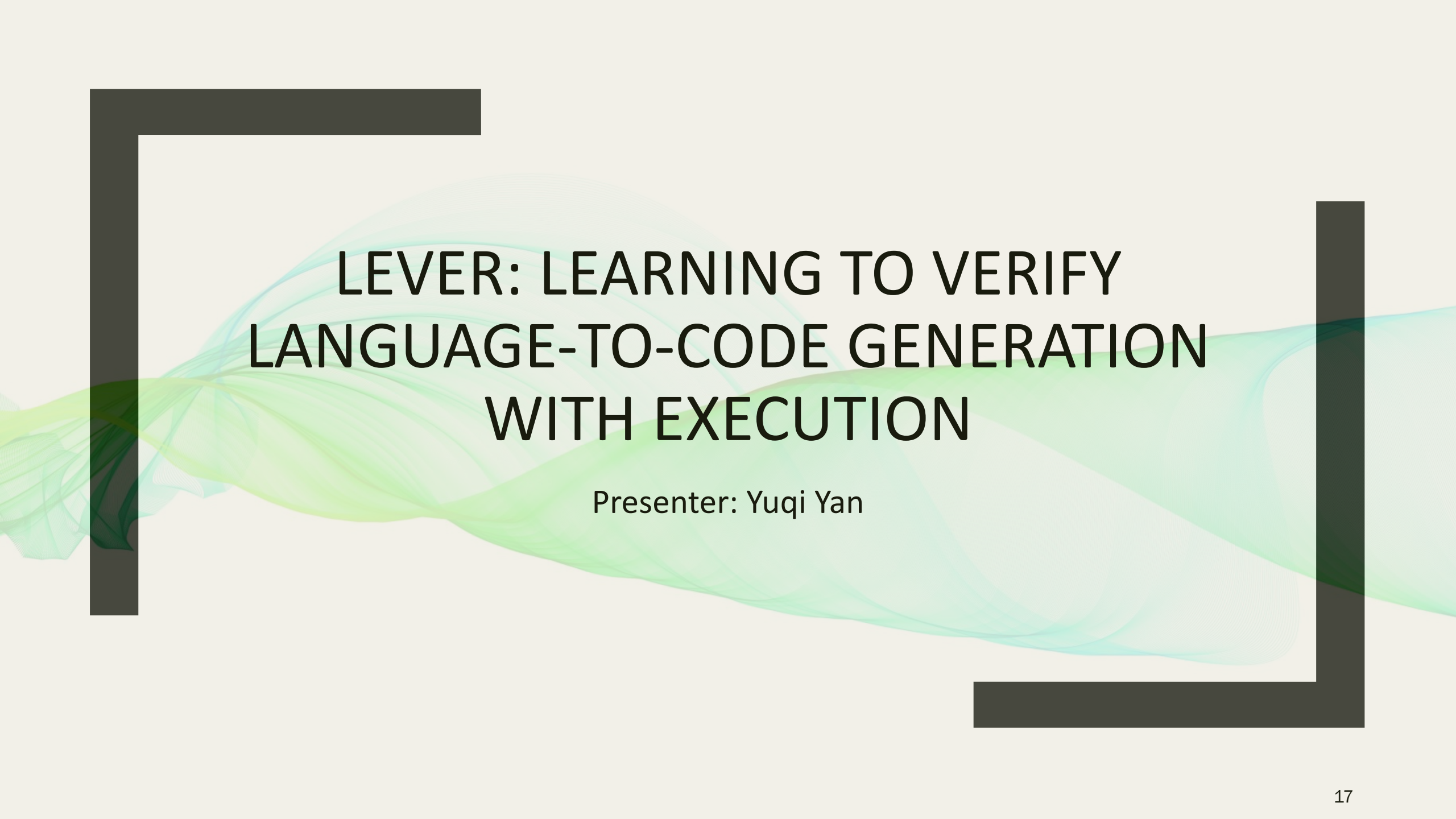
Table 4: Breakdown on percentages of error types fixed by SELF-DEBUGGING.

(a) Breakdown on Spider with `code-davinci-002`. (b) Breakdown on Transcoder with `gpt-3.5-turbo`, and MBPP with `gpt-4`.

Error type	%	Error type	Transcoder	MBPP
Wrong WHERE conditions	25.7	Output mismatch	61.9	69.2
Missing the DISTINCT keyword	17.1	Runtime errors	38.1	30.8
Wrong JOIN clauses	14.3			
Wrong number of SELECT columns	11.4			
Wrong INTERSECT/UNION clauses	8.6			
Wrong aggregate functions and keywords	5.8			
Wrong COUNT columns	5.7			
Wrong column selection	5.7			
Missing nested conditions	5.7			

Conclusion

- SELF-DEBUGGING helps large language models fix code errors by themselves.
- In text-to-SQL tasks, it improves performance by 2-3% on average and 9% on tougher problems.
- For translating code and text-to-Python tasks, it boosts accuracy by up to 12%.
- Teaching models self-debugging improves coding performance.
- Future work aims to enhance model's code explanation and feedback for better debugging.
- Initial findings suggest model-generated error feedback needs improvement for more helpful messages.



LEVER: LEARNING TO VERIFY LANGUAGE-TO-CODE GENERATION WITH EXECUTION

Presenter: Yuqi Yan

Introduction

Prior Work

Obtaining test cases can be challenging

Heuristic methods often struggle to capture the semantic features of execution results effectively.

Method Introduction

Proposing the LEVER method

Improves language-to-code generation by learning to verify the relationship between generated programs and their execution results

Performance Evaluation

The LEVER method consistently outperforms baseline code language models

Establishing new state-of-the-art performance across all datasets.

Introduction

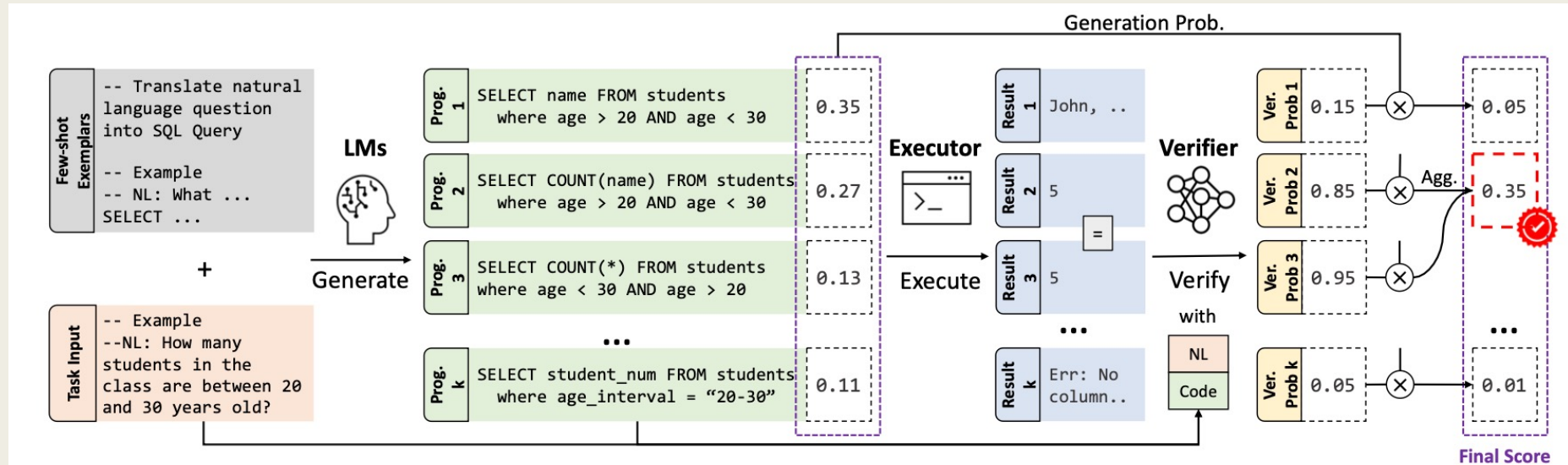


Figure 1: The illustration of LEVER using text-to-SQL as an example. It consists of three steps: 1) *Generation*: sample programs from code LLMs based on the task input and few-shot exemplars; 2) *Execution*: obtain the execution results with program executors; 3) *Verification*: using a learned verifier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

Approach

-- Language-to-Code Generation with Code LLMs

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

$\text{Prompt}(x, \{(x_i, y_i)\}_{i < m})$ is a string representation of the overall input

(generation is also often conditioned on a fixed set of m exemplars)

$$\hat{y}_{\text{greed}} \approx \text{argmax}_y P_{\text{LM}}(y|x)$$

Approach

-- Reranking of Program Candidates

The idea of discriminative reranking is to learn a scoring function $R(x, \hat{y})$

that measures how likely \hat{y} is the best output for input x

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

Given $R(\cdot)$, the reranker outputs the program with the highest reranking score among the set of candidates S

Approach

- Reranking of Program Candidates
- Program Sampling from Code LLMs

Give the input x

instead of performing a greedy search, obtain k programs from $P_{LM}(y|x)$ with temperature sampling

$$\{\hat{y}_i\}_{i=1}^k \sim P_{LM}(y|x)$$

Deduplication to form a set of n (As the same programs may be sampled more than once)

$$S = \{\hat{y}_i\}_{i=1}^n, n < k$$

Approach

- Reranking of Program Candidates
- Verification with Execution

Parameterizing the discriminative reranker as a verification model

$$P_{\theta}(v|x, \hat{y}, \mathcal{E}(\hat{y})) \quad v \in \{0,1\}$$

The reranking probability is the joint probability of generating and passing validation.

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

Approach

-- Learning the Verifiers

-- Training Data Creation

For language-to-code datasets, each example is typically a triplet of (x, y^*, z^*) where $z^* = \varepsilon(y^*)$ is the gold execution result and y^* is the gold program.

Collecting training data:

- Obtaining 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 $P_{LM}(\hat{y}|x)$ and then remove all the duplicated programs.
- For each program candidate $\hat{y} \in S$:
 - Obtain its execution result $\hat{z} = \varepsilon(\hat{y})$.
 - Compare its execution result \hat{z} with the gold standard execution result z^* to obtain its binary verification label v .
 - i.e., $v = \mathbb{1}(\hat{z} = z^*)$.
- For the dataset containing the gold program y^* , use $(x, y^*, z^*, v_{i=1})$ as additional validation training samples.
- With the above steps, a set of validation training samples $\{(x, \hat{y}_i, \hat{z}_i, v_i) \mid \hat{y}_i \in S\}$ is created for each input x .

Approach

- Learning the Verifiers
- 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

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

Experimental Setup

--Datasets

Conducting experiments on four language-to-code datasets across domains of semantic parsing, table QA, math reasoning and basic Python programming.

main settings of these four datasets

	Spider	WikiTQ	GSM8k	MBPP
Domain	Table QA	Table QA	Math QA	Basic Coding
Has program Target	✓ SQL	✓* SQL	✗ Python	✓ Python
<i>Data Statistics</i>				
# Train	7,000	11,321	5,968	378
# Dev	1,032	2,831	1,448	90
# Test	-	4,336	1,312	500
<i>Few-shot Generation Settings</i>				
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.

Experimental Setup

--Code LLMs

-- Baselines and Evaluation Metric

Evaluating LEVER with three code LLMs: Codex, InCoder-6B, and CodeGen-16B-multi

Comparing LEVER to several baseline approaches for generating programs using code LLMs:

- Greedy
- Maximum Likelihood (ML)
- **Error Pruning + ML (EP + ML)**
- Error Pruning + Voting (EP + Votingmost)

Evaluation metric: Use execution accuracy as the main evaluation metric for all datasets

Experimental Setup

-- Implementation Details

Verifier training: Validation training data is created by sampling from LLMs in the training set. However, a large number of samples may lead to a memory shortage. Therefore, random downsampling is performed for each example at each iteration. This ensures that the validator sees a different program in each cycle.

Execution result representation:

- For Spider and WikiTQ, use the linearized resulting tables from SQL execution as the execution results.
- For GSM8k, use the value of the variable named “answer” after executing the program as the execution results.
- For MBPP, use the type and value (cast to string) returned by the functions.

All execution errors are represented as “ERROR: [reason]”, such as “ERROR: Time out”

Main Result

--Effectiveness of LEVER

Methods	Dev
<i>Previous Work without Finetuning</i>	
Rajkumar et al. (2022)	67.0
MBR-Exec (Shi et al., 2022)	75.2
Coder-Reviewer (Zhang et al., 2022)	74.5
<i>Previous Work with Finetuning</i>	
T5-3B (Xie et al., 2022)	71.8
PICARD (Scholak et al., 2021)	75.5
RASAT (Qi et al., 2022)	80.5
<i>This Work with code-davinci-002</i>	
Greedy	75.3
EP + ML	77.3
LEVER 🚀	81.9 ± 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
<i>Previous Work without Finetuning</i>		
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
<i>Previous Work with Finetuning</i>		
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
<i>This Work with code-davinci-002</i>		
Greedy	49.6	53.0
EP + ML	52.7	54.9
LEVER 🚀	64.6 ± 0.2	65.8 ± 0.2

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

Methods	Dev	Test
<i>Previous Work without Finetuning</i>		
MBR-Exec (Shi et al., 2022)	-	63.0
Reviewer (Zhang et al., 2022)	-	66.9
<i>This Work with codex-davinci-002</i>		
Greedy	61.1	62.0
EP + ML	62.2	60.2
LEVER 🚀	75.4 ± 0.7	68.9 ± 0.4

Table 5: Execution accuracy on the MBPP dataset.

Methods	Dev	Test
<i>Previous Work without Finetuning</i>		
PAL (Gao et al., 2022)	-	72.0
Codex + SC [†] (Wang et al., 2022)	-	78.0
PoT-SC (Chen et al., 2022b)	-	80.0
<i>Previous Work with Finetuning</i>		
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
<i>This Work with codex-davinci-002</i>		
Greedy	68.1	67.2
EP + ML	72.1	72.6
LEVER 🚀	84.1 ± 0.2	84.5 ± 0.3

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

Methods	InCoder-6B		CodeGen-16B	
	Spider	GSM8k	Spider	GSM8k
<i>Previous work:</i>				
MBR-EXEC	38.2	-	30.6	-
Reviewer	41.5	-	31.7	-
<i>Baselines:</i>				
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).

Main Result

-- Ablations with LEVER

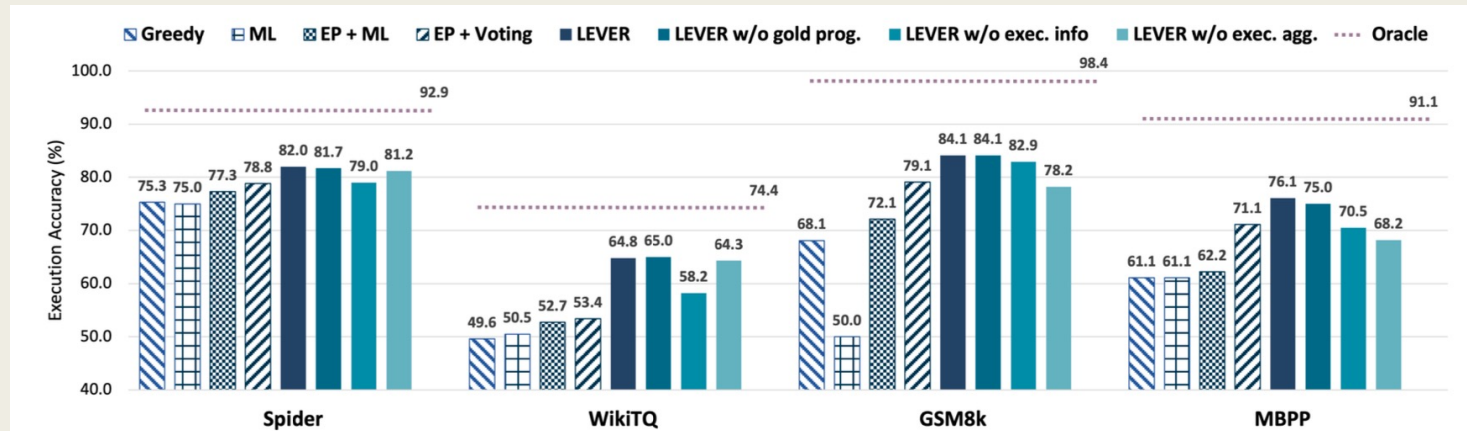


Figure 2: Comparison of LEVER and baselines with Codex-Davinci. LEVER and its ablation results are in solid bars.

- Effect of including execution results
- Effect of execution result aggregation
- Weakly-supervised settings

Methods	InCoder-6B		CodeGen-16B	
	Spider	GSM8k	Spider	GSM8k
<i>Previous work:</i>				
MBR-EXEC	38.2	-	30.6	-
Reviewer	41.5	-	31.7	-
<i>Baselines:</i>				
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).

Analysis

-- Training Example Scaling

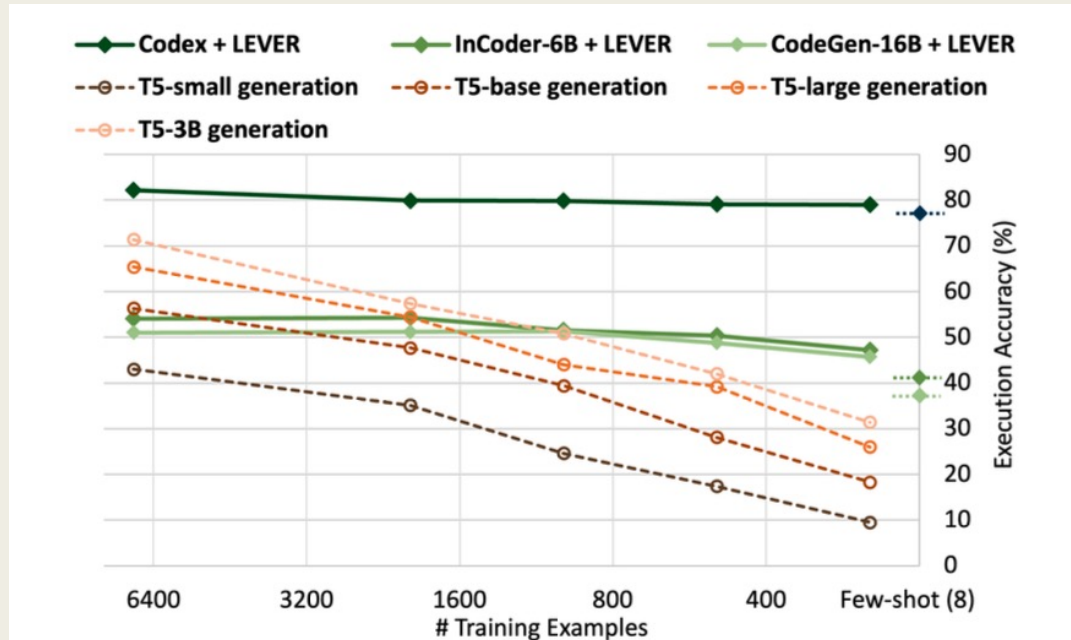
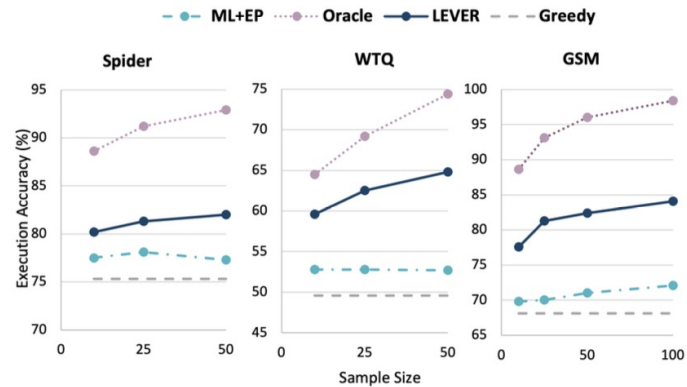


Figure 3: Verification vs. generation performance when decreasing the number of training examples for Spider. Data markers on the y -axis denote the EP+ML baseline, and the x -axis is on the logarithmic scale. T5-base is used as the base model for LEVER. WikiTQ and GSM8k results can be found in [Figure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

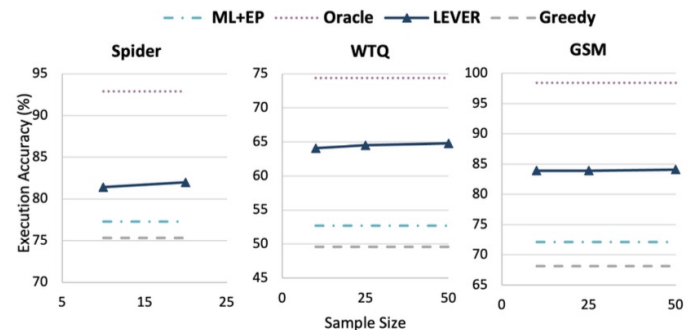
- Demonstrates the change in LEVER's performance on the Spider dataset as the number of training samples decreases.
- LEVER can work with limited resources.
- For harder datasets and weaker LLMs, LEVER has a greater impact of training samples.
- Compare LEVER's performance to a T5 model fine-tuned directly for generation, using the same number of training examples. When there are fewer training examples, the performance of the fine-tuned T5 model drops dramatically.

Analysis

-- Sample Size Scaling



(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.

- 4a: It was found that during inference, LEVER is highly sensitive to the sample size. When the sample size per example is reduced from 50 to 10, LEVER's performance drops by 1.8% (Spider) to 5.2% (WikiTQ).

- 4b: In contrast, LEVER is highly insensitive to the sample size of training data. The performance gap across the three datasets remains below 1%.

Analysis

-- Verifier and Generator Calibration

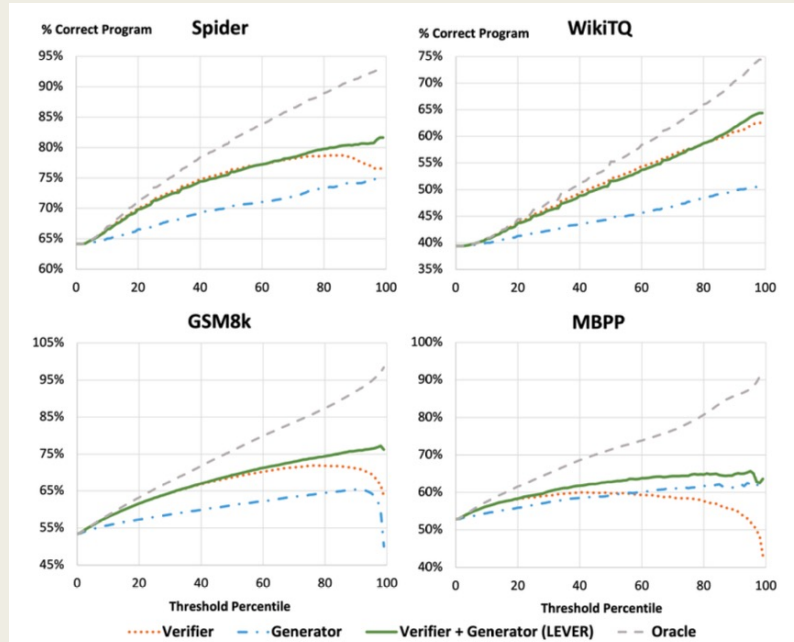
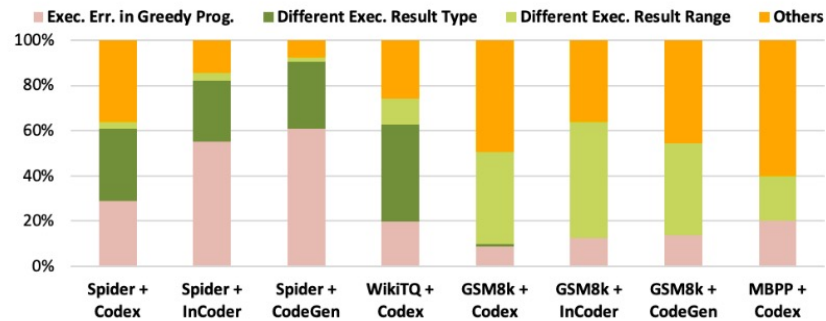


Figure 5: Calibration of the verifier, generator (Codex-Davinci), and their combined probability (used by LEVER). The sampled programs are first ranked by the model probabilities. The x -axis represents the percentage of samples excluded after thresholding, and the y -axis represents the percentage of 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.

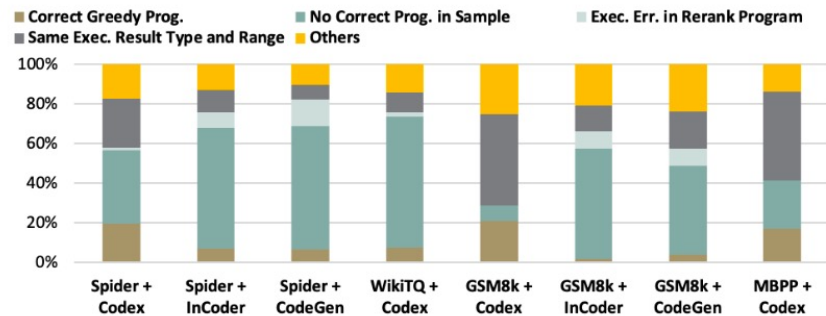
- At lower percentile thresholds, the verifier typically exhibits better calibration than the generator.
- When distinguishing among top-ranked programs, the generator is often better calibrated.
- Combining the probabilities from both the verifier and generator yields the best results across all test datasets.
- Especially on the GSM8k dataset.

Analysis

-- Qualitative Analysis



(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.

The reasons for LEVER's success or failure in improving LLMs' performance are as follows:

- LEVER often succeeds in reranking programs based on crucial information provided by execution results, such as execution errors, variable types, and ranges.
- LEVER may fail if there are no correct programs in the samples, particularly with weaker LLMs.
- When the execution results of incorrect programs match those of correct programs, LEVER may also fail to enhance LLMs' performance.

Limitation

- LEVER needs execution data and a **suitable environment**, but not all applications have them.
- Running model-generated programs can be risky as not all code may be safe.
- PASS@1 metric is used in experiments for tasks like text-to-SQL and math reasoning. For general programming tasks like MBPP, other metrics like PASS@k or N@k **may show different results**.

Conclusion

- The paper suggests mixing generation and verification probabilities for reranking, instead of directly rejecting samples based on verifier output.
- 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 suggests that the program execution results are crucial for verification and the proposed approach is generalizable across different LLMs.

Questions?