SemCoder: Training Code Language Models w/ Comprehensive Semantics Reasoning

Yangruibo (Robin) Ding, Jinjun Peng, Marcus J. Min,
Gail Kaiser, Junfeng Yang, Baishakhi Ray

Columbia University

Semantics

Natural Language

[Flexible, Understandable, Static]

Implement a function that takes a list of potential energies, sorts them in ascending order, removes duplicates, and returns the indices of the unique sorted energies.

`[10.5, 8.2, 10.5, 7.1, 8.2] -> [3, 1, 0]`

Source Code

[Formal, Symbolic, Static & Dynamic]

```
def func(elems: List[float]) -> List[int]:
    elems_indices = list(enumerate(elems))
    sorted_elems_indices = sorted(elems_indices, key=lambda x: x[1])
    unique_elems = []
    unique_indices = []
    for index, elem in sorted_elems_indices:
        unique_elems.append(elem)
        unique_indices.append(index)
    return unique_indices
```

`[10.5, 8.2, 10.5, 7.1, 8.2]` ⚙ `[3, 1, 4, 0, 2]` ✗

```
def func(elems: List[float]) -> List[int]:
    elems_indices = list(enumerate(elems))
    sorted_elems_indices = sorted(elems_indices, key=lambda x: x[1])
    unique_elems = []
    unique_indices = []
    for index, elem in sorted_elems_indices:
        if elem not in unique_elems:
            unique_elems.append(elem)
            unique_indices.append(index)
    return unique_indices
```

`[10.5, 8.2, 10.5, 7.1, 8.2]` ⚙ `[3, 1, 0]` ✓

Related Work

- State-of-the-art Code LMs struggle to reason about runtime behavior[1][2][3][4]
  - I/O Prediction
  - Program States
  - Execution Path
  - Execution Traces

- Naïve SFT provides limited improvement for execution reasoning [1][2]

```
from typing import List

def unique_sorted_indices(energies: List[float]) -> List[int]: # [INPUT] {"energies": [10.5, 8.2, 10.5, 7.1, 8.2]}
[/INPUT]
  energy_dict = {}   # [STATE-0] {"energy_dict": {}} [/STATE-0]
  for idx, energy in enumerate(energies): # [STATE-1] {"idx": 0, "energy": 10.5} [/STATE-1][STATE-3] {"idx": 1,
"energy": 8.2} [/STATE-3] ... [STATE-8] {"idx": 4, "energy": 8.2} [/STATE-8]
...
return unique_sorted_indices # [OUTPUT] [3, 1, 0] [/OUTPUT]
```

---

[1] Gu, et al., 2024. CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution. ICML'24.
[2] Ni et al., 2024. NExT: Teaching Large Language Models to Reason about Code Execution. ICML'24.
[3] Chen et al., 2024. Reasoning Runtime Behavior of a Program with LLM: How Far Are We? ICSE'25.
[4] Liu, et al., 2024. CodeMind: A Framework to Challenge Large Language Models for Code Reasoning.

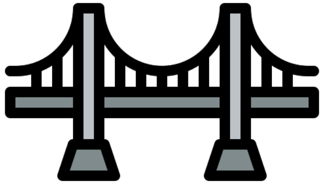
SemCoder: Reasoning Comprehensive Code Semantics

Approximate

- Overall Objectives
- High-level Functionalities
- Docstring & Documentation

Symbolic

- Symbolic Logic
- Algorithms
- Control Flow



Operational

- Input Coverage
- Execution Order
- Program States Transition

Abstract

- Ignore low-level details
- Invariant Properties
- Important Attributes

Code Semantics Learning Objective: Monologue Reasoning

> *The Internet data is not the data you want for your Transformers – a nearest neighbor actually gets you really far, surprisingly. What you want is the* ***inner-thought monologue of your brain****. If we had billions of that, AGI is here, roughly speaking.*
>
> *– Andrej Karpathy, 2024*

Forward Monologue

Symbolic: Code

```
def unique_sorted_indices(energies: List[float])
-> List[int]:
  energy_dict = {}
  for idx, energy in enumerate(energies):
    energy_dict.setdefault(energy, idx)
  sorted_unique_energies = sorted(set(energies))
  unique_sorted_indices = [energy_dict[energy]
for energy in sorted_unique_energies]
  return unique_sorted_indices

# Input: [10.5, 8.2, 10.5, 7.1, 8.2]
# Output: ???
```

Step-1: Approximate: Functionalities

The function `unique_sorted_indices` takes a list of floating-point numbers named `energies` as input. Its purpose is to find the indices of the unique elements in the input list when sorted in ascending order. Here’s how it works:

- A dictionary `energy_dict` is created to map each unique energy value to its first occurrence index.
- The `enumerate` function is used to iterate through the list and populate the dictionary with energy values as keys and their respective indices as values.
- Then, the function creates a sorted list of unique energy values from the input list using `sorted(set(energies))`.
- Finally, it creates a list of indices that correspond to these sorted unique energies using a list comprehension that retrieves the original indices from the dictionary.

Step-2: Abstract: Properties & Constraints

- The input list can contain duplicate values and is of variable length.
- The output list will have unique energy values sorted in ascending order and will contain their original indices from the input list.
- If an energy value is repeated, only its first occurrence's index is stored in the dictionary.
- The function ensures that the indices in the output list reflect the order of the unique values after sorting.

Step-3: Operational: Execution “Simulatio

```
1. Initialization:
   - `energy_dict` is initialized as an empty dictionary.
2. Filling the Dictionary:
   - The `for` loop begins, iterating through the list with `enumerate`:
     - Iter 1: ..
     ...
     - Iter 5: For `idx = 4`, `energy = 8.2`. Since 8.2 already exists in the dictionary, it is not
updated.
   Final state of `energy_dict` after the loop: `{10.5: 0, 8.2: 1, 7.1: 3}`.
3. Sorting Unique Energies:
   - The unique energies are derived and sorted, resulting in `sorted_unique_energies = [7.1, 8.2, 10.5]`.
4. Retrieving Original Indices:
   - A list comprehension retrieves the original indices corresponding to these sorted unique energies:
     - For `energy = 7.1`, the index is `3`.
     - For `energy = 8.2`, the index is `1`.
     - For `energy = 10.5`, the index is `0`.
   - Thus, `unique_sorted_indices` becomes `[3, 1, 0]`.
5. Return Statement:
   - The function returns the list `[3, 1, 0]`.
```

[1] Hunt and Thomas, 2000. The Pragmatic Programmer: from journeyman to master

Backward Monologue

Symbolic: Code

```
def unique_sorted_indices(energies: List[float])
-> List[int]:
  energy_dict = {}
  for idx, energy in enumerate(energies):
    energy_dict.setdefault(energy, idx)
  sorted_unique_energies = sorted(set(energies))
  unique_sorted_indices = [energy_dict[energy]
for energy in sorted_unique_energies]
  return unique_sorted_indices

# Input: ???
# Output: [3, 1, 0]
```

Step-1: Approximate: Functionalities

Step-2: Abstract: Properties & Constraints

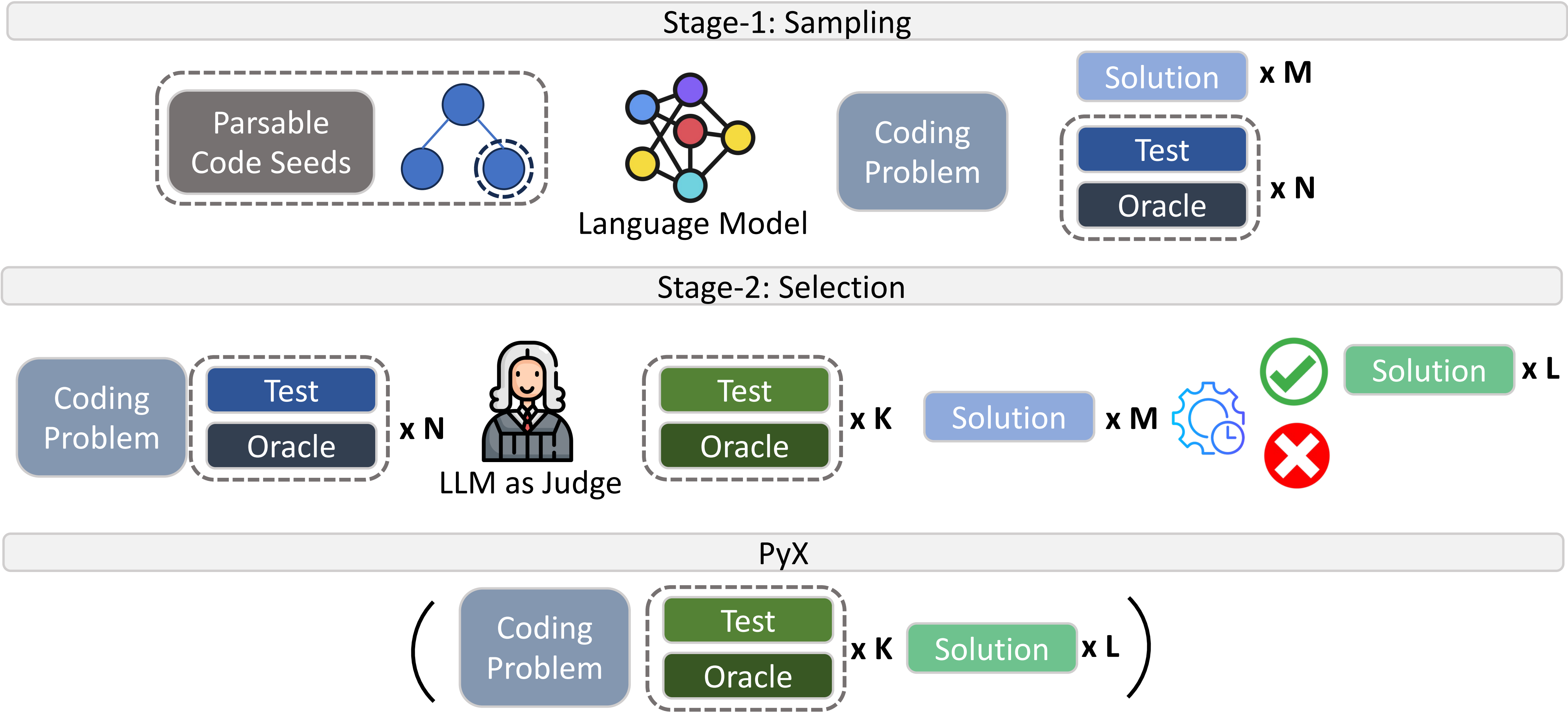
- The input list may contain duplicate energy values, but only the first occurrence's index will be recorded in `energy_dict`.
- The output will be ordered according to the sorted unique energy values.
- The length of the output list will equal the number of unique energy values in the input list

Step-3: Abstract & Operational: Execution "Deduction"

~~~
- Key constraints according to the output:
  - `energy_dict` must map some unique energies to these indices.
  - The first unique energy corresponds to index `3`, the second unique to `1`, and
the third unique to `0`.
- Energy Values:
  - Suppose the values in the `energies` list that correspond to these indices are:
    - `energies[3]` -> first unique energy (let's call it `e1`)
    - `energies[1]` -> second unique energy (let's call it `e2`)
    - `energies[0]` -> third unique energy (let's call it `e3`)
- Expected Ordering:
  - In the sorted list of unique energies, `e1` must be the smallest, followed by
`e2`, and then `e3`.
- Choice of Values:
  - A simple set of values array can be created. We must ensure that:
    - The energy at index `3` is less than the energy at index `1` and both should
be less than the energy at index `0`.

Let's select:
- `energies[3] = 1.0` (first unique, smallest value)
- `energies[1] = 3.0` (second unique, middle value)
- `energies[0] = 5.0` (third unique, largest value)

To ensure proper indexing and repetition, we can fill the rest of the list with
duplicates of such values, for instance:
```python
energies = [5.0, 3.0, 5.0, 1.0]

With this arrangement, we maintain the index mapping:
- `5.0` at index `0` (first occurrence),
- `3.0` at index `1` (first occurrence),
- `1.0` at index `3` (first occurrence).
~~~

PyX: The Dataset

PyX: Fully Executable Dataset with Tests



Stage-1: Sampling

Parsable Code Seeds

Language Model

Coding Problem

Solution x M

Test

Oracle

x N

Stage-2: Selection

Coding Problem

Test

Oracle

x N

LLM as Judge

Test

Oracle

x K

Solution x M

✓ Solution x L

✗

PyX

(Coding Problem

Test

Oracle

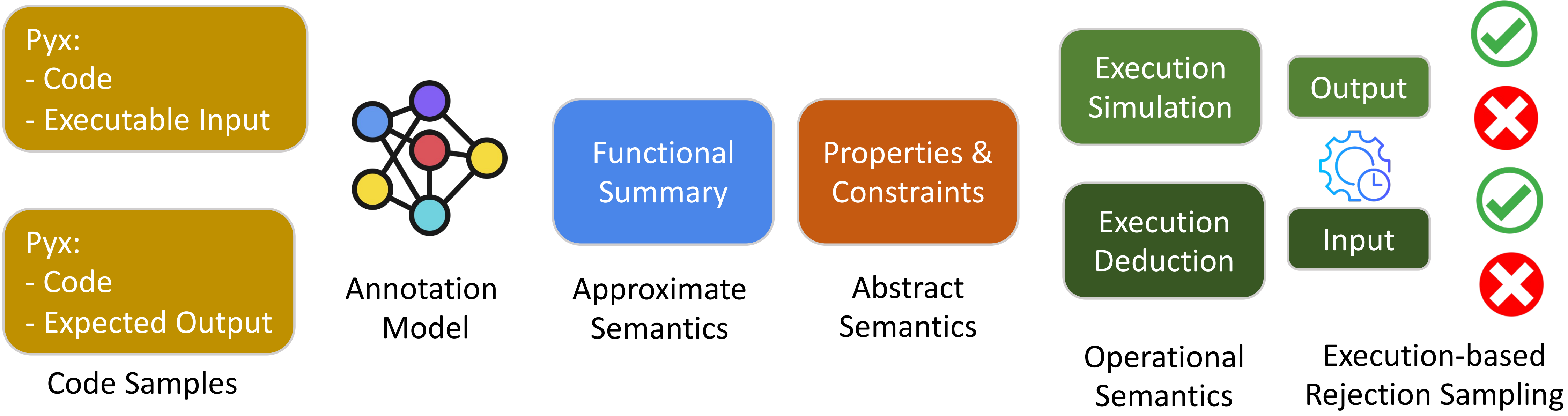
x K

Solution x L)

Wei et al., 2024. Magicoder: Empowering Code Generation with OSS-Instruct. ICML'24.

Monologue Annotation: Rejection Sampling

Forward Monologue



Pyx:
- Code
- Executable Input

Pyx:
- Code
- Expected Output

Code Samples

Annotation Model

Functional Summary

Approximate Semantics

Properties & Constraints

Abstract Semantics

Execution Simulation

Execution Deduction

Operational Semantics

Output ✓ ✗

Input ✓ ✗

Execution-based Rejection Sampling

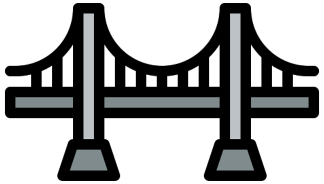
Backward Monologue

Joint Training: Generation and Reasoning

Implement a function that takes a list of potential energies, sorts them in ascending order, removes duplicates, and returns the indices of the unique sorted energies.

```
def unique_sorted_indices(energies: List[float])
-> List[int]:
  energy_dict = {}
  for idx, energy in enumerate(energies):
    energy_dict.setdefault(energy, idx)
  sorted_unique_energies = sorted(set(energies))
  unique_sorted_indices = [energy_dict[energy]
for energy in sorted_unique_energies]
  return unique_sorted_indices\
```

Natural Language



Source Code

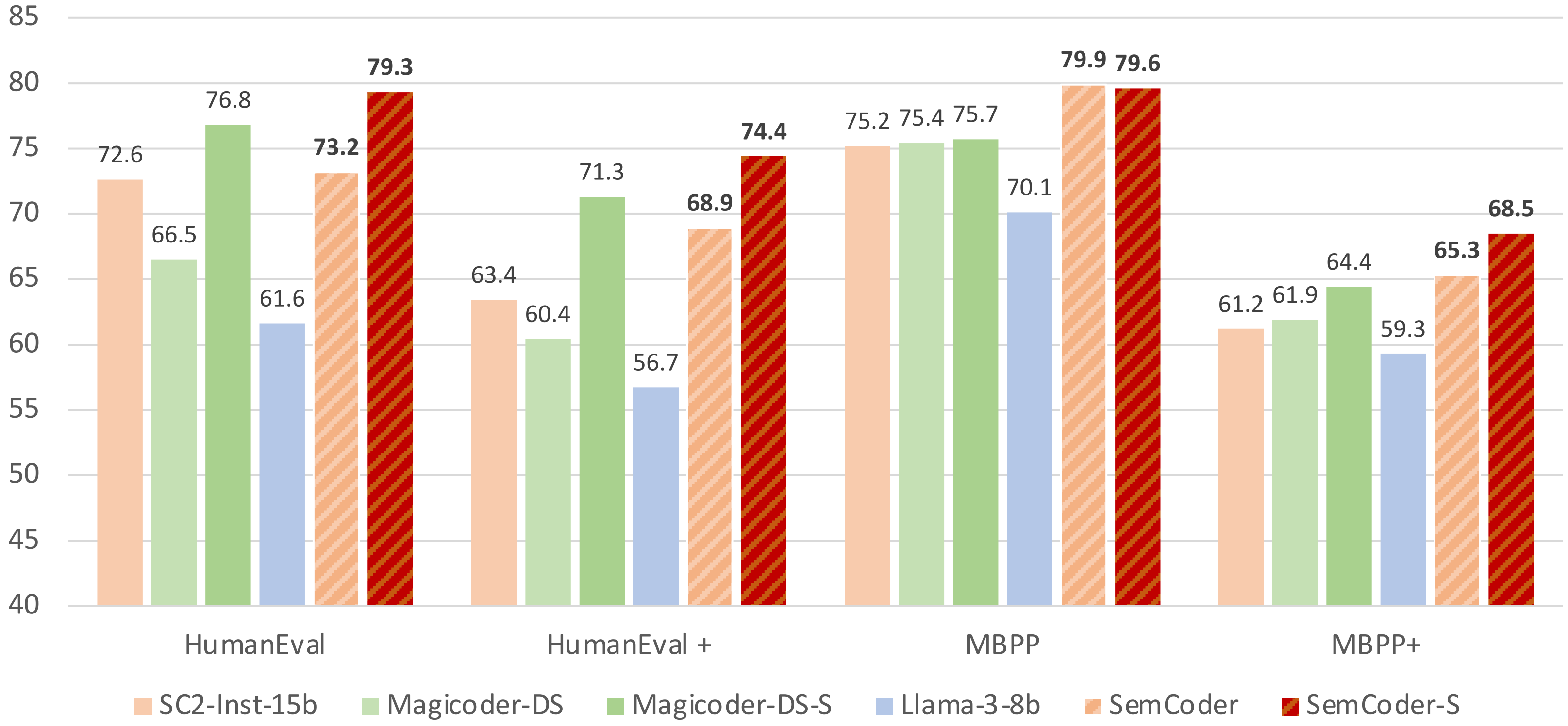
- The input list can contain duplicate values and is of variable length.
- The output list will have unique energy values sorted in ascending order and will contain their original indices from the input list.
- If an energy value is repeated, only its first occurrence's index is stored in the dictionary.
- The function ensures that the indices in the output list reflect the order of the unique values after sorting.

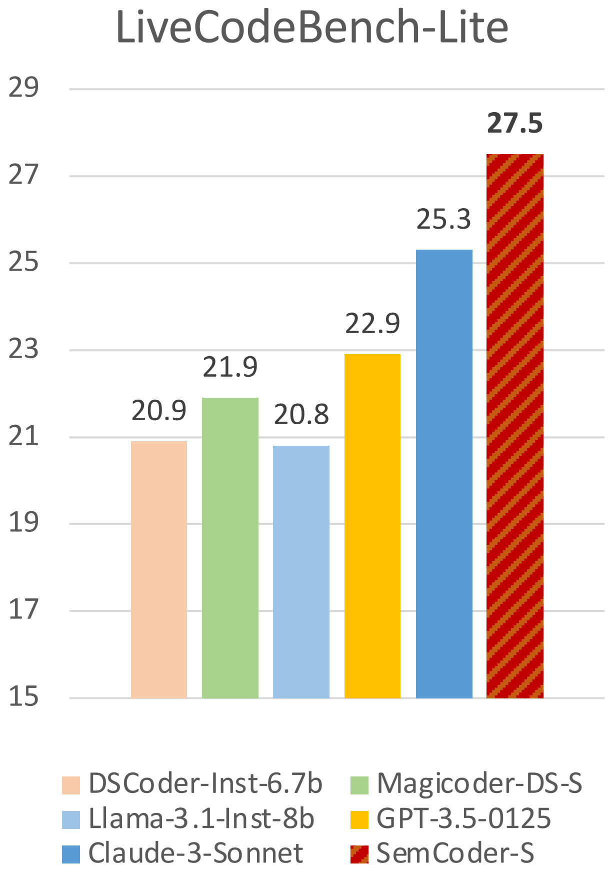
```
Sorting Unique Energies:
 The unique energies are derived and sorted,
resulting in `sorted_unique_energies = [7.1, 8.2,
10.5]`.
Retrieving Original Indices:
 For `energy = 7.1`, the index is `3`.
 For `energy = 8.2`, the index is `1`.
 For `energy = 10.5`, the index is `0`.
 Thus, `unique_sorted_indices` becomes `[3, 1, 0]`.
```

Experiments

Results: Code Generation

- SemCoder shows competitive performance in code generation

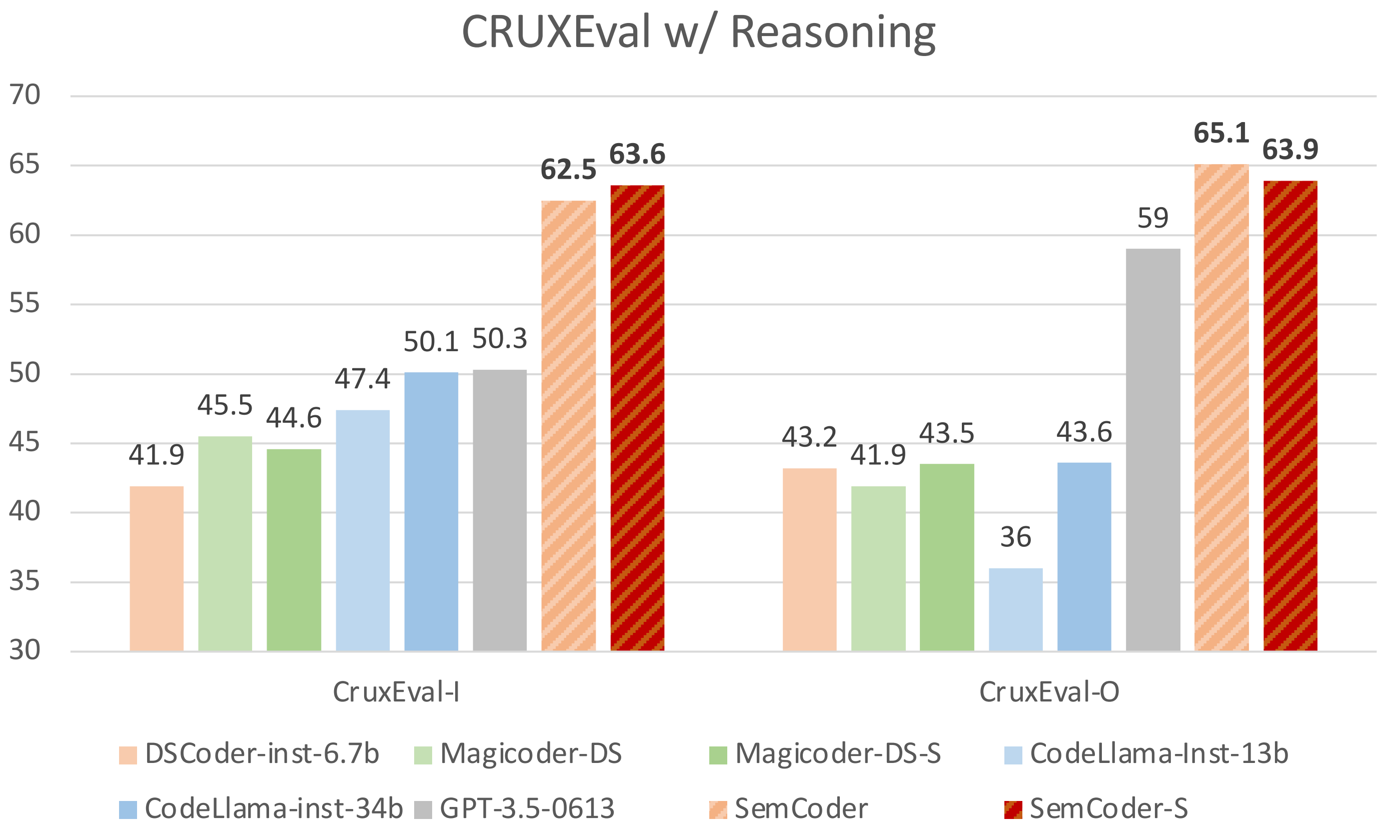


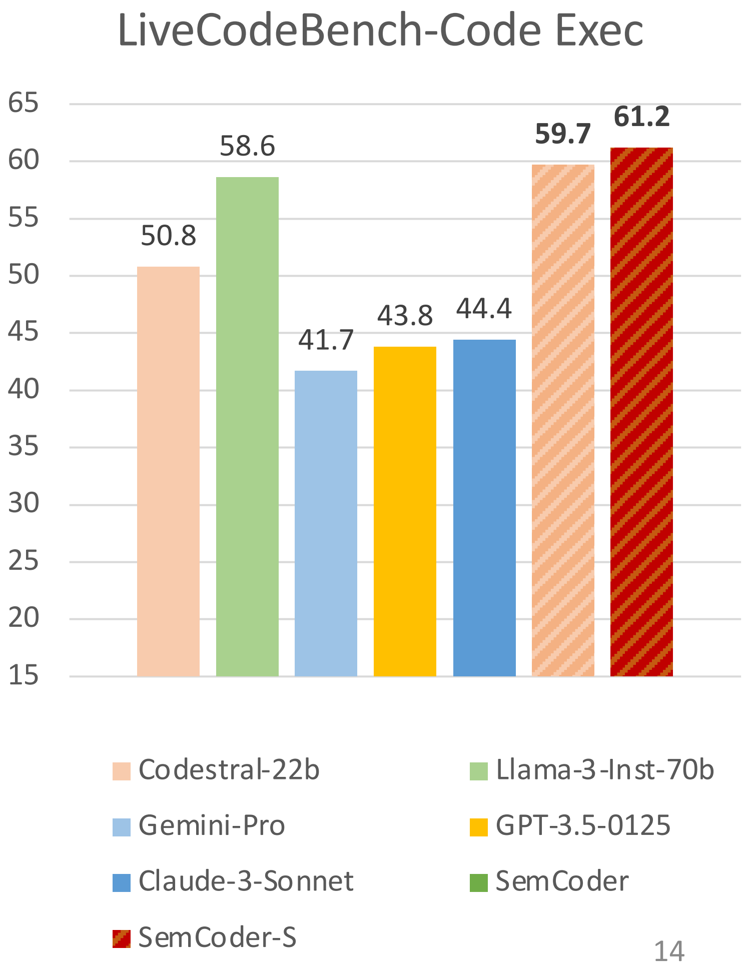


[1] Jain, et al., 2024. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code.
[2] Liu et al., 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation.

Results: Execution Reasoning

- SemCoder significantly outperforms in execution reasoning





[1] Gu, et al., 2024. CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution. ICML'24.
[2] Jain, et al., 2024. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code.

Thanks!!

Pre-print: https://arxiv.org/abs/2406.01006
Model, Data, & Code: https://github.com/ARiSE-Lab/SemCoder