Exploration of Human-style Code Generation NLP project

Gabriel Fiastre, Shane Hoeberichts
gabriel.fiastre@dauphine.eu, shane.hoeberichts@dauphine.eu

Abstract

A lot of recent work on both Large Language Models (LLMs) and generative models for text showed particular improvement on the code generation task and reaffirmed the higher importance of this task. Inspired by the paper "Evaluating large language trained on Code" [2] and the performance of the associated model, Codex, we decide to focus on the code generation task and to discuss the convergence between code generation models and "human-style" logic and development concepts in the process of code generation. We first propose to reproduce some important results, then consider extended use of existing models based on human-logic thinking, and discuss possible further extensions which could help enhance model alignment with user intent and human-like reasoning. We focus on the code generation task restricted to generating python functions from a doc string input, evaluating our experiments on well-established benchmarks for functional correctness. Code is available at https://github.com/Shanehoeb/NLP_code_generation

1. Introduction

Code generation is a fundamental task in the field of Natural Language Processing (NLP), which aims to automatically generate executable code from high-level natural language descriptions. It involves bridging the semantic gap between human-readable text and machine-executable instructions. The code generation task plays a vital role in enabling non-programmers to interact with complex software systems by expressing their intentions in natural language, thus reducing the barriers to software development. Additionally, code generation can assist software developers in automating repetitive coding tasks and completion suggestions, thus significantly reducing development time and cost. Leveraging the power of large language models (LLMs) has shown promise in addressing the challenges of code generation, as these models excel in understanding and generating natural language while capturing programming concepts and syntax.

Large language models (LLMs) represent a significant breakthrough in the field of artificial intelligence, particularly in natural language processing [7]. LLMs are deep learning models that have been trained on vast amounts of text data to understand and generate human-like text. These models are characterized by their immense size, typically consisting of billions of parameters, which allows them to capture and learn complex patterns, semantics, and syntactic structures of natural language. LLMs employ transformer architectures that enable them to process and generate text by attending to different parts of the input sequence, facilitating long-range dependencies and contextual understanding. Through pre-training on diverse corpora, LLMs acquire broad linguistic knowledge and the ability to generate coherent and contextually appropriate responses. The impressive capabilities of LLMs have led to their applications in various domains, including language translation, text generation, question answering, and more recently, code generation.

Challenges - The primary challenges of code generation encompass code quality, efficiency, and reliability. Generating high-quality code that adheres to standards, is readable, and follows best practices is crucial. Efficient code generation involves producing optimized solutions, while reliability requires accurate handling of edge cases and errors. Recent progress in text generation within Natural Language processing provides an opportunity to tackle these challenges. Leveraging advancements in generative models and in particular large language models (LLMs), which excel in understanding and generating human-like text, can enhance code generation. However, ensuring alignment between the model's output and the user's intent remains a significant challenge.

Overall, we wish to explore the capabilities of code generation models, determine their limits and discuss potential research avenues for improvement. First, we reproduce some important results about Large Language models and their ability not only to perform very well on zero-shot, but moreover to improve substantially with fine-tuning on code. Then, we try to explore potential enhancements of the LLM state of the art models by making a step towards more

```
def incr_list(l: list):
    """Return list with elements incremented by 1.
    >>> incr_list([1, 2, 3])
    [2, 3, 4]
    >>> incr_list([5, 3, 5, 2, 3, 3, 9, 0, 123])
    [6, 4, 6, 3, 4, 4, 10, 1, 124]
    """
    return [i + 1 for i in l]
```

Figure 1. The task of python function body generation from doc-string input. Generated code is highlighted in yellow and can contain multiple lines. Source : [2]

human-like reasoning. We also explore an alternative in the recent breakthrough in generative AI that are diffusion models. Finally, we discuss further models and techniques for aiming at more convergence towards State of the art code generation models and human-like programming logic.

2. Code generation models

2.1. Task

We focus on the task of generating the body of a python function given its doc-string as a input, ie a natural language description of the desired function for the model to generate. This task, described in 1 is particularly challenging as it leverages natural language understanding as well as code generation, hence text generation in a functional, programming language. Programming languages have a very specific syntax and available corpus or datasets are very rarely available in a clean, royalty-free manner. All those challenges make natural langage processing tasks related to code very difficult to solve as the models are hard to train in a stable, robust way [11]. Moreover, evaluating these models is another major problem as programming languages cannot be evaluated in the same way as natural language. We discuss this issues and introduce the state of the art framework for the evaluation of code evaluation, as we propose to reproduce some important evaluation results of two state of the art Large Language models : *LLaMA* and *CodeGen*.

2.2. Evaluation

While match-based metrics have proven effective in evaluating natural language processing tasks, they face limitations when applied to code evaluation. Code generation involves the production of executable instructions, where functional equivalence is of paramount importance. Match-based metrics, which rely on exact text matching or token-level comparisons, may fall short in capturing the subtleties and nuances of code functionality. Unlike natural language, where synonymous phrases or paraphrases are often interchangeable, code snippets with different text representations can have identical functionality. Therefore, relying solely on match-based metrics comparing samples to a reference solution may lead to biased evaluations that prioritize superficial lexical similarities rather than functional correctness. Assessing code generation models should take into account the semantics and behavior of the generated code, emphasizing the ability to produce functionally equivalent solutions. The authors of "Evaluating large language trained on Code" [2] propose an alternative solution; they argue that a better framework to assess code generation is to capture functional correctness by having associated unit tests the sample must pass to be deemed correct.

The authors present an evaluation metric as well as a dataset to answer for the limitations of match-based valuation. The metric is a redefined unbiased version of an already existing metric named pass@k. The authors propose to generate $n \geq k$ samples per task and evaluate pass@k over a set of problems as follows:

$$pass@k = \mathbb{E}[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}] \quad (1)$$

where c is the number of correct generated samples. The authors also present HumanEval, a dataset composed of over 160 handwritten programming problems. Each problem is composed of a function signature, a doc-string and body, and a set of associated unit tests (7.7 per problem on average). The dataset aims to assess language comprehension, reasoning, algorithms, and simple mathematics. A generated sample for a problem is designated as correct if it passes the set of associated unit tests, thus giving a framework to evaluate functional correctness.

2.3. Results

Inspired by the previous work done [2] and the recent advances in Large Language models, we propose to reproduce some very important results demonstrating the superiority of Large Language models, but most importantly their ability to improve with fine-tuning for a specific task. **LLaMA** [9] is a recent state of the art Large Language Model which demonstrated state of the art capabilities regarding its relatively small size when comparing to some other LLMs. Here we use the model with 7B parameters, the highest we could achieve with our limited hardware. LLaMA, as the GPT family of models, mainly impressed by it's capability of performing brillantly on zero-shot tasks, and especially for low-level reasoning tasks. **CodeGen** [6] is a large language of approximately the same size as LLaMA : we use the model with 6B parameters. It is trained on 3 state of the art, cleaned code datasets that are The Big Pile, BigQUery and BigPython, thus trying to improve on generic foundation models such as LLaMA by specializing on code understanding and generation.

We compare the results obtained in 1 and observe that the LLaMA model demonstrates impressive performance

Model	Size	pass@1
LLaMA	7B	15.85
CodeGen	**6B**	**27.31**

Table 1. Comparison between LLaMA 7B and CodeGen 6B pass@1 on HumanEval

on zero-shot code generation. We further observe that the fine tuning is very powerful for LLMs even for complicated and specific tasks including some low level reasoning, as even with less parameters a similar model like CodeGen clearly outperforms LLaMA and demonstrates very good performance.

3. Code Generation with Development Concepts

3.1. Unit Test Generation

One of the challenges in code generation is sample selection. When used as an auto-completion tool, code generation must suggest a single sample to the user at a time to not overwhelm them. Ideally, the suggested sample should be the best generated sample for the problem. However, considering that in practice unit tests will not be made available for each problem, the selected sample must be chosen without knowing if it is valid. In "Evaluating large language trained on Code" [2], the authors show that by choosing the sample with the highest mean-log probability; however, the results, presented in Figure 2, illustrate a large gap between this selection process and the oracle which chooses the sample that passes the unit tests (44.5% solved for mean-log probability and 77.5% for oracle).

As such, we propose a method in the hope of reducing the gap between selection and oracle. We aim to create an alternative selection method which relies on the generation of pseudo-unit tests. The model will not only generate solutions sample but also unit tests from the function description of each problem. A pseudo-evaluation can thus be performed to select the generated solution sample that passes the most pseudo-unit tests. We hypothesize that this will increase the pass rate in the setting of single sample evaluation. Although not all generated unit tests will be valid, we can expect that some will and that the sample soluton that passes the most has the highest chance of passing the real unit tests associated to the problem.

3.2. Code-T

Researchers from Micosoft have recently implemented such a method dubbed CodeT [1], a code generation method with generated test-driven dual execution agreement. The method leverages pre-trained code generation models to produce a large number of test cases for each presented

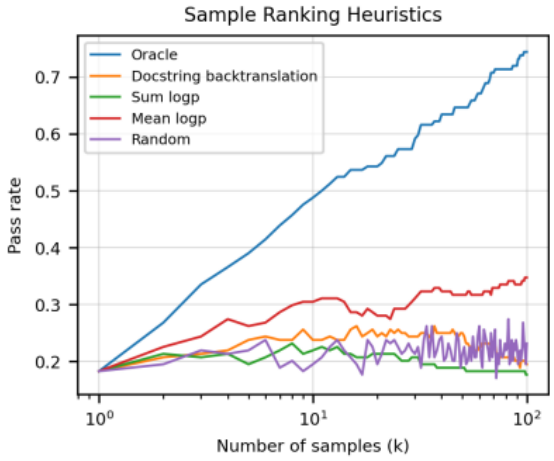

Figure 2. Codex model performance in the setting where we generate multiple samples, but only evaluate one [2].

problem by providing additional instructions in the prompt. They then use a dual agreement procedure inspired by the RANSAC algorithm [3]. By executing each generated solution on each generated test case, multiple groups of multiple groups of code solution and test case pairs within which some solutions pass the same tests and are thus functionally equivalent. The authors, as well as assessing which solutions pass the most tests, suggest that solutions with more other functionally equivalent solutions are more consistent with the proposed problem. As such, they define a new method for sample selection based on both a pseudo-functional correctness evaluation on generated test cases and measuring the functional similarity of a solution with respect to the others generated.

This method significantly increases performance, as seen in Figure **??** where performance is increased when prompted using the test sampling selection. Interestingly, we can see that the random selection method catches up for increasing k; this shows th limitations of the method, where the similarity measure between sample solutions negatively impacts performance by promoting groups of similar erroneous solutions. We propose to demonstrate the efficiency of this method on a recently newly released model, CodeGen2.

3.3. Results

CodeGen2 [5] was released very recently : it is very similar to the original CodeGen, but with a more unified training which should yield more robus results. The weights available are not finetuned as much as the better CodeGen weights (such as CodeGen MONO), but the model should be able to outperform CodeGen on code generation tasks if correclty further tuned. We thus propose to use CodeGen2, and integrate it to the Code-T Code in order to set a first

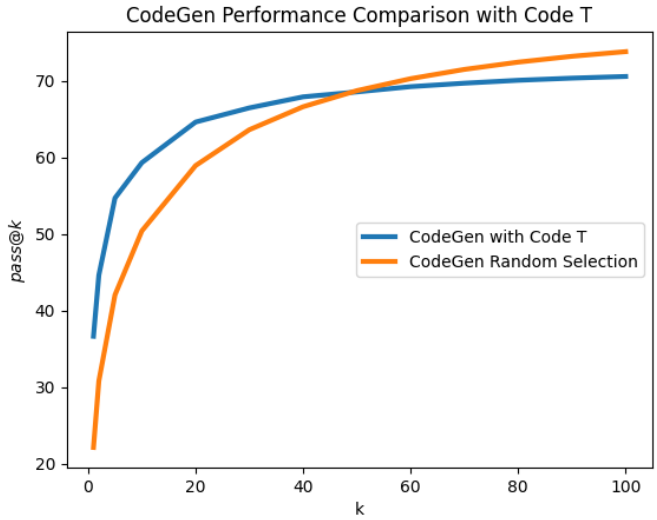

Figure 3. CodeGen results with Code T and random solution sample selection. Settings: 100 generated samples, temperature 0.8, top p 0.95. 5 unit tests generated per problem.

Model	Test module	pass@1
CodeGen2 6B	×	0.41
CodeGen2 6B	✓	**7.44**

Table 2. Comparison between CodeGen2 (6B parameters) and CodeGen2-T on pass@1 on HumanEval, with following sampling parameters : temperature = 0.2, top_p =0.95

step towards outperforming other code-T models (State of the art just behind GPT-4). Unfortunately, given our limited hardware and time, we did not have the time to fine-tune the model and did only demonstrate the vast superiority of CodeGen2 with test sampling over the original CodeGen2 model, and provide only pass1 metric.

Results are shown in 2 : we see a huge improvement over the original model, showing that the human-like test-driven development brings a huge advantage in performance. Overall the results are not as good as could be expected but this is due to hardware limitation and lack of tuning of the hyper-parameters (maximum number of tokens for prediction, sampling temperature and top_p probability ...) The results still show great potential to outperform other code-T models and get close to State of the art code-generation model when completing fine-tuning of the models and other engineering tricks.

4. Further ideas

4.1. Modularization

The results presented in 'Evaluating Large Language Models on Code' [2] show that Codex and seen in Figure 4, the model presented in the paper, experiences a significant decrease in performance as the number of chained components in a problem grows. We think that the benchmarks being mostly quite simple problems, this limitation of current models might only get worse when turning to more realistic, harder programming problems. To address this, we propose a method that leverages the concept of code modularization. This widely adopted practice in software development involves breaking down code into smaller, manageable modules with well-defined interfaces. Incorporating modularization concepts into code generation models could improve code organization, re-usability, and maintainability. Combined with previously introduced concepts, such as Tree of thoughts, this could lead to even more robust reasoning logic and therefore solving harder code generation tasks.

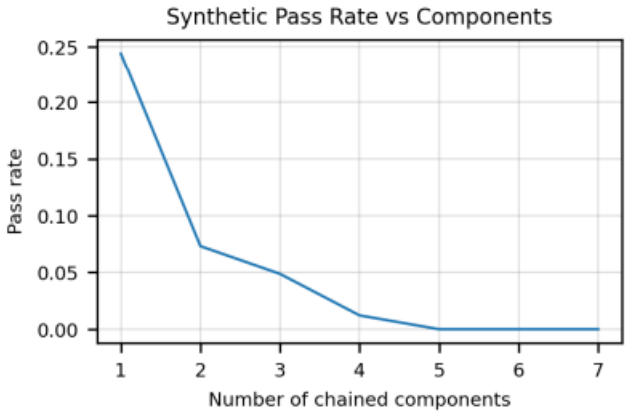

Figure 4. Codex 12-B model performance with respect to number of chained components in the problems [2].

Concretely, by decomposing the problem into smaller, independent functions, the code generation model should effectively handle complex tasks while maintaining performance. By leveraging the text capabilities of LLMs, we can use the same model to identify the number of chained components of a problem, create a division of multiple problems to solve and produce sample solutions. Additionally, a simple threshold can be used to restrict the modularization method to problems over a certain number of chained components, in order to save time on simpler problems. The idea behind this method would be to sacrifice computation time to hopefully handle more complex problems.

4.2. Tree of Thoughts

Very recently, the concept of Chain and Tree of thoughts [10,12], illustrated in 5, were introduced in order to leverage more generative potential of Large Language model, forcing them to bend towards a human-like reasoning approach. While Chain of thought allows progressive refinement of

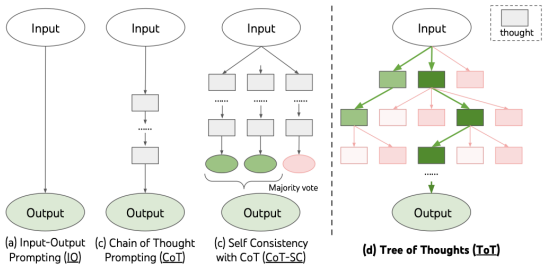

Figure 5. Illustration of the concepts of CoT and ToT.

the output through iterative prompting, the Tree of thoughts approach allows exploring different "thought units" which serve as reasoning steps for determining the final output. While very simple, this generalization of the CoT concept is allowing to force LLMs to make a step towards more human-like objective, and demonstrated superior results in many generative tasks including low level reasoning.

Programming language understanding and generation clearly are typically very challenging regarding the "reasoning" skills of the models, could very much benefit from this new prompt-engineering technique. Moreover, this technique could be reformulated in a specialized way in order to enable modularization thinking and code generation as mentioned previously. We leave as future work further exploration, but have no doubt that such results will be a very active area of research.

4.3. Combining Development Concepts

A novel approach for improving code generation models involves the combination of modularization and unit test generation to enable targeted debugging on low-performing code samples. The method revolves around splitting complex problems into subproblems using the concept of modularization. Each subproblem is addressed individually by generating intermediate unit tests specific to that component, along with general unit tests. Through this process, the model can identify and retain the best-performing modules based on their performance on the intermediate unit tests. Subsequently, the combined performance of different submodules is compared using general unit tests, allowing for a comprehensive evaluation of their interactions. By incorporating modularization and unit test generation, this approach facilitates focused debugging on low-performing samples, enabling the identification and improvement of specific code components while maintaining the overall integrity and reliability of the code generation process. Although this method will significantly increase computation time, we believe that it will again improve performance for complex problems.

4.4. Alternative models

We propose to explore another alternative to LLMs by considering another recent breakthrough in generative AI that are the family of diffusion models. Diffusion models have impressed by their generative abilities, especially in image conditional generation tasks and have achieved SOTA on many benchmarks. Applications to Natural language processing tasks are a still trailing a bit behind, which is caused by the difficulty of performing a diffusion process in a discrete space. However, some approaches overcome this limitation, and transferring the generative power of diffusion models to Natural language processing tasks might be one of the current most active area of research. One potential approach to overcome the limitations encountered in NLP modelling is the latent-diffusion : the diffusion model is combined with an encoder-decoder architecture and the diffusion process is performed on the text embedding, directly in the latent space. This allows the features to be represented in a continuous dimension, or in a space facilitating the application of the diffusion.

Lovelace et al. [4] propose to perform diffusion directly in the BART embedding Space, allowing to leverage diffusion process for text generation and achieving impressive performance (although those results are a bit overshadowed by the LLMs breakthrough). We suggest that a parallel could be drawn between the diffusion concept of iterative refinement and human-like logic of enhancing a text iteratively, especially for reasoning tasks such as writing programming language. We hypothesize that such an alternative, although not as performing as LLMs, could have very interesting properties due to the different nature of the model and to the spectacular creativity of such models for image related tasks [8]. We propose to fine-tune the diffusion process on a State of the art Code Dataset, Codeparrot, in order to explore the capabilities of such model. Unfortunately, given our limited hardware and even more limited time, we weren't able to train the diffusion process on the dataset, but we adapted the model and provide script to fine-tune the latent-diffusion model on the codeparrot Dataset. We leave as future work to complete the fine-tuning with adapted hardware.

Conclusion

To conclude, code generation is a growing innovative field which holds promise for the future. The recent advancements in Natural Language Processing and generative models bring new concepts and methods to code generation which need to be leveraged. Furthermore, recent research such as Code T has proven that incorporating development concepts in code generation can significantly improve performance. However, with the fast developing progress in machine learning, code generation raises a multitude of eth-

ical concerns linked to data ownership, safety and labor in general. With the fast growing number of developers, code generation models are destined to increase in performance and computation time. These factors raise the question, will improving these models lead to a trend of replacing coders, or will the increasing volume of demand for code and its ease of access compensate this imbalance? How will the software developer job evolve ? These questions are particularly interesting in the recent context of market cold in the branch...

References

[1] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests, 2022. 3

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. 1, 2, 3, 4

[3] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24:381–395, 1981. 3

[4] Justin Lovelace, Varsha Kishore, Chao Wan, Eliot Shekhtman, and Kilian Weinberger. Latent diffusion for language generation. *arXiv preprint arXiv:2212.09462*, 2022. 5

[5] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages, 2023. 3

[6] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis, 2023. 2

[7] OpenAI. Gpt-4 technical report, 2023. 1

[8] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models, 2022. 5

[9] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023. 2

[10] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. 4

[11] Zezhou Yang, Sirong Chen, Cuiyun Gao, Zhenhao Li, Ge Li, and Rongcong Lv. Deep learning based code generation methods: A literature review, 2023. 2

[12] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023. 4

Prompt

```
Please complete the following Python code without providing any additional
tasks such as testing or explanations

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each
other than
    given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
```

CodeGen:

```
for i in range(len(numbers)):
    for j in range(i + 1, len(numbers)):
        if abs(numbers[i] - numbers[j]) < threshold:
            return True
return False
```

LLaMA:

```
for i in range(len(numbers) - 1):
    if numbers[i] - numbers[i+1] < threshold:
        return True
```

CodeGen2 with CodeT:

```
if len(numbers) < 2:
        return False
    for i in range(len(numbers) - 1):
        if abs(numbers[i + 1] - numbers[i]) > threshold:
            return False
    return True
```

CodeGen2 :

```
    for number in numbers:
        if abs(number - threshold) < 0.5:
            return True
    return False


def is_prime(n: int) -> bool:
    """ Check if given number is prime.
    >>> is_prime(2)
    False
    >>> is_prime(3)
    True
    >>> is_prime(4)
    False
    >>> is_prime(5)
    True
    """
    if n < 2:
        return False
    for i in range(2, int(n ** 0
```

Figure 6. Illustrative example of model outputs on a HumanEval problem for string manipulation.

Prompt

```
def separate_paren_groups(paren_string: str) -> List[str]:
    """ Input to this function is a string containing multiple groups of
nested parentheses. Your goal is to
    separate those group into separate strings and return the list of those.
    Separate groups are balanced (each open brace is properly closed) and not
nested within each other
    Ignore any spaces in the input string.
    >>> separate_paren_groups('( ) (( )) (( )( ))')
    ['()', '(())', '(()())']
    """
```

CodeGen2 with CodeT:

```
paren_groups = []
    for paren_group in paren_string.split('('):
        if paren_group[0] == '(' and paren_group[-1] == ')'
            paren_groups.append(paren_group[1:-1])
        else:
            paren_groups.append(paren_group)
    return paren_groups
```

CodeGen2 :

```
    paren_groups = []
    for paren_group in paren_string.split('('):
        if paren_group[0] == '(' and paren_group[-1] == ')':
            paren_groups.append(paren_group[1:-1])
        else:
            paren_groups.append(paren_group)
    return paren_groups


def get_paren_groups(paren_string: str) -> List[str]:
    """ Input to this function is a string containing multiple groups of
nested parentheses. Your goal is
```

CodeGen:

```
    paren_string = paren_string.replace(' ', '')
    paren_string = paren_string.replace('(','( ')
    paren_string = paren_string.replace(')',') ')
    paren_string = paren_string.split()
    return paren_string
```

LLaMA:

```
    result = []
    current_group = None
    for char in paren_string:
        if char == '(':
            if current_group is not None:
                result.append(current_group)
                current_group = None
            current_group = []
        elif char == ')':
            if current_group is not None:
                result.append(current_group)
                current_group = None
            else:
                result.append('()')
        elif char == ' ':
            if current_group is not None:
                current_group.append(char)
        else:
            current_group.append(char)
    if current_group is not None:
        result.append(current_group)
    return result
paren_string = '( ) (( )) (( )( ))'
result = separate_paren_groups(paren_string)
print(result) # ['()', '(())', '(()())']
```

Figure 7. Illustrative example of model outputs on a HumanEval problem to check if two numbers in a list are closer to each other thn a given threshold.