LLM Code Generation

Willis Reid
April 2024
University of North Carolina

Table of Content

- Overview

Machine learning is the future.

- Goal

AlphaCodium multi-phase, code-oriented, test-based iterative method improves LLM performance when faced with code issues.

- Test Methodology

Results are regularly and noticeably improved by the suggested flow.

- Results & Metrics

AlphaCode applied a brute force-style strategy with a notably greater quantity of LLM calls.

- This will be for future work

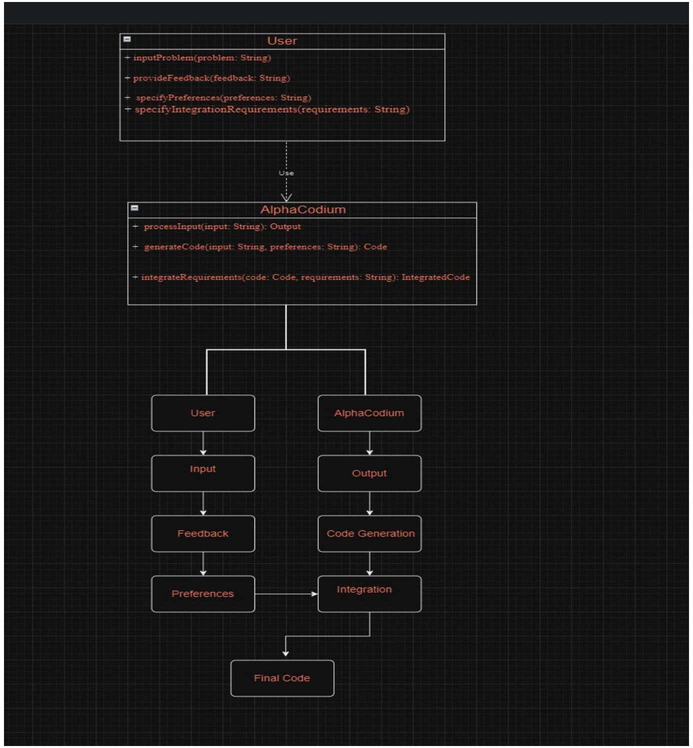
Display how humans make errors while coding.

Motivation

I chose this topic because I found it interesting and I believe many people will enjoy it too. As I delve deeper and examine the complexity of this topic, my findings continue to astound me. My main reason for choosing this topic is my fascination with it, and I'm eager to share the intriguing thoughts and discoveries I come across.

Existing Related Approaches

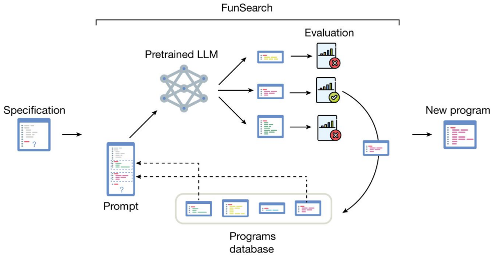
- Created a user diagram to illustrate project workflow.
- Executed terminal commands to process and run code efficiently.
- Procured the dataset required for the project from Hugging Face.



Method # 1

Evaluating Large Language Models Trained on Code.

	INTRODUCTORY	INTERVIEW	COMPETITION
GPT-NEO 2.7B RAW PASS@1	3.90%	0.57%	0.00%
GPT-NEO 2.7B RAW PASS@5	5.50%	0.80%	0.00%
1-SHOT CODEX RAW PASS@1	4.14% (4.33%)	0.14% (0.30%)	0.02% (0.03%)
1-SHOT CODEX RAW PASS@5	9.65% (10.05%)	0.51% (1.02%)	0.09% (0.16%)
1-SHOT CODEX RAW PASS@100	20.20% (21.57%)	2.04% (3.99%)	1.05% (1.73%)
1-SHOT CODEX RAW PASS@1000	25.02% (27.77%)	3.70% (7.94%)	3.23% (5.85%)
1-SHOT CODEX FILTERED PASS@1	22.78% (25.10%)	2.64% (5.78%)	3.04% (5.25%)
1-SHOT CODEX FILTERED PASS@5	24.52% (27.15%)	3.23% (7.13%)	3.08% (5.53%)



Evaluating Large Language Models Trained on Code

```
You will be given a string of words separated by
    commas or spaces. Your task is
to split the string into words and return an array
    of the words.

For example:
words_string("Hi, my name is John") == ["Hi", "my",
    "name", "is", "John"]
words_string("One, two, three, four, five, six") ==
    ["One", "two", "three", "four", "five", "six"]
"""

### COMPLETION 1 (CORRECT): ###
words = []
splitted_word = ""
for char in s:
    if char == "␣":
        if splitted_word != "":
            words.append(splitted_word)
            splitted_word = ""
    elif char != ",":
        splitted_word += char
if splitted_word != "":
    words.append(splitted_word)
return words


### COMPLETION 2 (WRONG): ###
return s.split()


### COMPLETION 3 (WRONG): ###
words = s.split("␣")
words = [w.strip(",") for w in words]
words = [w.strip() for w in words]
return words


### COMPLETION 4 (WRONG): ###
# Split the string into words
words = s.split(',')
# Remove any whitespace around each element
words = [w.strip() for w in words]
# Return an array of the words in the string
return words


### COMPLETION 5 (WRONG): ###
return s.split()


### COMPLETION 6 (WRONG): ###
return s.split()


### COMPLETION 7 (WRONG): ###
return s.split(',')


### COMPLETION 8 (WRONG): ###
return s.split(sep=None)
```

```
def is_prime(n):
    """Return true if a given number is prime, and
        false otherwise.
    >>> is_prime(6)
    False
    >>> is_prime(101)
    True
    >>> is_prime(11)
    True
    >>> is_prime(13441)
    True
    >>> is_prime(61)
    True
    >>> is_prime(4)
    False
    >>> is_prime(1)
    False
    """
```

```
### COMPLETION 1 (WRONG): ###
for i in range(2, n):
    if n % i == 0:
        return False
return True


### COMPLETION 2 (WRONG): ###
for i in range(2, n):
    if n % i == 0:
        return False
return True


### COMPLETION 3 (WRONG): ###
# TODO: implement this function
pass


### COMPLETION 4 (CORRECT): ###
if n < 2: return False
if n == 2: return True
if n%2 == 0: return False
return not any(n%k == 0 for k in range(3,int(n
    **0.5)+1,2))


### COMPLETION 5 (WRONG): ###
def helper(n):
    # check if an odd number is multiple of any
        integer up to the square root of the number
    # if this is true, n is not prime
    for i in range(2, int(n**0.5 + 1)):
        if n % i == 0:
            return False
    return True
# if helper returns true, then n is prime, else n
    is not prime
if helper(n):
    return True
else:
    return False


### COMPLETION 6 (CORRECT): ###
prime = True
if n == 1:
    return False
for i in range(2, n):
    if n % i == 0:
        prime = False
return prime


### COMPLETION 7 (WRONG): ###
for i in range(2, n):
    if n % i == 0:
        return False
return True


### COMPLETION 8 (CORRECT): ###
if n == 2 or n == 3:
    return True
if n < 2 or n % 2 == 0:
    return False
if n < 9:
    return True
if n % 3 == 0:
    return False
r = int(n**0.5)
f = 5
while f <= r:
    if n % f == 0:
        return False
    if n % (f+2) == 0:
        return False
    f += 6
return True
```

Method # 2

CodeChain: Improving Code Generation through Modular Self-Revision

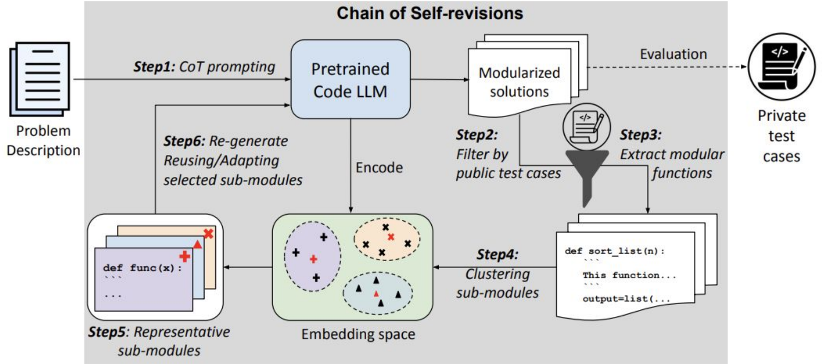
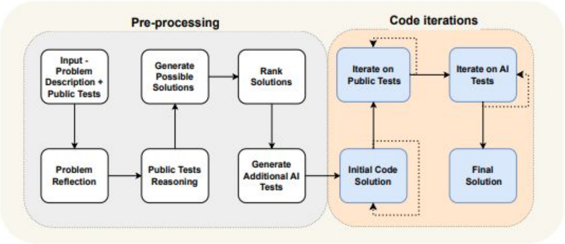


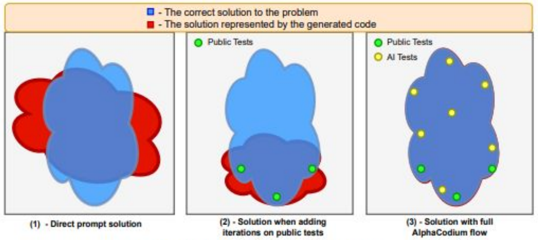
Figure 2: An overview of CodeChain: a pretrained LLM is first instructed with chain-of-thought prompting to generate a set of modularized solutions. Generated sub-modules are then extracted from potentially correct solutions and grouped into different semantic clusters. The cluster centroids are selected as representative sub-modules to condition the next self-revision round. The model is instructed to reuse or adapt these modules into its revised solutions.

Method That Was Duplicate

AlphaCodium



(a) The proposed AlphaCodium flow.



(b) Illustrating the improvement from AlphaCodium.

Results

model	set	method	score (pass@5)
DeepSeek -33B [3]	validation	Direct	7%
		AlphaCodium	**20%**
	test	Direct prompt	12%
		AlphaCodium	**24%**
GPT-3.5	validation	Direct prompt	15%
		AlphaCodium	**25%**
	test	Direct prompt	8%
		AlphaCodium	**17%**
GPT-4	validation	Direct prompt	19%
		AlphaCodium	**44%**
	test	Direct prompt	12%
		AlphaCodium	**29%**

model	set	method	score
GPT-3.5	validation	AlphaCodium (pass@5)	**25%**
		CodeChain (pass@5)	17%
	test	AlphaCodium (pass@5)	**17%**
		CodeChain (pass@5)	14%
GPT-4	validation	AlphaCodium (pass@5)	**44%**
AlphaCode		AlphaCode (pass@10@1K)	17%
		AlphaCode (pass@10@100K)	24%
GPT-4	test	AlphaCodium (pass@5)	**29%**
AlphaCode		AlphaCode (pass@10@1K)	16%
		AlphaCode (pass@10@100K)	28%

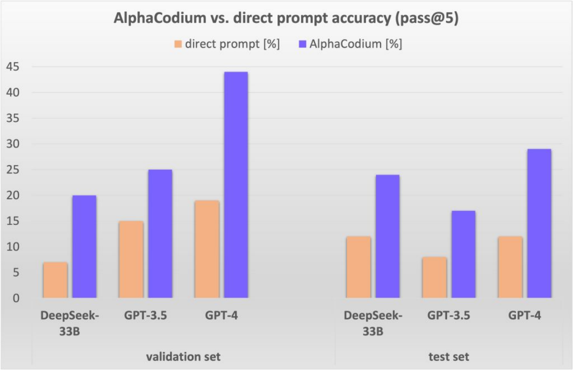
Results

Compares AlphaCodium's performance with other models like AlphaCode and CodeChain.

Valuation Metric: Utilizes the pass@k metric to quantify the percentage of successfully solved problems using k generated solutions per problem.

AlphaCodium: demonstrates its effectiveness in solving problems compared to single, well-designed direct prompts.

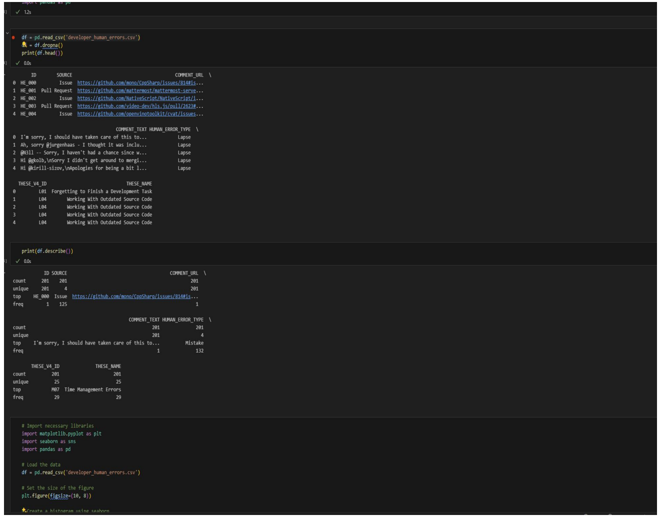
AlphaCode and CodeChain: Represents other models used for comparison in the analysis.

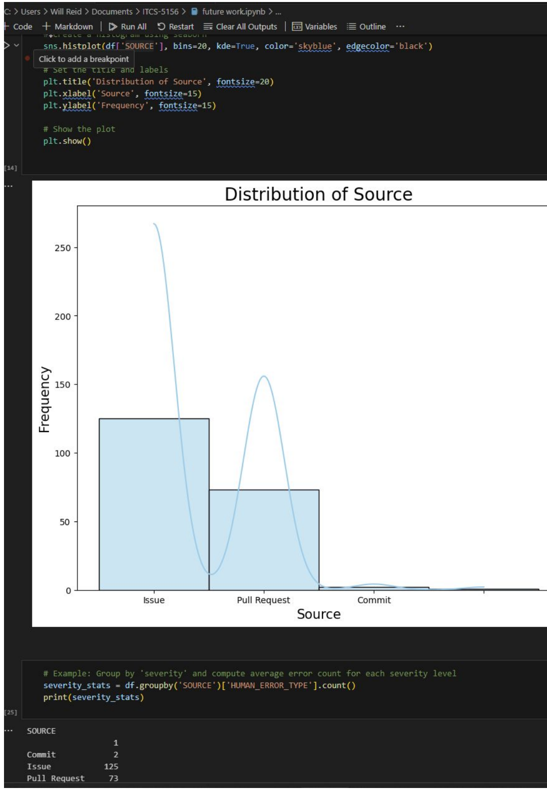



WHAT'S
NEXT?

Future Work

- Identified human errors in coding through dataset.
- Curated dataset showcasing common coding mistakes and associated code snippets.
- Parsed text to detect errors and visualized alongside code outputs.
- Aimed to provide clear understanding of coding mistakes through visualization.





Conclusion

- Many people fear the impact of machine learning in our field.
- Common concern: Fear of job loss or inability to find work.
- Clarification: Not necessarily true.
- Emphasize the potential for growth and adaptation in the industry.

Embracing Fear

https://www.youtube.com/watch?v=AgyJv2Qelwk&ab_channel=Fireship

Resources

Ridnik, Tal, et al. "Code Generation with Alphacodium: From Prompt Engineering to Flow Engineering." *arXiv.Org*, 16 Jan. 2024, arxiv.org/abs/2401.08500. Accessed 23 Feb. 2024. (https://arxiv.org/abs/2401.08500)

Chen, Mark, et al. "Evaluating Large Language Models Trained on Code." *arXiv.Org*, 14 July 2021, arxiv.org/abs/2107.03374. Accessed 23 Feb. 2024. (https://arxiv.org/abs/2107.03374)

Le, Hung, et al. "CodeChain: Towards Modular Code Generation through Chain of Self-Revisions with Representative Sub-Modules." *arXiv.Org*, 28 Nov. 2023, arxiv.org/abs/2310.08992. Accessed 23 Feb. 2024. (https://arxiv.org/abs/2310.08992)