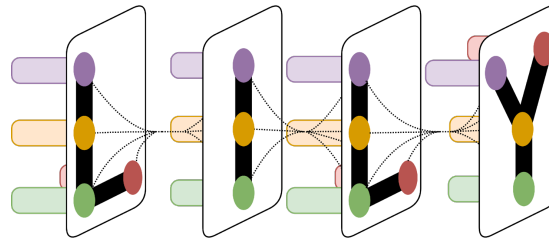


Enhancing Language Models for Program Synthesis using Execution

Talk @MIT_CSAIL

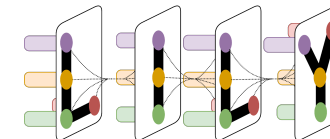


Yale LILY Lab

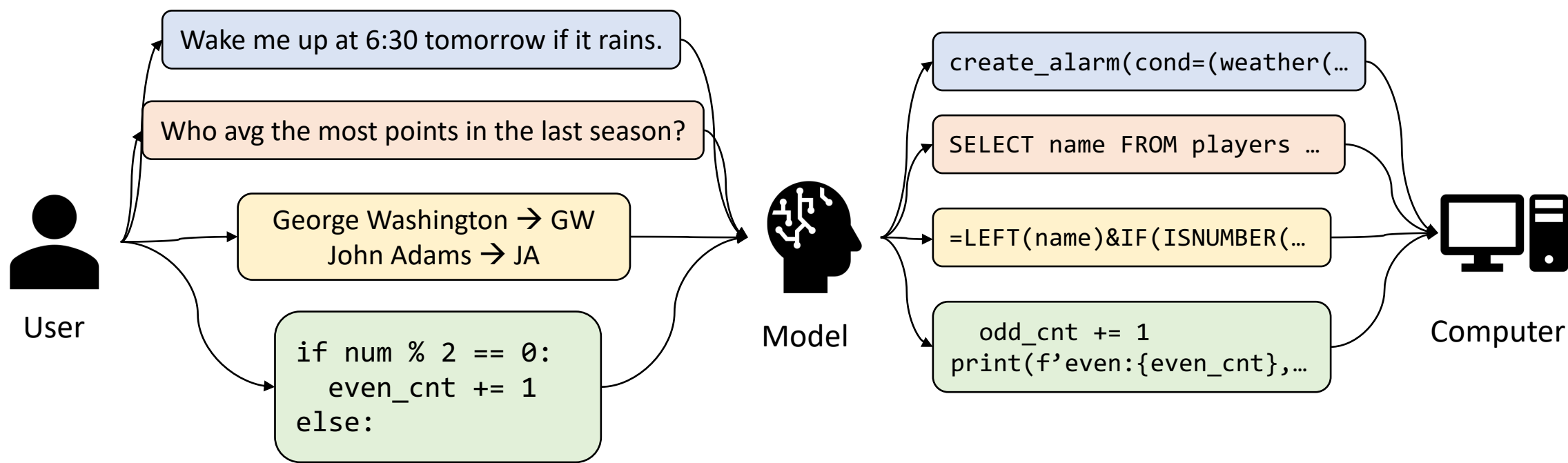
Ansong Ni

03-13-2023

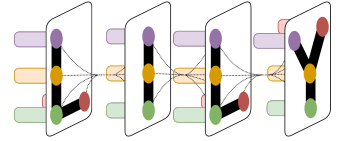
What is Program Synthesis?



- Here we broadly define ***program synthesis*** as the tasks to automatically generate programs from the user intent.



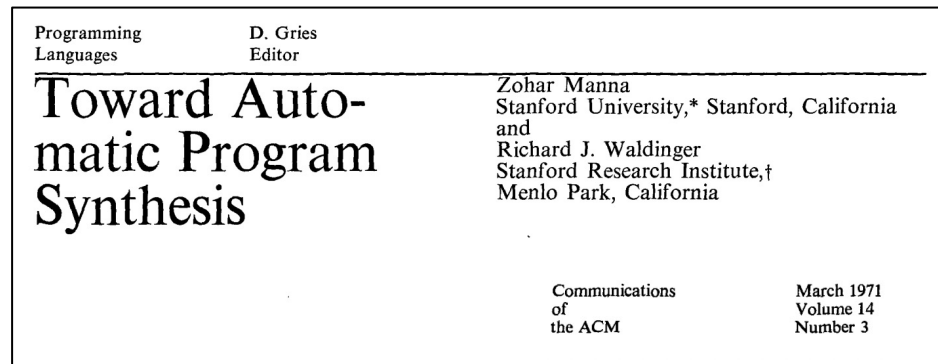
Why is Program Synthesis Important?



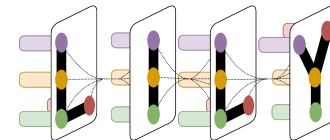
- On a higher level:
 - It is one of the ***oldest*** and ***hardest*** problems in AI and CS:

This process of constructing instruction tables should be very fascinating. There need be no real danger of it ever becoming a drudge, for any processes that are quite mechanical may be turned over to the machine itself.

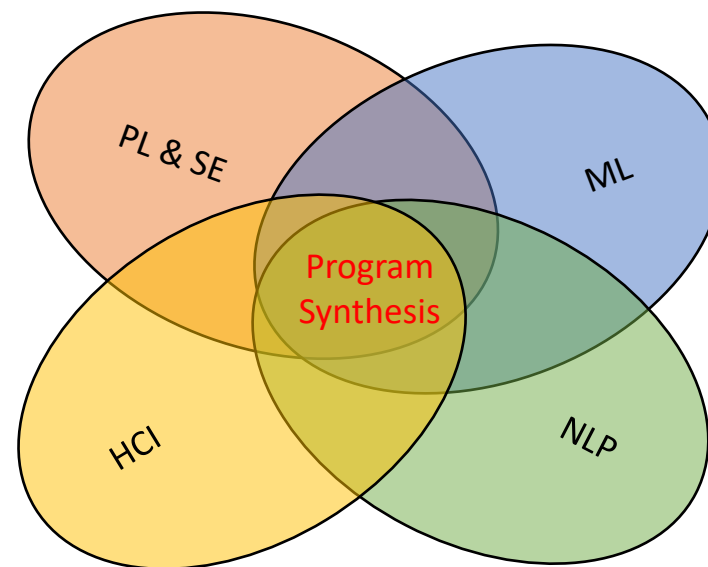
— Alan Turing (1945)



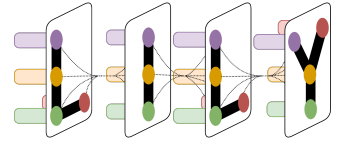
Why is Program Synthesis Important?



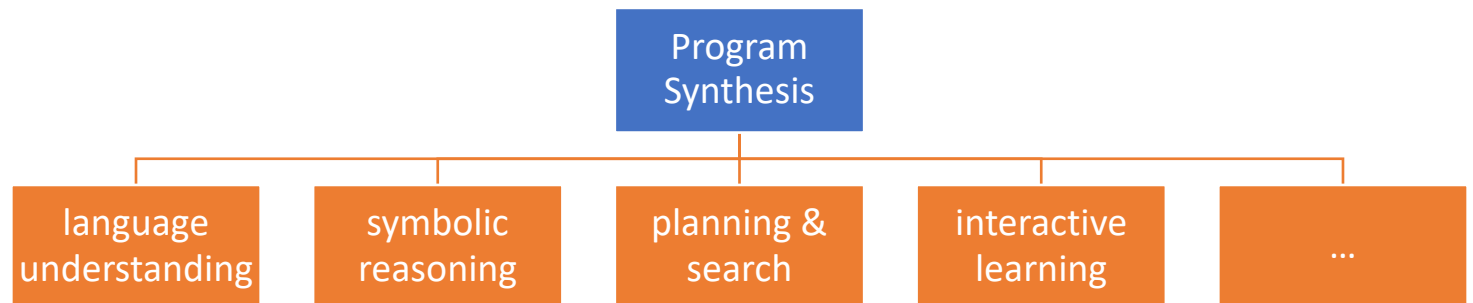
- On a higher level:
 - It is one of the *oldest* and *hardest* problems in AI and CS
 - It involves several important areas in CS
 - Programming Languages (PL)
 - Software Engineering (SE)
 - Machine Learning (ML)
 - Natural Language Processing (NLP)
 - Human-Computer Interaction (HCI)
 - ...



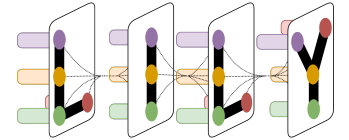
Why is Program Synthesis Important?



- On a higher level:
 - It is one of the *oldest* and *hardest* problems in AI and CS
 - It involves several important areas in CS
- It is a great testbed for intelligence:
 - language understanding
 - symbolic reasoning
 - planning & search
 - interactive learning
 - ...



Why is Program Synthesis Important?



- It empowers many real-world applications:

Full Name	Last Name	First Name
Abadi Mehe	Abadi	
Ahokainen Noora	Ahokainen	
Akkiraju, Ravi		
Aksenova, Evelyn		
Alexopoulos, Alexia		
Allu, Deevana		
Anderson, Kaari		
Andreas, Katerina		
Ankitam, Baaji		
Aunina, Kittja		
Banis, Irini		
Baric, Elvis		
Batard, Alexandre		
Berger, Dominic		
Berger, Lena		
Berger, Lucas		

Type a few examples
to show a clear pattern

FlashFill

Large
Language
Model

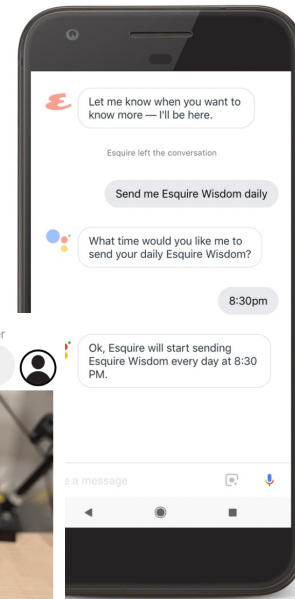
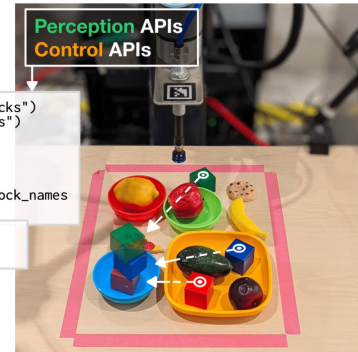
Policy Code

```
block_names = detect_objects("blocks")
bowl_names = detect_objects("bowls")
for bowl_name in bowl_names:
    if is_empty(bowl_name):
        empty_bowl = bowl_name
        break
objs_to_stack = [empty_bowl] + block_names
stack_objects(objs_to_stack)

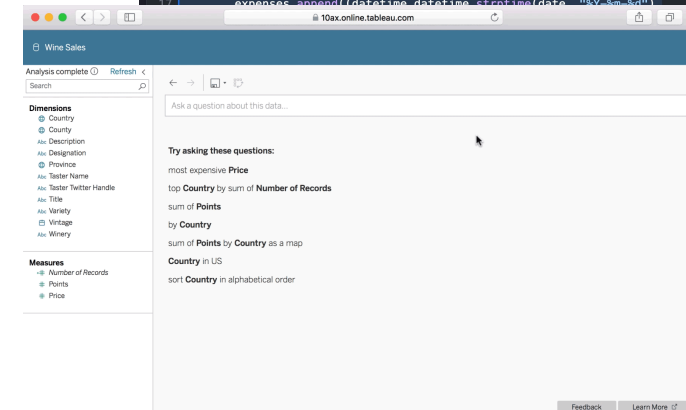
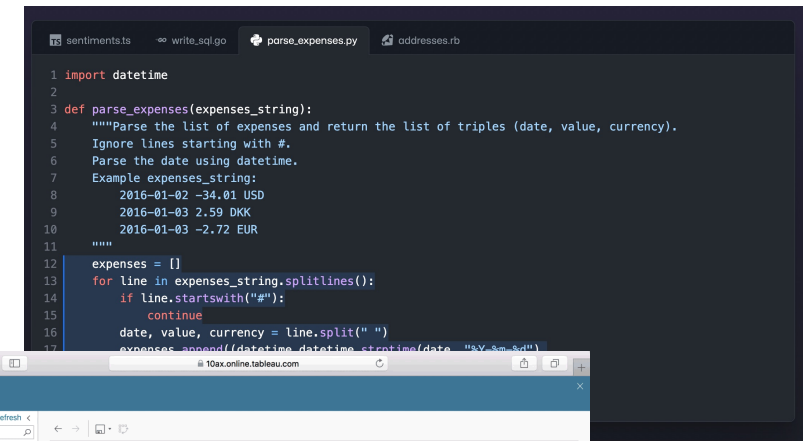
def is_empty(name):
    ...

def stack_objects(obj_names):
    n_objs = len(obj_names)
    for i in range(n_objs - 1):
        obj0 = obj_names[i + 1]
        obj1 = obj_names[i]
        pick_place(obj0, obj1)
```

Robotics Control



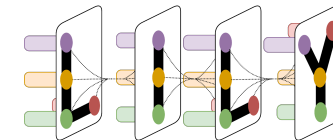
Virtual Assistants



Database Query and Visualization

Images from: <https://developers.googleblog.com/2018/03/new-creative-ways-to-build-with-actions.html>; <https://support.microsoft.com/en-us/office/save-time-with-flash-fill-9159216a-75a0-4c11-82e6-8eca29cb3b89>; <https://github.com/features/copilot>; <https://code-as-policies.github.io/>; <https://www.tableau.com/blog/ask-data-simplifying-analytics-natural-language-98655>

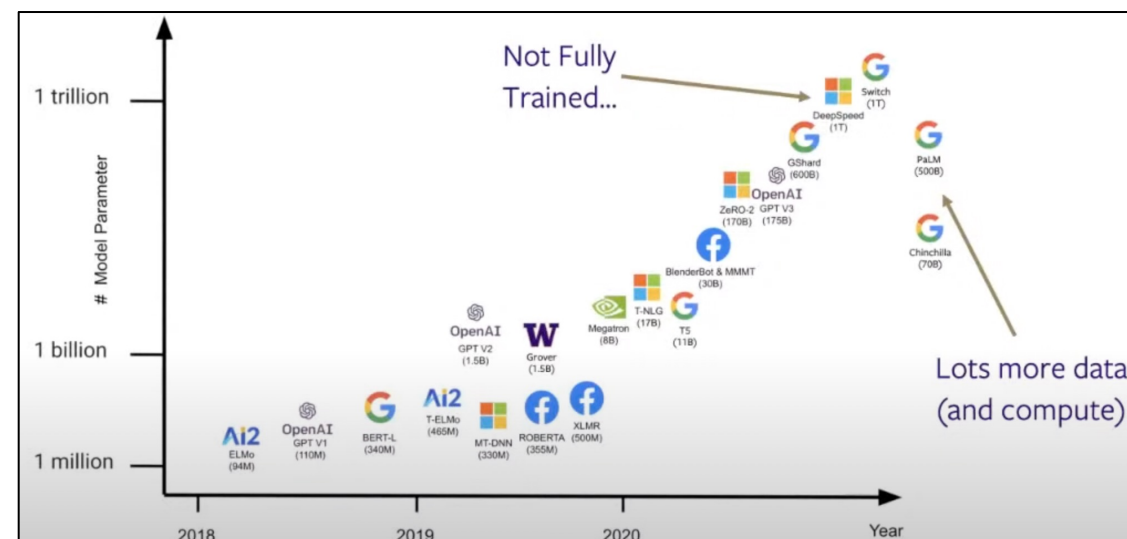
Program Synthesis with Language Models



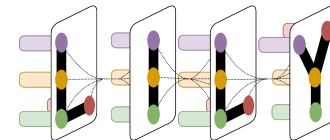
- What is a language model?
 - Predicts the next word given the context
 - Learns to maximize $P_{\theta}(x_n | x_1, x_2, \dots, x_{n-1})$ given training data — self-supervised training
- *The status quo* of pretrained language models:
 - They are getting **larger**
 - ... and **better**
 - Multi-task learning
 - Zero/Few-shot (in-context) learning
 - Instruction tuning
 - ...

$$P(X) = \prod_{i=1}^I P(x_i | x_1, \dots, x_{i-1})$$

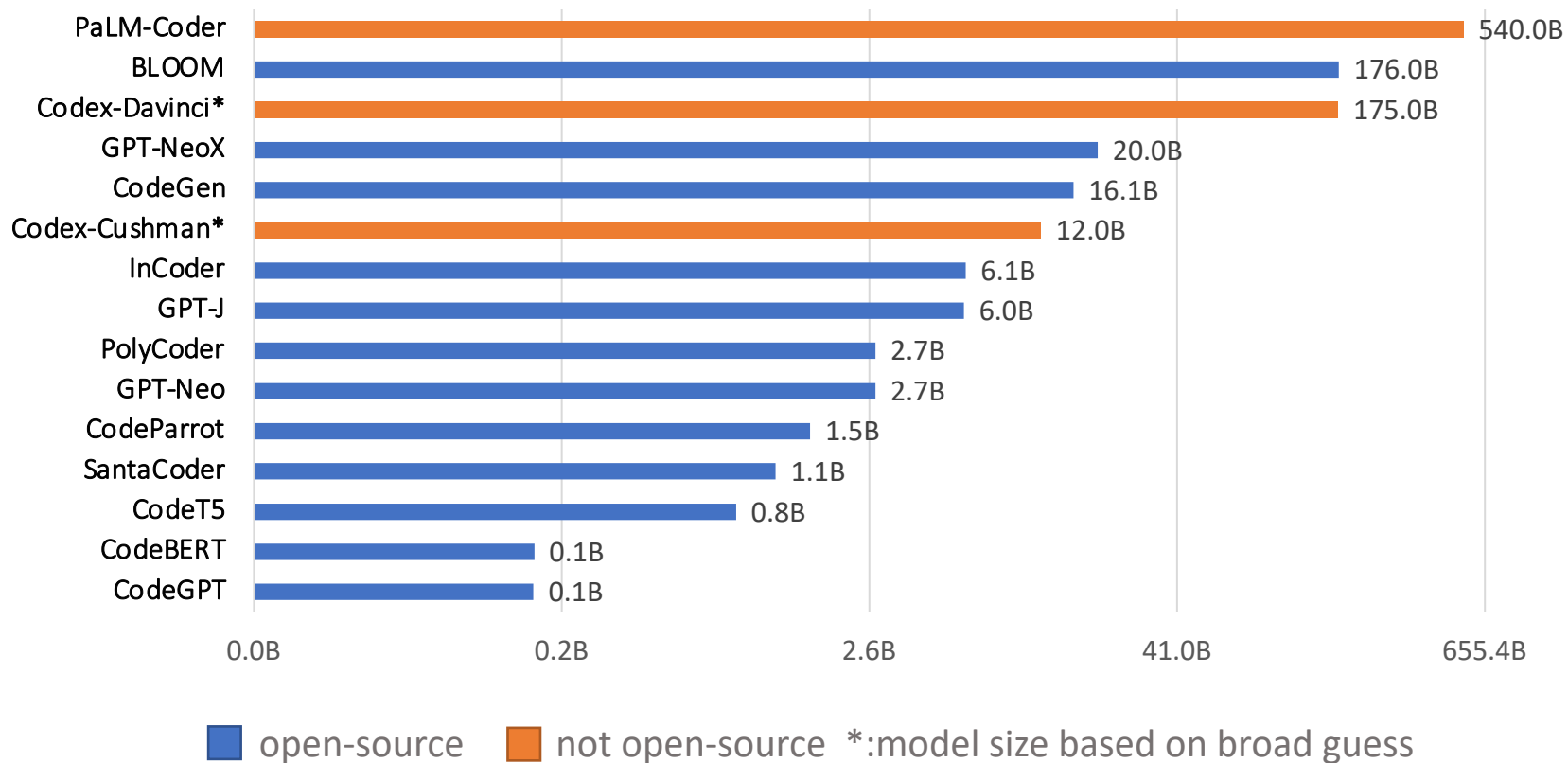
Next Word Context



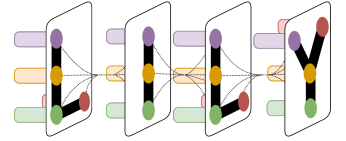
Program Synthesis with Language Models



- (Large) language models trained on code:

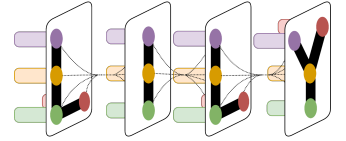


Program Synthesis with Language Models



- What does it mean for program synthesis in the “*LLM era*” ?
 - The search space can be greatly reduced/optimized by pretrained CodeLMs
- **We can do more!**
 - From domain specific languages (DSL)
 - SQL (Zhong et al., 2017; Yu et al., 2018)
 - λ -calculus (Zettlemoyer and Collins, 2005; Rabinovich et al., 2017)
 - Karel (Bunel et al., 2018; Chen et al., 2019)
 - To general-purpose programs
 - Basic Python programming (Chen et al., 2021; Austin et al., 2021)
 - Data science code generation (Lai et al., 2022; Yin et al., 2022)
 - Competition-level code generation (Hendrycks et al., 2021; Li et al., 2022)

Program Synthesis with Language Models



- What does it mean for program synthesis in the “*LLM era*” ?
 - The search space can be greatly reduced/optimized by pretrained CodeLMs
- **We can do more!**
 - From supervised learning to zero/few-shot prompting

AN

can you write a wrong implementation of `is_prime` function in python



Sure! Here's a wrong implementation of a function to check if a number is prime or not:

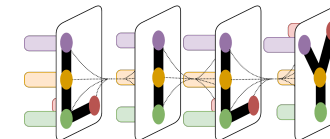


python

Copy code

```
def is_prime(number):  
    if number <= 1:  
        return False  
    for i in range(2, number):  
        if (number % i) == 0:  
            return False  
    return True
```

Program Synthesis with Language Models



- So are LLMs the solution to program synthesis?
 - Examples: programming with ChatGPT

AN can you write a wrong implementation of `is_prime` function in python

Sure! Here's a wrong implementation of a function to check if a number is prime or not:

```
python Copy code
def is_prime(number):
    if number <= 1:
        return False
    for i in range(2, number):
        if (number % i) == 0:
            return False
    return True
```

This implementation is wrong because it only checks if the number is divisible by integers less than itself, but a number can also be divisible by numbers greater than itself. For example, this function would return `True` for `4`, which is not a prime number.

A better implementation would be to check if the number is divisible by any number up to its square root.

AN can you give me a wrong implementation for determine odd/even numbers

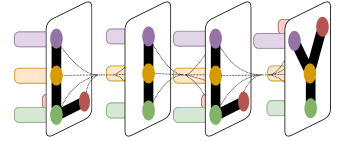
Sure, here is an incorrect implementation to determine if a number is odd or even:

```
python Copy code
def is_odd_or_even(n):
    if n % 2 == 0:
        return "even"
    else:
        return "odd"
```

This implementation only works for positive integers, but it fails for negative numbers and zero. A correct implementation to handle all cases would be:

```
python Copy code
def is_odd_or_even(n):
    if n == 0:
        return "even"
    elif n % 2 == 0:
        return "even"
    else:
        return "odd"
```

Program Synthesis with Language Models

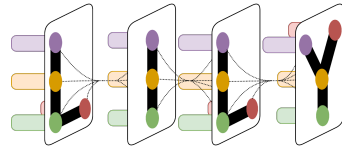


- What is happening here?
 - LLMs are only trained on the ***surface form*** of programs
 - ...yet the ***semantics*** of a program is in its ***execution***
- But the best of all...
 - Execution can be done automatically!

How to incorporate program semantics into LMs using execution to improve their performance?

Paper Presentation 1:

Learning Math Reasoning from Self-Sampled Correct and Partially-Correct Solutions



Yale LILY Lab

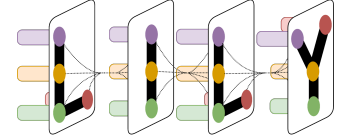


Ansong Ni, Jeevana Priya Inala, Chenglong Wang, Oleksandr Polozov,
Christopher Meek, Dragomir Radev, Jianfeng Gao

ICLR'23



Task and Motivation



- Task: finetuning pretrained LMs for generating programs from math problems described in natural language
- Motivation - ***Program Aliasing***:
 - The same specification (i.e., question) can be satisfied by different programs
 - However, the training data typically only have one reference solution for learning
 - This causes overfitting as the model keeps seeing the same solution over multiple epochs of training

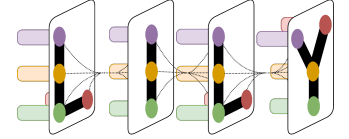
NL Input	A goods train runs at a speed of 72kmph and crosses a 250M long platform in 26 seconds. What is the length of the goods train?
----------	--

Ref. Program	<pre>n0 = 72 n1 = 250 n2 = 26 t0 = n0 * 0.2778 t1 = n1 / t0 t2 = n2 - t1 answer = t0 * t2</pre>
--------------	---

Alt. Program	<pre>n0 = 72 n1 = 250 n2 = 26 t0 = n0 * 0.2778 t1 = n2 * t0 answer = t1 - n1</pre>
--------------	--



Task and Motivation



- Task: finetuning pretrained LMs for generating programs from math problems described in natural language
- Motivation - *Program Aliasing*
- **Observation:**
 - During inference, the model can sometimes generate programs that are correct but not necessarily the gold one
 - Can we encourage this behavior during training and learning from the self-sampled program solutions?
 - **YES!**

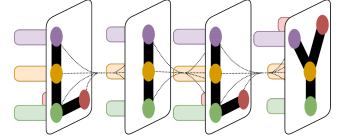
NL Input	A goods train runs at a speed of 72kmph and crosses a 250M long platform in 26 seconds. What is the length of the goods train?
----------	--

Ref. Program	<pre>n0 = 72 n1 = 250 n2 = 26 t0 = n0 * 0.2778 t1 = n1 / t0 t2 = n2 - t1 answer = t0 * t2</pre>
--------------	---

Alt. Program	<pre>n0 = 72 n1 = 250 n2 = 26 t0 = n0 * 0.2778 t1 = n2 * t0 answer = t1 - n1</pre>
--------------	--



Self-Sampling Framework with Full Correctness



- Use a buffer \mathcal{B} to save self-sampled programs
- Online sampling and filtering
 - Attempt to sample alternative correct solutions from the model during training (L4)
 - Execute the program samples (L6)
 - Filter the samples by:
 - **Full correctness:** matches the gold final execution result (L7)
 - Duplication (L8): pruning out “trivial variants”
 - Save them in the buffer for learning (L9)

Algorithm 1 Training Update

Input:

Parameterized model $P_\theta(y|x)$;

Executor $\mathcal{E} : \mathcal{Y} \rightarrow \mathcal{Z}$;

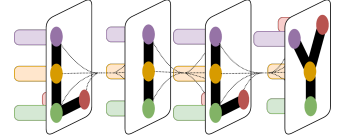
A training example (x, y^*, z^*) ;

Buffer \mathcal{B} for this input x

```
1: if  $|\mathcal{B}| = 0$  then
2:    $\mathcal{B} \leftarrow \mathcal{B} + \{y^*\}$  /* initialize buffer */
3: end if
4:  $\hat{Y} \leftarrow \text{SampleSolutions}(x, P_\theta, \mathcal{B})$ 
5: for  $\hat{y}$  in  $\hat{Y}$  do
6:    $\hat{z} \leftarrow \mathcal{E}(\hat{y})$  /* execute solution */
7:   if  $\text{isCorrect}(\hat{z}, z^*)$  then
8:     if not  $\text{isDuplicate}(\hat{y}, \mathcal{B})$  then
9:        $\mathcal{B} \leftarrow \mathcal{B} + \hat{y}$  /* save to buffer */
10:    end if
11:  end if
12: end for
13:  $\theta \xleftarrow{\text{update}} \nabla_\theta \mathcal{L}(x, \mathcal{B}, P_\theta)$ 
```



Self-Sampling Framework with Full Correctness



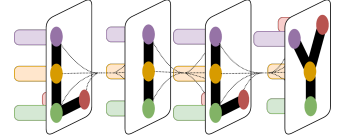
- Objectives for learning from multiple targets
 - **MLE**: maximize the likelihood of generating the reference program;
 - **MLE-Aug**: simply summing the loss from the saved correct programs, it encourages the model to put equal weights on all targets;
 - **MML**: maximize the marginal likelihood of all saved correct solutions, but note that the gradient is in proportion to the likelihood;
 - **β -MML** (Guu et al., 2017): an interpolation between MML and MLE-Aug, with $\beta \in (0, 1]$

Name	Loss Functions $\mathcal{L}(x, \mathcal{B}, P_\theta)$	Gradients $\nabla_\theta(x, \mathcal{B}, P_\theta)$
MLE	$-\log P_\theta(y^* x)$	$-\nabla_\theta \log P_\theta(y^* x)$
MLE-Aug	$-\sum_{\hat{y} \in \mathcal{B}} \log P_\theta(\hat{y} x)$	$-\sum_{\hat{y} \in \mathcal{B}} \nabla_\theta \log P_\theta(\hat{y} x)$
MML	$-\log \sum_{\hat{y} \in \mathcal{B}} P_\theta(\hat{y} x)$	$-\sum_{\hat{y} \in \mathcal{B}} \frac{P_\theta(\hat{y} x)}{\sum_{\tilde{y} \in \mathcal{B}} P_\theta(\tilde{y} x)} \nabla_\theta \log P_\theta(\hat{y} x)$
β -MML	$-\frac{1}{\beta} \log \sum_y P_\theta(\hat{y} x)^\beta$	$-\sum_{\hat{y} \in \mathcal{B}} \frac{P_\theta(\hat{y} x)^\beta}{\sum_{\tilde{y} \in \mathcal{B}} P_\theta(\tilde{y} x)^\beta} \nabla_\theta \log P_\theta(\hat{y} x)$

Different loss functions and gradients used for learning from multiple targets



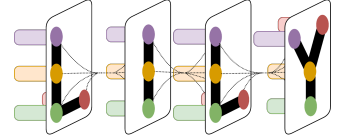
Learning from Partially-Correct Solutions (PCSs)



- Learning from self-sampled correct solutions are great, but...
 - It is also hard to sample, especially for complex programs, it is hard to “creatively” write a different program that is also correct.
 - There are many failed attempts for the model to be creative, and they were almost there!
 - Can we learn from those programs as well?
 - **YES!**



Learning from Partially-Correct Solutions (PCSs)



- **Intermediate state s_i** is the set of all variables values in the scope after executing the first i steps as the **program prefix $y_{\leq i}$**
 - Note: the state representation is name-agnostic since variable names do not typically contributes to the semantics of the solutions

A goods train runs at a speed of 72kmph and crosses a 250M long platform in 26 seconds. What is the length of the goods train?

```
n0 = 72
n1 = 250
n2 = 26
t0 = n0 * 0.2778
t1 = n1 / t0
t2 = n2 - t1
answer = t0 * t2
```

Solutions

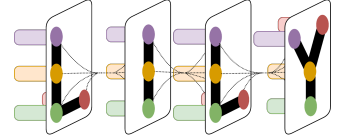
Execute

```
{72}
{72, 250}
{72, 250, 26}
{72, 250, 26, 20.0}
{72, 250, 26, 20.0, 12.5}
{72, 250, 26, 20.0, 12.5, 13.5}
output = 270
```

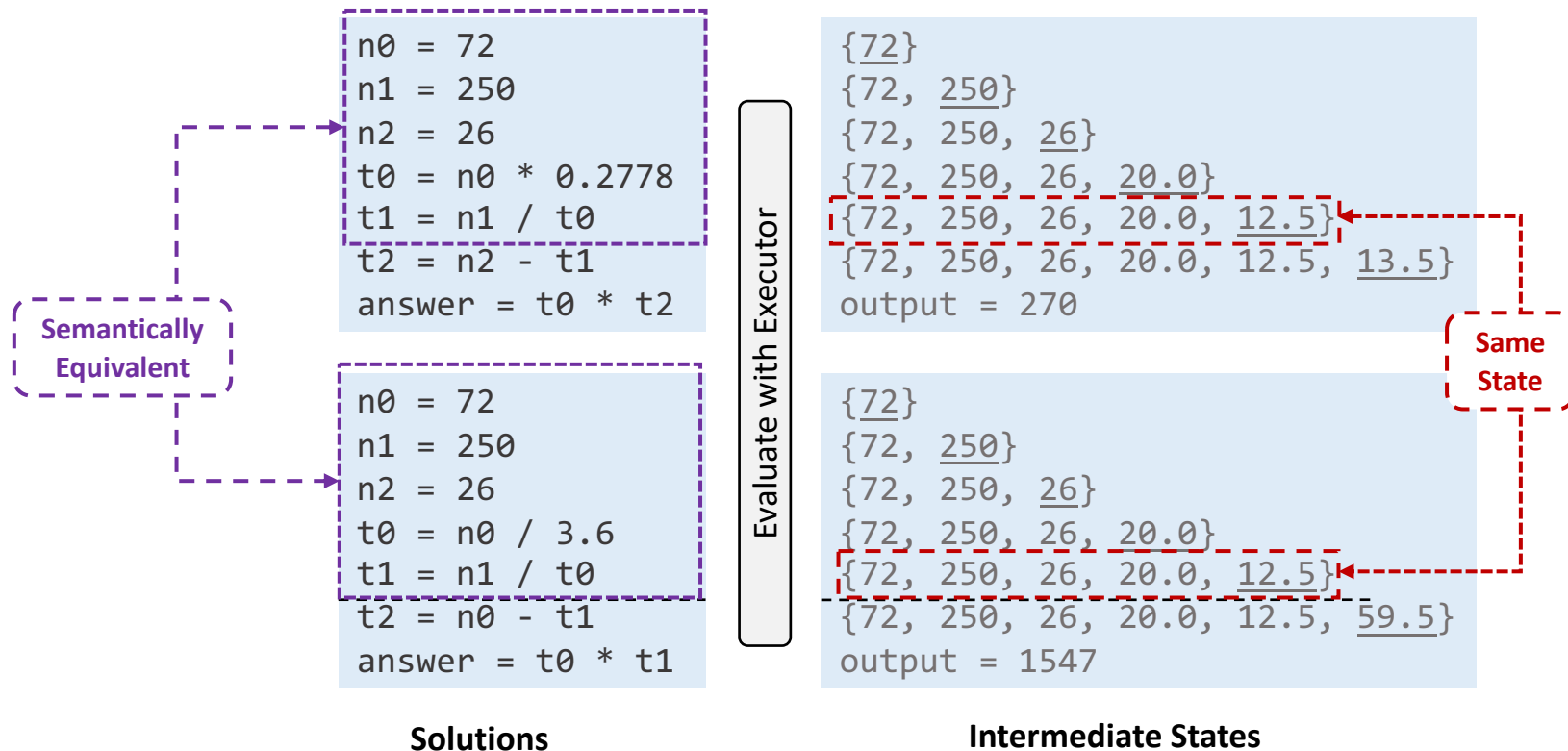
Intermediate States



Learning from Partially-Correct Solutions (PCSs)

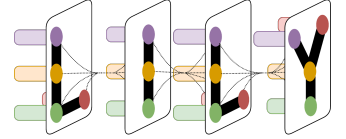


- Prefixes of two programs $y_{\leq i}$ and $y'_{\leq j}$ are ***semantically equivalent*** if and only if $s_i = s'_j$
 - i.e., those two program prefixes produces the exact same set of variable values



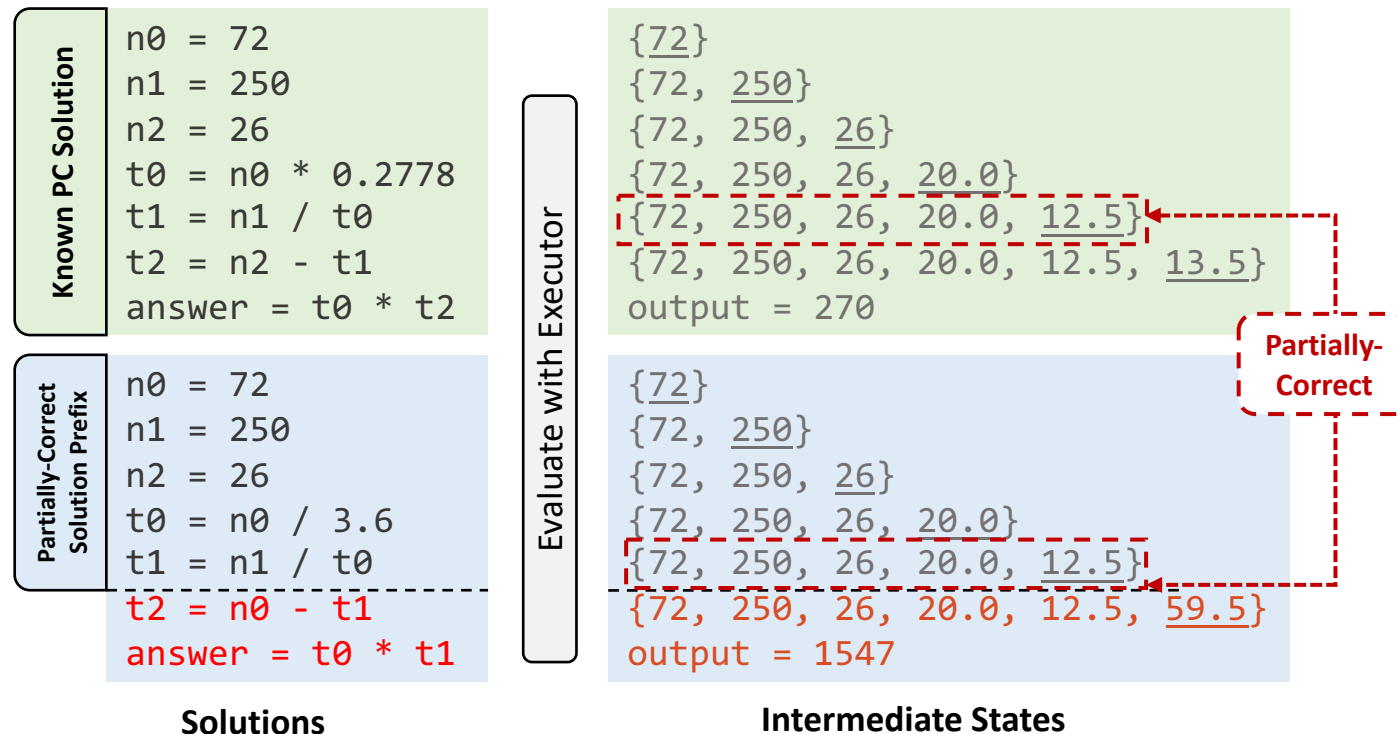


Learning from Partially-Correct Solutions (PCSs)



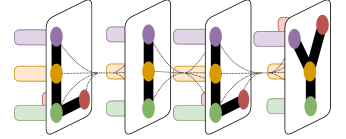
- A program prefix $y_{\leq i}$ is **partially-correct** if and only if it is semantically equivalent to the prefix of a *known partially-correct solution* $y_{\leq j}^*$
 - Since we keep all known partially-correct solutions in buffer \mathcal{B} , we have

$$\text{PartiallyCorrect}(y_{\leq i}) \iff \exists y^* \in \mathcal{B}. \exists j \leq |y^*| \text{ s.t. } s_j^* = s_i$$





Learning from Partially-Correct Solutions (PCSs)



- Modification to the main algorithm
 - Guided-sampling from known PCS prefix

Algorithm 2 *SampleSolutions*(x, P_θ, \mathcal{B}) with partially-correct solutions

Input: Model $P_\theta(y|x)$; the NL input x and a set of partially-correct solutions \mathcal{B}

Output: Solution samples \hat{Y} .

- 1: Select $\hat{y}_{\leq i} \in \mathcal{B} \setminus \{\hat{y} | \mathcal{E}(\hat{y}) = z^*\}$ uniformly at random */* sample PCS prefix for completion */*
 - 2: Sample a set of completions $Y_p \sim P_\theta(\hat{y}_{>i} | \hat{y}_{\leq i}, x)$
 - 3: $\hat{Y} \leftarrow \{[\hat{y}_{\leq i} || \hat{y}_{>i}]\}_{\hat{y}_{>i} \in Y_p}$ */* concatenate completions with the solution prefix */*
 - 4: **return** \hat{Y}
-

- Identify partially-correct program prefixes
- Filtering solution prefixes
 - PCS is only saved if it is not a prefix of any know PCS
- Learning objectives
 - With an auto-regressive generation model, the learning of $P(y_{\leq i} | x)$ is independent of $y_{>i}$, thus no change to the learning objectives are required.

Algorithm 1 Training Update

Input:

Parameterized model $P_\theta(y|x)$;

Executor $\mathcal{E} : \mathcal{Y} \rightarrow \mathcal{Z}$;

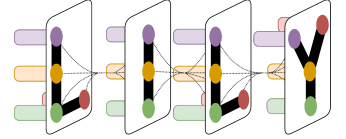
A training example (x, y^*, z^*) ;

Buffer \mathcal{B} for this input x

- 1: **if** $|\mathcal{B}| = 0$ **then**
 - 2: $\mathcal{B} \leftarrow \mathcal{B} + \{y^*\}$ */* initialize buffer */*
 - 3: **end if**
 - 4: $\hat{Y} \leftarrow \text{SampleSolutions}(x, P_\theta, \mathcal{B})$
 - 5: **for** \hat{y} **in** \hat{Y} **do**
 - 6: $\hat{z} \leftarrow \mathcal{E}(\hat{y})$ */* execute solution */*
 - 7: **if** *isCorrect*(\hat{z}, z^*) **then**
 - 8: **if not** *isDuplicate*(\hat{y}, \mathcal{B}) **then**
 - 9: $\mathcal{B} \leftarrow \mathcal{B} + \hat{y}$ */* save to buffer */*
 - 10: **end if**
 - 11: **end if**
 - 12: **end for**
 - 13: $\theta \xleftarrow{\text{update}} \nabla_\theta \mathcal{L}(x, \mathcal{B}, P_\theta)$
-



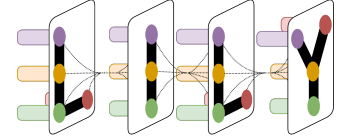
Experimental Setup



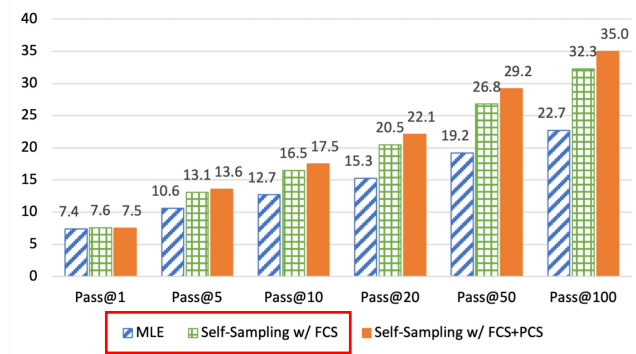
- Datasets:
 - **MathQA-Python-Filtered**: we do template-based deduplication is applied to the original MathQA-Python dataset to better evaluation generalization
 - **GSM5.5K-Python**: we automatically converted the natural language formulas to program solutions in the same style as MathQA-Python
- Language model:
 - We use **GPT-Neo 125M and 2.7B** as our main LM to study
- Evaluation metrics:
 - We use **pass@k** following recent work in math reasoning and program synthesis



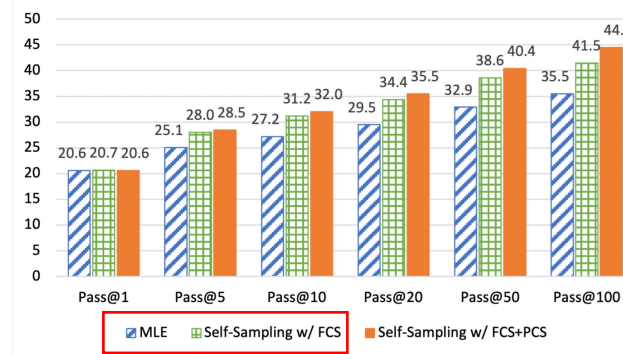
Main Results



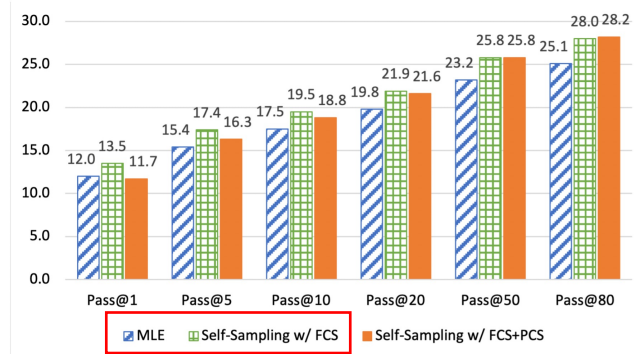
- Learning from self-sampled solutions improves pass@k



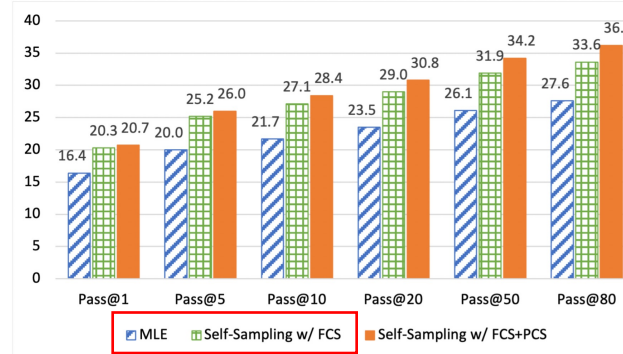
(a) GSM5.5K-Python with GPT-Neo 125M



(b) GSM5.5K-Python with GPT-Neo 2.7B



(c) MathQA-Python-Filtered with GPT-Neo 125M

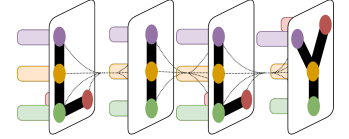


(d) MathQA-Python-Filtered with GPT-Neo 2.7B

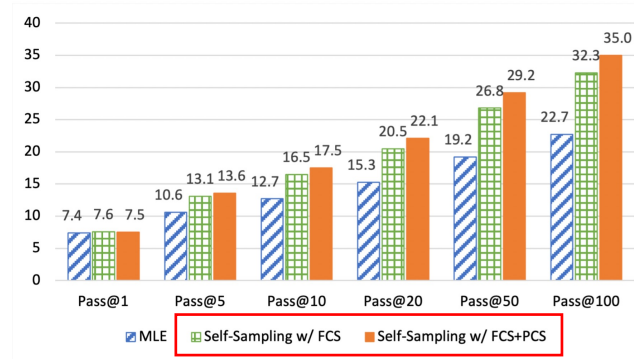
Comparing self-sampling with MLE baseline



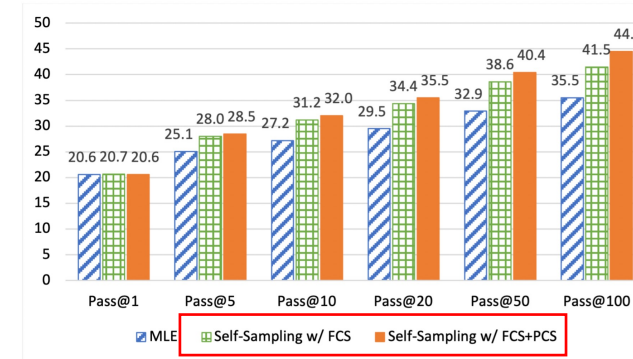
Main Results



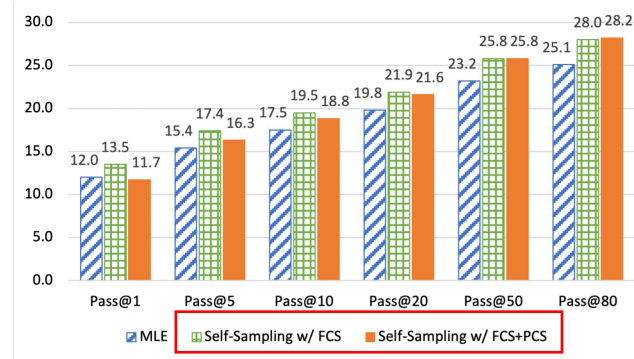
- Partially-correct solutions further improve model performance



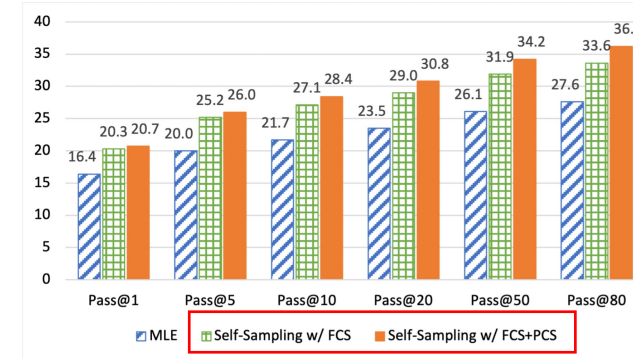
(a) GSM5.5K-Python with GPT-Neo 125M



(b) GSM5.5K-Python with GPT-Neo 2.7B



(c) MathQA-Python-Filtered with GPT-Neo 125M

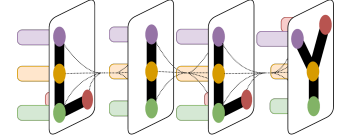


(d) MathQA-Python-Filtered with GPT-Neo 2.7B

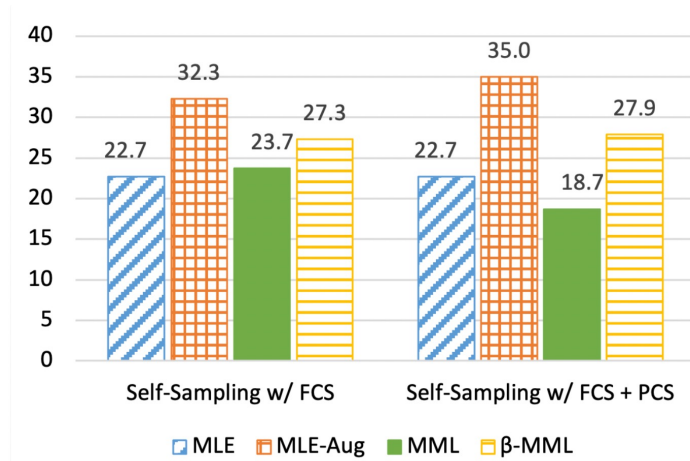
Comparing self-sampling with MLE baseline



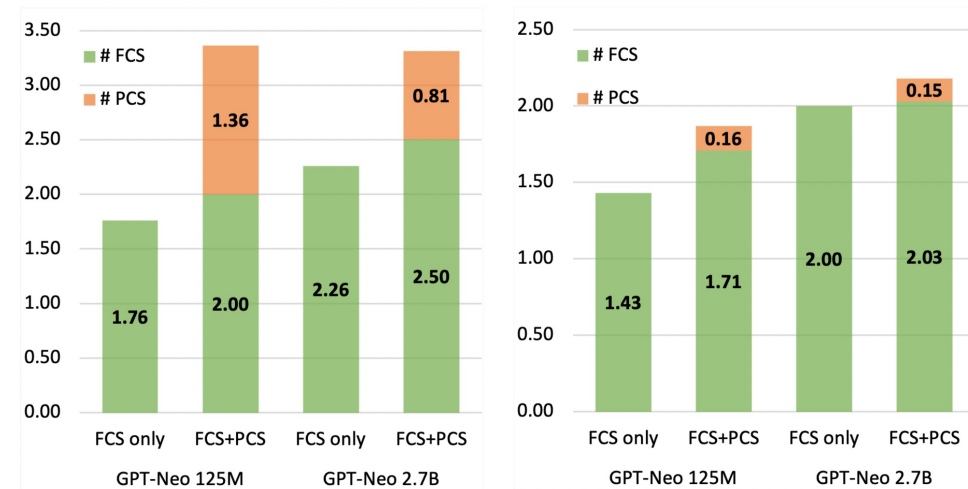
Ablation Studies and Analysis



- MLE-Aug loss function works the best
 - It draws learning signal equally from all saved solutions
 - MML works the worse, especially when also learning from PCSs
- Dynamics between # of PCSs and FCSs saved in the buffer
 - More saved solutions typically results in better pass@k performance
 - Large models are better at completing PCS prefixes to be FCS



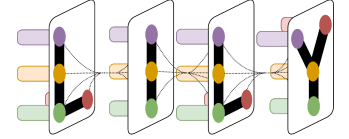
GSM5.5K-Python with finetuned
GPT-Neo 125M model



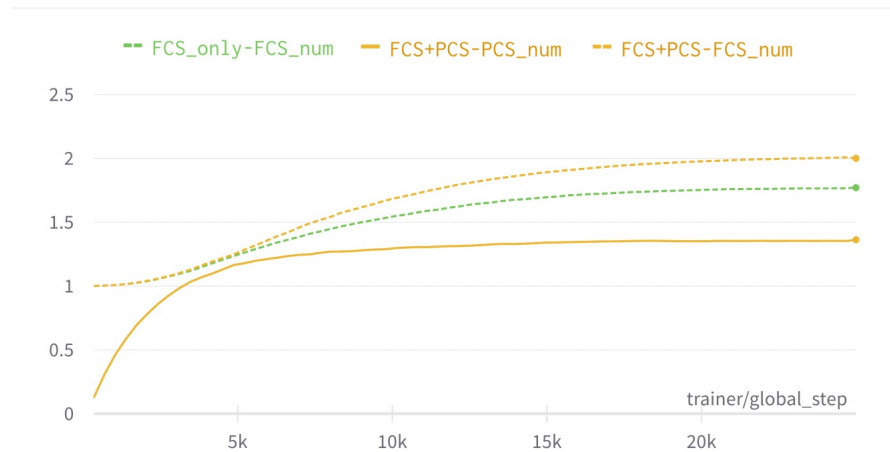
saved FCSs and PCSs per problem for GSM5.5K-Python (left)
and MathQA-Python-Filtered (right)



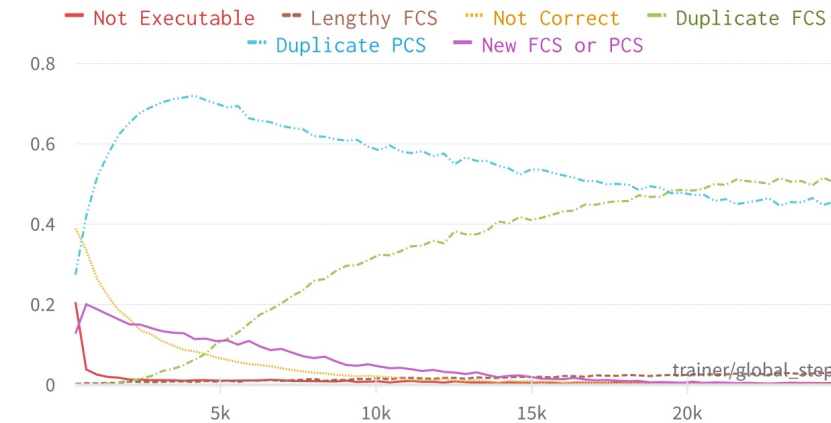
Ablation Studies and Analysis



- Dynamics between # of PCSs and FCSs saved in the buffer
 - Partially-correct solutions helps learning especially in early stages



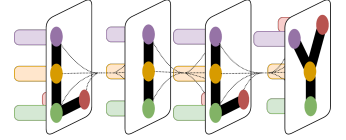
Growth of the number of saved FCS and PCS during training



Distribution of the characterization of self-sampled solutions during training



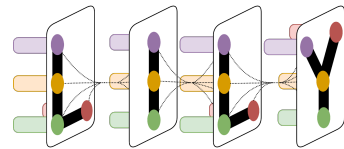
Takeaways



- Learning from self-sampled solutions can be useful given the right constraints
 - E.g., when you can easily prune out incorrect and duplicated ones
- Programs are not either correct or wrong, they can be partially-correct.
 - Note that our definition of partial correctness is different from say, passing 60% of the test cases, because that program would still be wrong;
 - Instead, by comparing execution traces, we identify the first 60% of the program is on the right track

Paper Presentation 2:

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



Yale LILY Lab

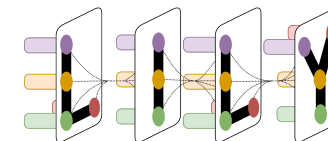


Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih,
Sida I. Wang*, Xi Victoria Lin*

arxiv'23, preprint



Task: Language-to-Code Generation



- Task: **language-to-code generation** using LLMs in few-shot learning
 - Cornerstone for many tasks in NLP and ML

Complex question What are the name and budget of the departments with average instructor salary greater than the overall average?

Complex SQL

```
SELECT T2.name, T2.budget
FROM instructor as T1 JOIN department as
T2 ON T1.department_id = T2.id
GROUP BY T1.department_id
HAVING avg(T1.salary) >
(SELECT avg(salary) FROM instructor)
```

Spider (Yu et al., 2018)

Problem: Beth bakes 4, 2 dozen batches of cookies in a week. If these cookies are shared amongst 16 people equally, how many cookies does each person consume?

Solution: Beth bakes 4 2 dozen batches of cookies for a total of $4 \times 2 = \ll 4 \times 2 = 8 \gg$ 8 dozen cookies
There are 12 cookies in a dozen and she makes 8 dozen cookies for a total of $12 \times 8 = \ll 12 \times 8 = 96 \gg$ 96 cookies
She splits the 96 cookies equally amongst 16 people so they each eat $96 / 16 = \ll 96 / 16 = 6 \gg$ 6 cookies
Final Answer: 6

Problem: Mrs. Lim milks her cows twice a day. Yesterday morning, she got 68 gallons of milk and in the evening, she got 82 gallons. This morning, she got 18 gallons fewer than she had yesterday morning. After selling some gallons of milk in the afternoon, Mrs. Lim has only 24 gallons left. How much was her revenue for the milk if each gallon costs \$3.50?

Mrs. Lim got 68 gallons - 18 gallons = $\ll 68 - 18 = 50 \gg$ 50 gallons this morning.
So she was able to get a total of 68 gallons + 82 gallons + 50 gallons = $\ll 68 + 82 + 50 = 200 \gg$ 200 gallons.
She was able to sell 200 gallons - 24 gallons = $\ll 200 - 24 = 176 \gg$ 176 gallons.
Thus, her total revenue for the milk is $\$3.50 / \text{gallon} \times 176 \text{ gallons} = \$ \ll 3.50 \times 176 = 616 \gg$ 616.
Final Answer: 616

GSM8k (Cobbe et al., 2021)

Year	City	Country	Nations
1896	Athens	Greece	14
1900	Paris	France	24
1904	St. Louis	USA	12
...
2004	Athens	Greece	201
2008	Beijing	China	204
2012	London	UK	204

x = Greece held its last Summer Olympics in which year?

y = 2004

WikiTableQuestions (Pasupat and Liang., 2015)

prompt

Write a python function to check if a given number is one less than twice its reverse. Your code should satisfy these tests:

```
assert check(70) == False
assert check(23) == False
assert check(73) == True
```

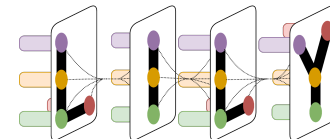
model

```
def check(n):
    if n == 2*int(str(n)[::-1])-1:
        return True
    else:
        return False
```

MBPP (Austin et al., 2021)



Motivation



- Task: natural language to code generation using large language models without parameter updates (i.e., finetuning)
- Motivation:
 - The CodeLMs are trained on surface code, how do we incorporate execution semantics into the generation process?
 - The cost for finetuning LLMs are huge, how do we improve them without changing their parameters?

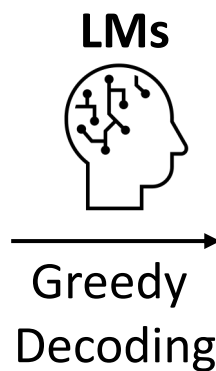
Few-shot Exemplars

```
-- Translate natural language  
question into SQL Query  
  
-- Example  
-- NL: What ...  
SELECT ...
```

+

Task Input

```
-- Example  
--NL: How many students in the  
class are between 20 and 30  
years old?
```

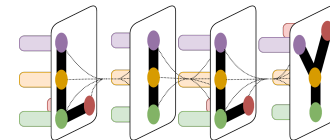


Prog.

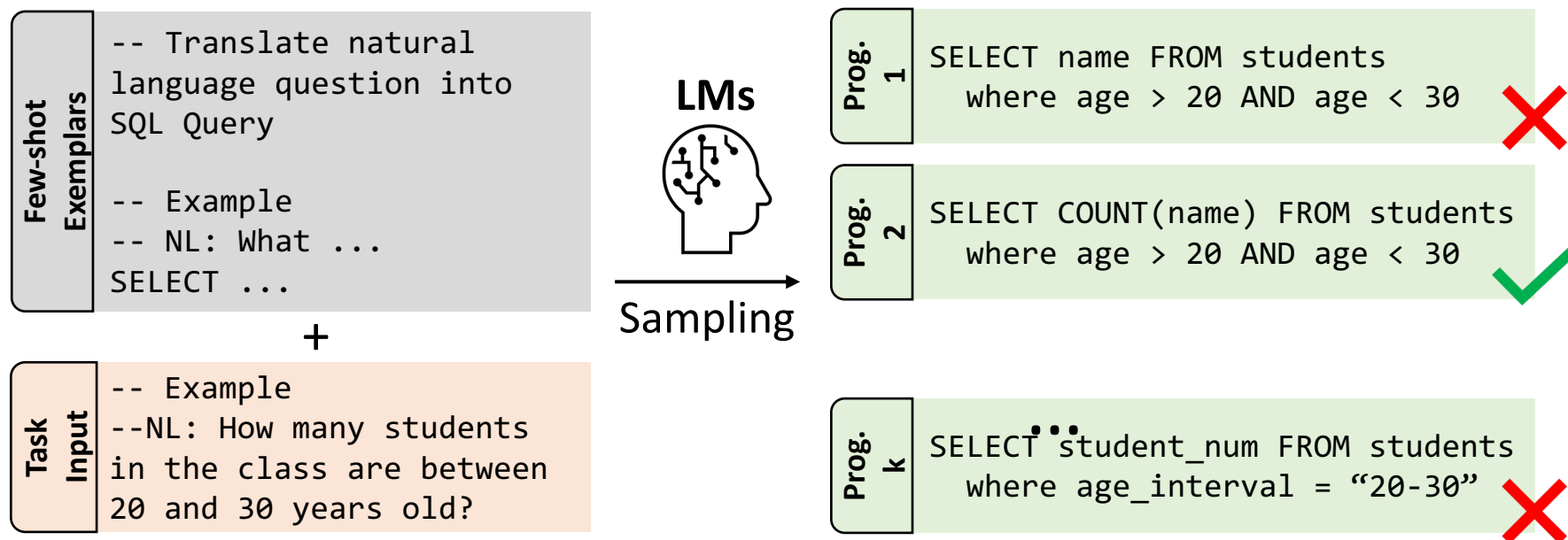
```
SELECT name FROM students  
where age > 20 AND age < 30
```



Motivation

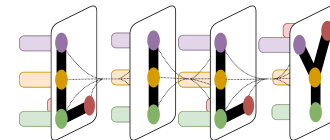


- Task: natural language to code generation using large language models without parameter updates (i.e., finetuning)
- Observation:
 - While CodeLMs struggles with precision in the few-shot setting, it often produces the correct output when enough samples are drawn.

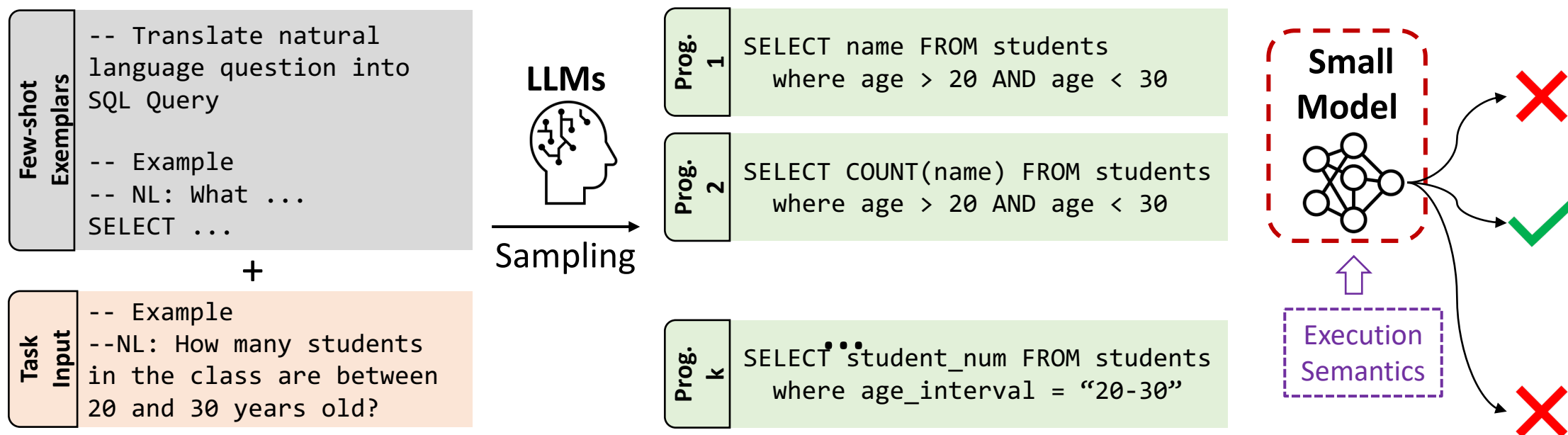




Motivation

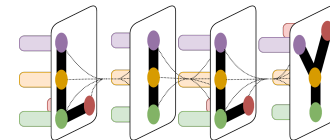


- If we can not directly finetune the LLMs...
 - Can we train *a separate (relatively) small model* as an “add-on”, to rerank the programs samples from LLMs?
 - Can we incorporate execution semantics in this model instead?

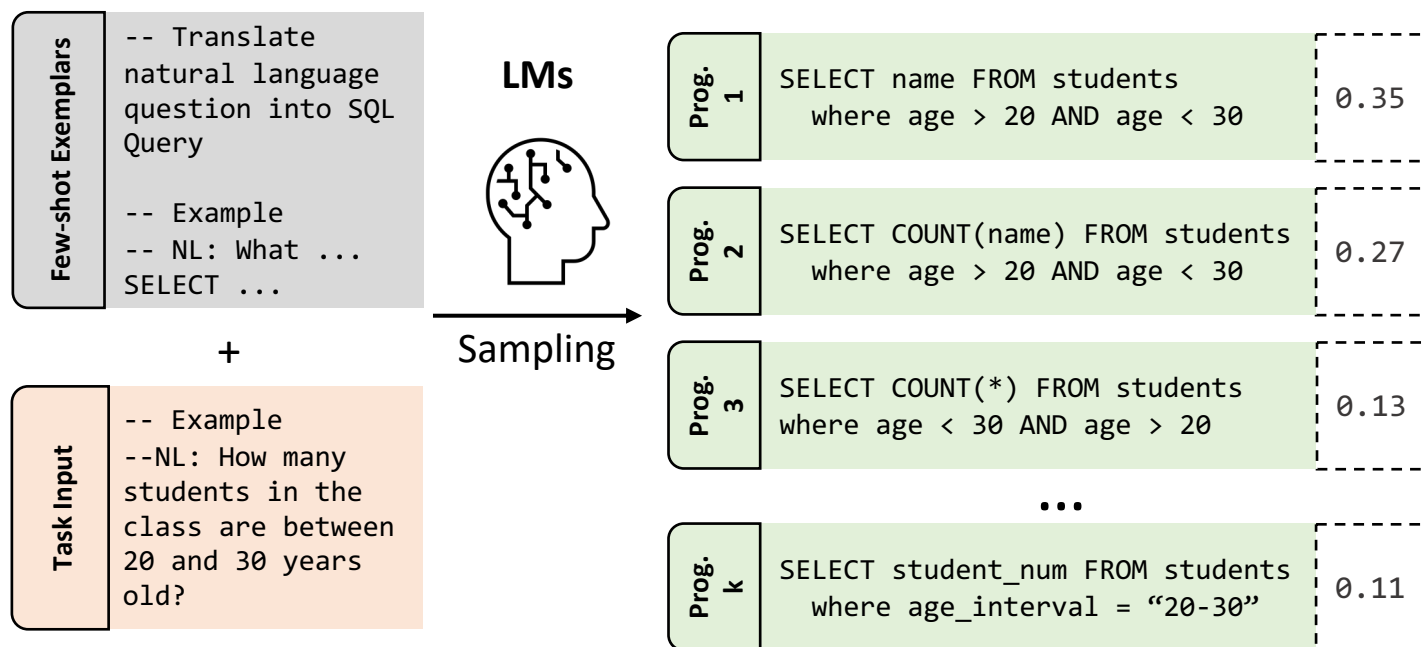




LEVER : An Overview

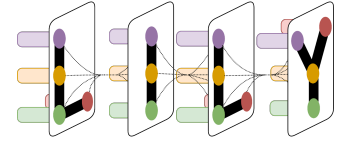


- We propose LEVER, which learns to verify language-to-code generation by LLMs trained code (CodeLMs), with the help of execution.
- LEVER has three main steps: **1) Generation**

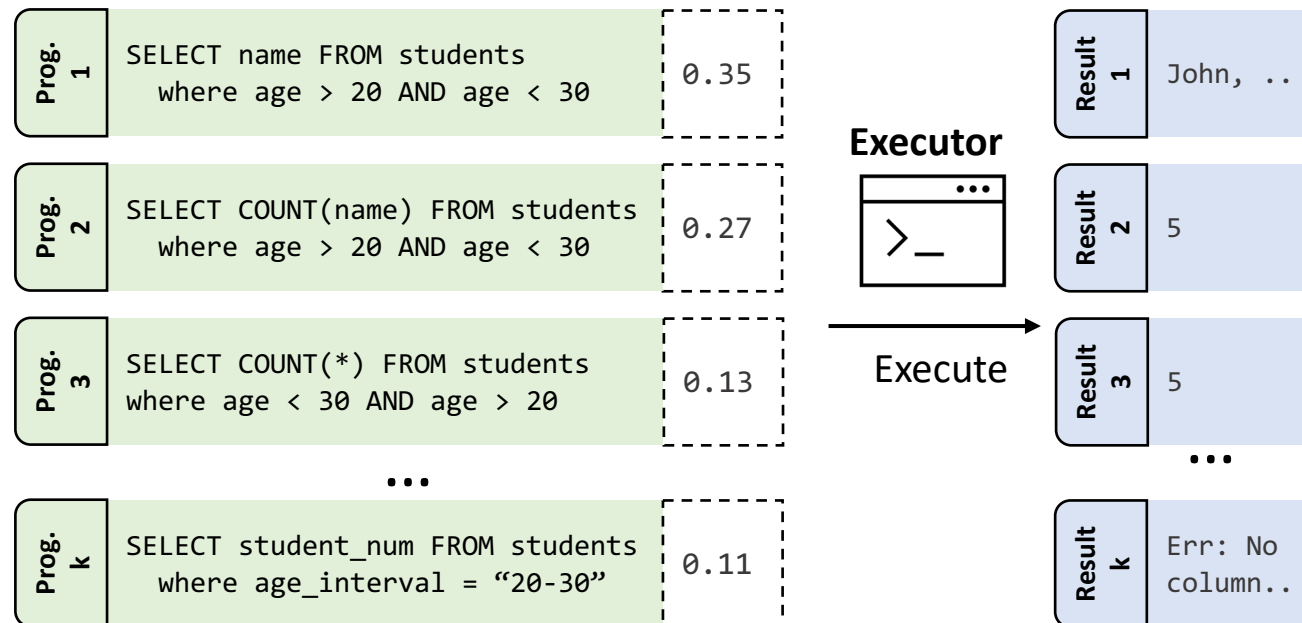




LEVER : An Overview

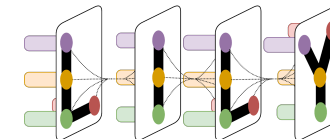


- We propose LEVER, which learns to verify language-to-code generation by LLMs trained code (CodeLMs), with the help of execution.
- LEVER has three main steps: 1) Generation; **2) Execution**

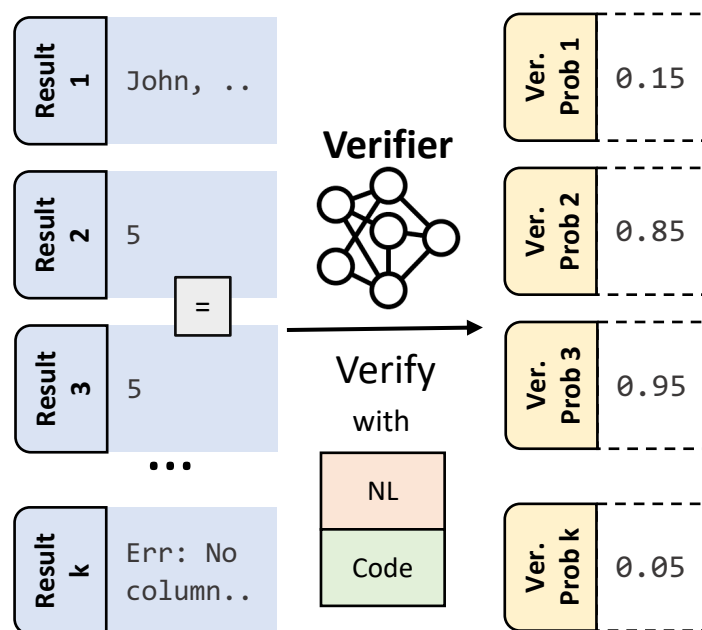




LEVER : An Overview

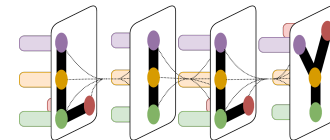


- We propose LEVER, which learns to verify language-to-code generation by LLMs trained code (CodeLMs), with the help of execution.
- LEVER has three main steps: 1) Generation; 2) Execution; **3) Verification**





LEVER : Detailed Formulation



- Detailed formulation:
 - We parameterize the verifier as a **binary classifier**, with the input as:
 - natural language x ; program sample \hat{y} ; and its execution result $\mathcal{E}(\hat{y})$

$$P_{\theta}(v_{=1}|x, \hat{y}, \mathcal{E}(\hat{y}))$$

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

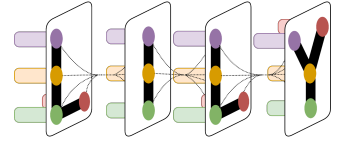
$$P_R(\hat{y}, v_{=1}|x) = P_{\text{LM}}(\hat{y}|x) \cdot P_{\theta}(v_{=1}|x, \hat{y}, \mathcal{E}(\hat{y}))$$

- We further aggregate the reranking probability of the programs in the samples that executes to the same result, and obtains **the final score**

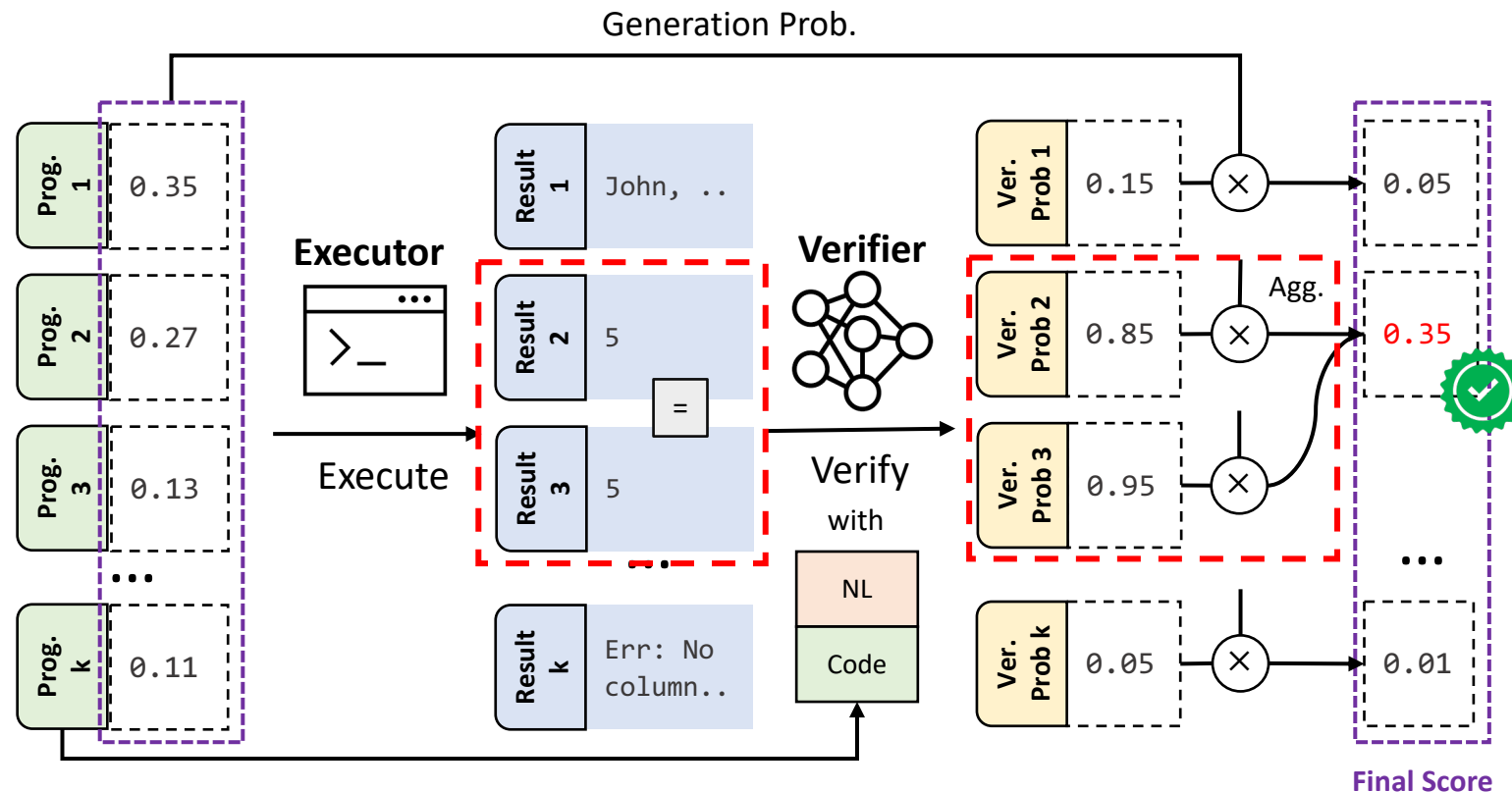
$$R(x, \hat{y}) = P_R(\mathcal{E}(\hat{y}), v_{=1}|x) = \sum_{y \in S, \mathcal{E}(y) = \mathcal{E}(\hat{y})} P_R(y, v_{=1}|x)$$



LEVER : Detailed Formulation

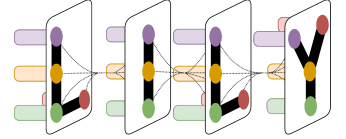


- Detailed formulation:
 - We further aggregate the reranking probability of the programs in the samples that executes to the same result, and obtains **the final score**





Learning of LEVER

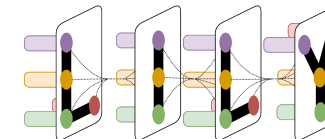


- Training data creation - for each input x :
 - Sample program candidates on the training set examples $\hat{y}_i \sim P_{LM}(y|x)$
 - Execute the programs to obtain their execution results $\hat{z}_i = \mathcal{E}(\hat{y}_i)$
 - Use gold exec. result or test cases to automatically label their correctness v_i
 - We created a set of training examples $\{x, \hat{y}_i, \hat{z}_i, v_i\}_{i=1}^n$ for each input x
- Learning objective:
 - Negative log-likelihood, 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:

- SQL** {
 - Spider (Yu et al., 2018): text-to-SQL semantic parsing;
 - WikiTQ (Pasupat et al., 2015): table question answering
- Python** {
 - GSM8k (Cobbe et al., 2021): math word problems
 - MBPP (Austin et al., 2021): basic python programming

Complex question What are the name and budget of the departments with average instructor salary greater than the overall average?

Complex SQL

```
SELECT T2.name, T2.budget
FROM instructor as T1 JOIN department as
T2 ON T1.department_id = T2.id
GROUP BY T1.department_id
HAVING avg(T1.salary) >
      (SELECT avg(salary) FROM instructor)
```

Spider

Year	City	Country	Nations
1896	Athens	Greece	14
1900	Paris	France	24
1904	St. Louis	USA	12
...
2004	Athens	Greece	201
2008	Beijing	China	204
2012	London	UK	204

x = Greece held its last Summer Olympics in which year?

y = 2004

WikiTQ

Problem: Beth bakes 4, 2 dozen batches of cookies in a week. If these cookies are shared amongst 16 people equally, how many cookies does each person consume?

Solution: Beth bakes 4 2 dozen batches of cookies for a total of $4 \times 2 = 8$ dozen cookies
There are 12 cookies in a dozen and she makes 8 dozen cookies for a total of $12 \times 8 = 96$ cookies
She splits the 96 cookies equally amongst 16 people so they each eat $96 / 16 = 6$ cookies
Final Answer: 6

Problem: Mrs. Lim milks her cows twice a day. Yesterday morning, she got 68 gallons of milk and in the evening, she got 82 gallons. This morning, she got 18 gallons fewer than she had yesterday morning. After selling some gallons of milk in the afternoon, Mrs. Lim has only 24 gallons left. How much was her revenue for the milk if each gallon costs \$3.50?

Mrs. Lim got 68 gallons - 18 gallons = 50 gallons this morning.
So she was able to get a total of 68 gallons + 82 gallons + 50 gallons = 200 gallons.
She was able to sell 200 gallons - 24 gallons = 176 gallons.
Thus, her total revenue for the milk is $\$3.50/\text{gallon} \times 176 \text{ gallons} = \616 .
Final Answer: 616

GSM8k

prompt Write a python function to check if a given number is one less than twice its reverse. Your code should satisfy these tests:

```
assert check(70) == False
assert check(23) == False
assert check(73) == True
```

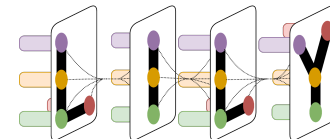
model

```
def check(n):
    if n == 2*int(str(n)[::-1])-1:
        return True
    else:
        return False
```

MBPP



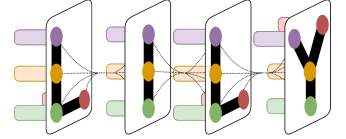
Experimental Setup



- Datasets: 4 language-to-code benchmarks from different domains
- CodeLMs:
 - Codex-davinci-002: best CodeLM available, accessible through API
 - InCoder-6B: open-source
 - CodeGen-16B: open-source
- Evaluation metric
 - Execution Accuracy (i.e., pass@1)
- Baselines:
 - Greedy: choose most likely token per decoding step
 - Maximum Likelihood (ML): choose the program with highest generation prob.
 - Error Pruning + ML (EP+ML): first prune out the programs with execution error
 - EP + Voting: majority vote of the error-free execution results



Codex + LEVER vs. Previous SoTA Methods



- Codex + LEVER achieves new SoTA results on all 4 benchmarks
 - It outperforms all previous finetuning and few-shot learning results

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\pm0.1

Spider

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 \pm 0.2	65.8\pm0.2

WikiTQ

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\pm0.2	84.5\pm0.3

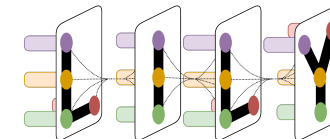
GSM8k

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\pm0.7	68.9\pm0.4

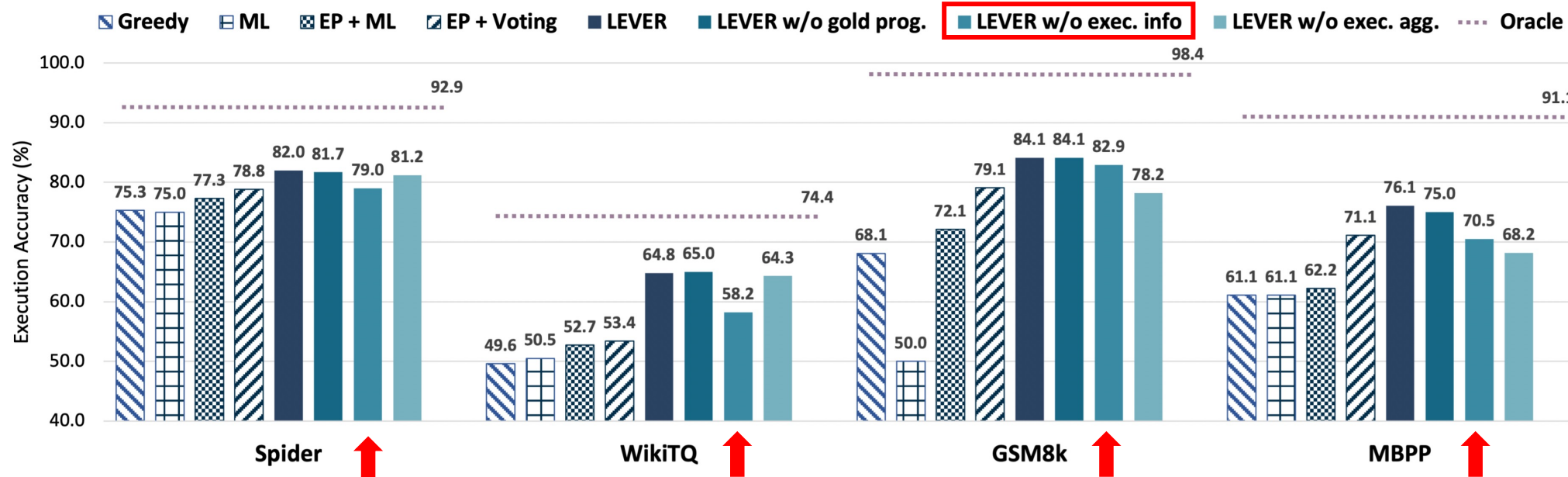
MBPP



Main Ablation on Codex

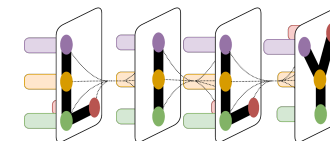


- Execution information are crucial to the performance improvement

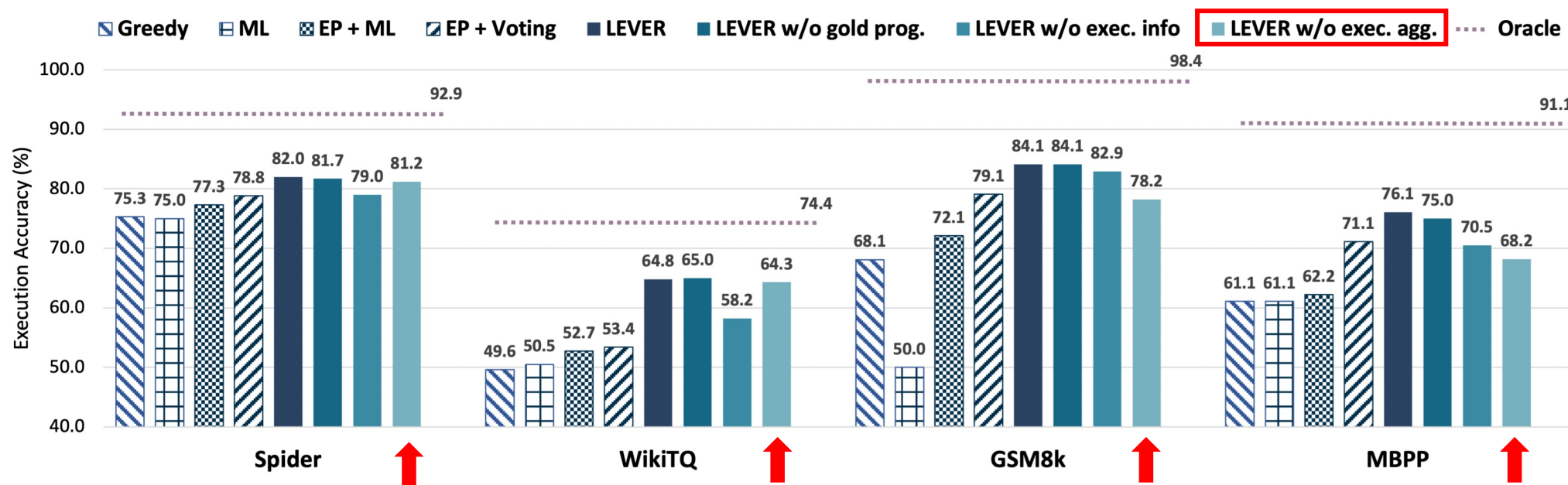




Main Ablation on Codex

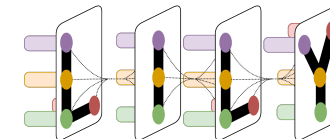


- Execution information are crucial to the performance improvement
- Exec. agg. works well for Python but not SQL generation datasets

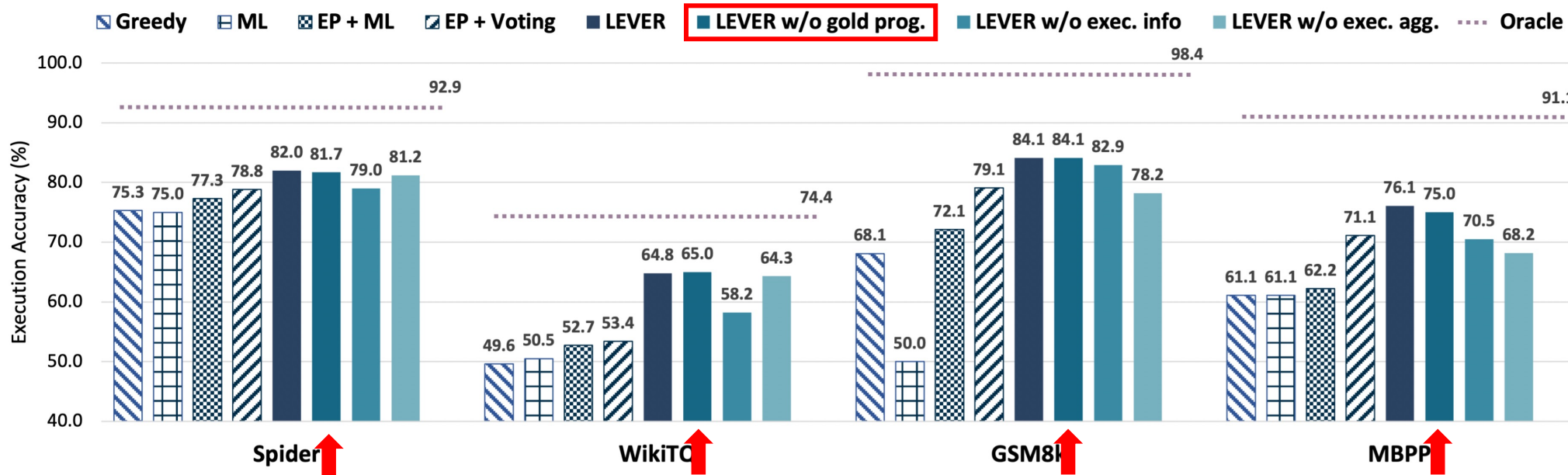




Main Ablation on Codex

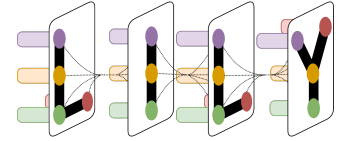


- Execution information are crucial to the performance improvement
- Exec. agg. works well for Python but not SQL generation datasets
- LEVER works well with weakly-supervised setting, where gold programs are not provided for learning





Results on Open-Source CodeLMs

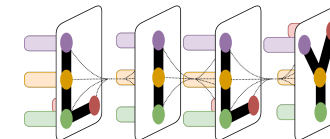


- Even larger improvements (e.g., up to 30.0%) are observed for InCoder and CodeGen models;
- Similar findings for ablation study
 - With the exception that voting & exec. agg. methods decreases the performance

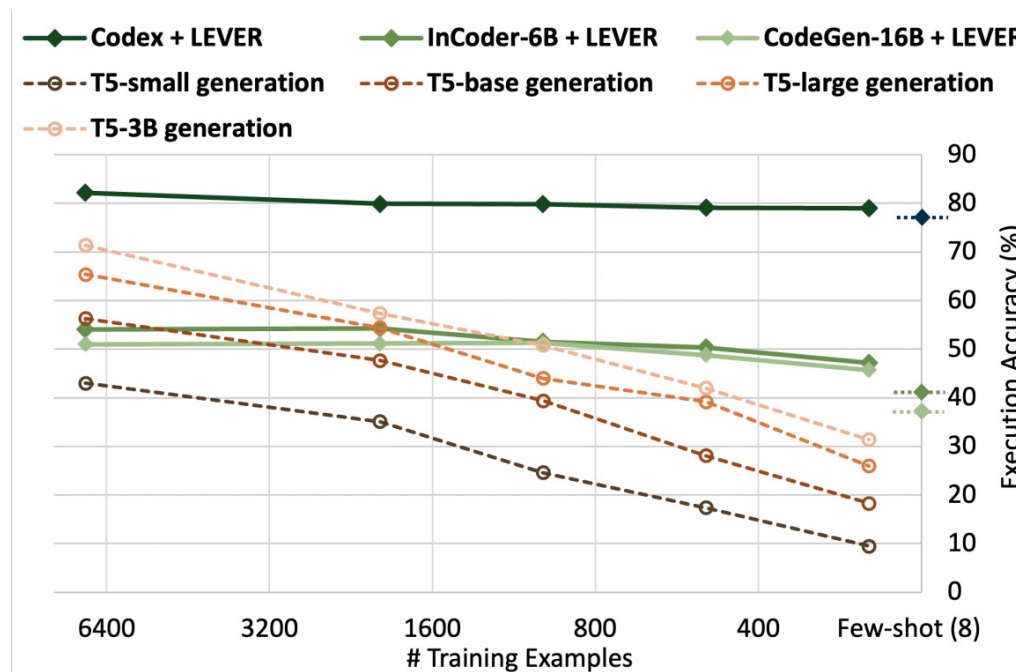
Methods	InCoder-6B		CodeGen-16B	
	Spider	GSM8k	Spider	GSM8k
<i>Previous work:</i>				
MBR-EXEC	38.2	-	30.6	-
Reviewer	41.5	-	31.7	-
<i>Baselines:</i>				
Greedy	24.1	3.1	24.6	7.1
ML	33.7	3.8	31.2	9.6
EP + ML	41.2	4.4	37.7	11.4
EP + Voting	37.4	5.9	37.1	14.2
LEVER 🌐	54.1	11.9	51.0	22.1
– gold prog.	53.4	-	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



Analysis: Data Scaling



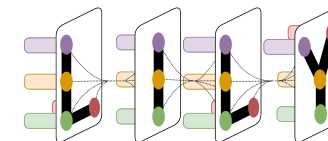
- LEVER works better under few-resource settings than generative finetuning



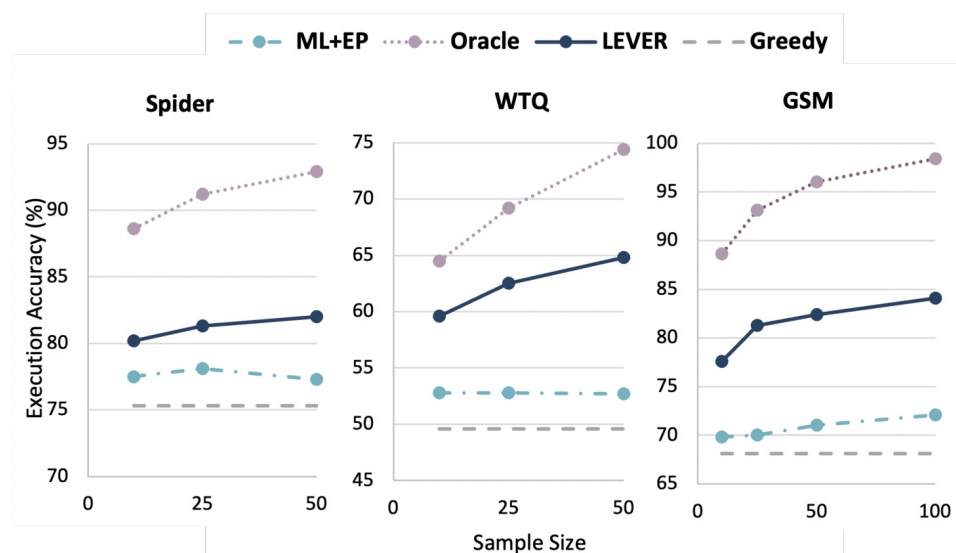
Verification vs. Generation w/ decreasing training data



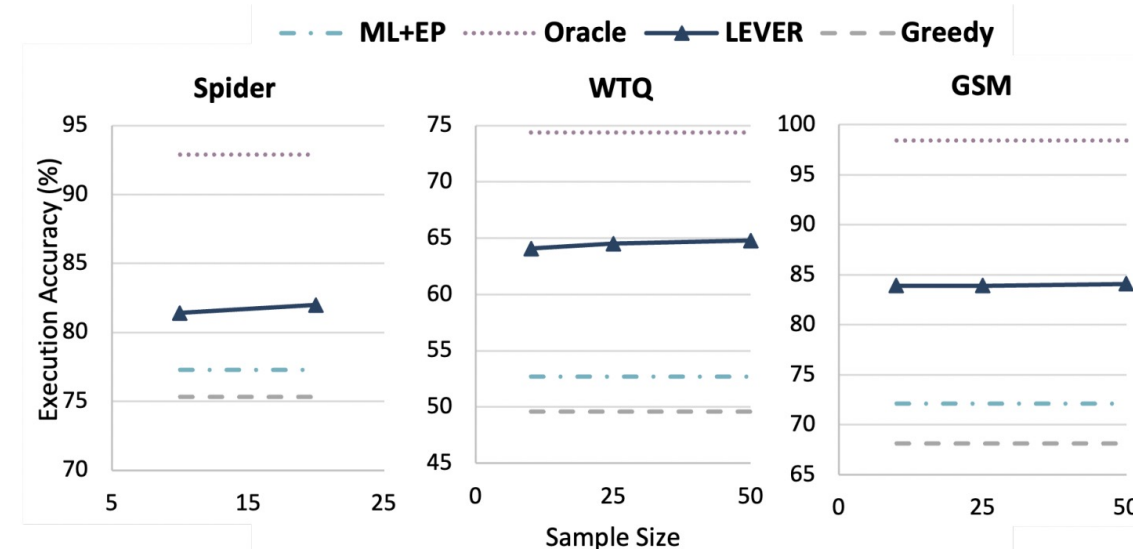
Analysis: Data Scaling



- LEVER is sensitive to the sample size at inference time but not training time thus a higher sampling budget should be applied during inference



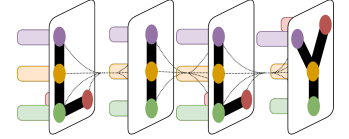
Ablation on sample size at inference time



Ablation on sample size at training time



Analysis: Between-LM Transfer



- LEVER still non-trivially improves the baseline performance in most cases;
- Transfer typically works better when the percentage of positive labels are closer

Target CodeLM & ML+EP Baseline		Source CodeLM (% 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)

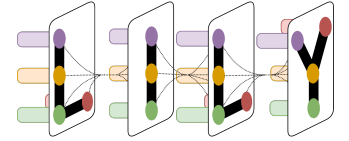
Between-LM Transfer results for Spider

Target CodeLM & ML+EP Baseline		Source CodeLM (% 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)

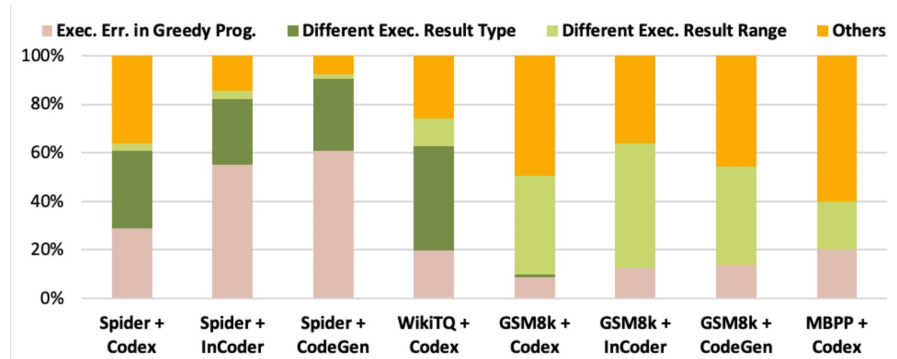
Between-LM Transfer results for GSM8k



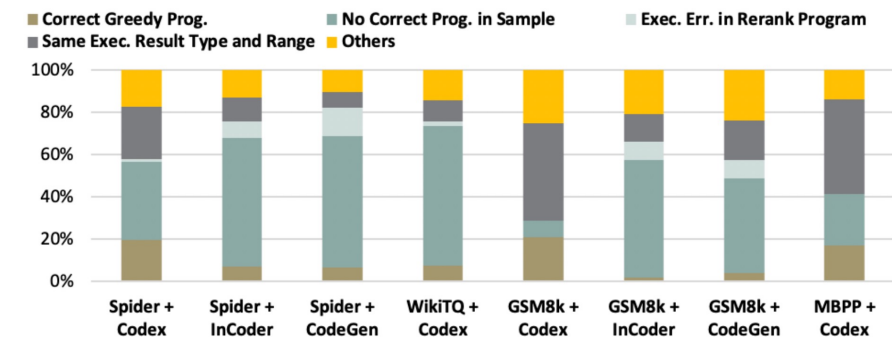
Quantitative Error Analysis



- When LEVER succeeds:
 - It is often because the execution results provide crucial information such as execution errors, variable type and range
- When LEVER fails:
 - 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 CodeLMs



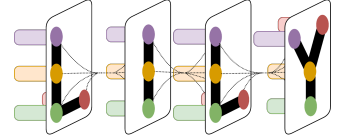
When LEVER reranks a correct program at the top but the greedy decoding fails.



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



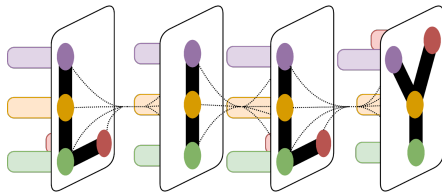
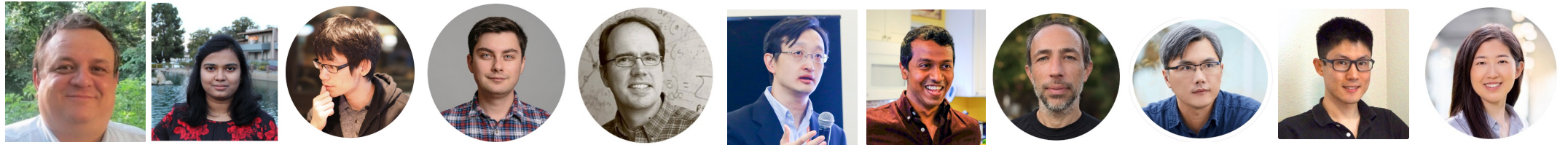
Takeaways



- How can we use a smaller model to help improve LLMs?
 - One way is to train a separate model that operates on the output of LLMs, such as verification, reranking, etc.
- We can also try to incorporate additional information (e.g., execution) in the separate model with the blackbox LLMs
- Neural-symbolic-neural approach is possible!

Thanks!

Also thanks to my wonderful collaborators:



Yale LILY Lab

 Meta AI

