Limitations in Code Generation LLMs: An Analysis

Isaac Thomas

1 Introduction

Task: Due to its perceivably positive impact on productivity of developers, researchers, and engineers, code generation is an extremely useful and exciting application of (large) language models. Despite promising results from GPT-4 [1] and other models on basic programming tasks, the limitations of current methods become more apparent as code generation tasks require more algorithmic reasoning. Shu et al. found preliminary evidence of this through testing GPT-4V, Code Llama, and GitHub Copilot on LeetCode problems, with the latter two models struggling to produce correct or even compilable code [12]. Older results on the less forgiving APPS competitive coding problem dataset are even worse, with egregiously low `pass@k` rates across all models tested [11]. We seek to analyze these kinds of failures in the hopes of finding patterns connecting them, which may yield architecture/data-driven improvements to LLM code generation abilities.

Model Summary: For our analysis, we use WizardCoder by Luo et al. [10], a high-ranking performer on code generation datasets like HumanEval [3] and Mostly Basic Python Problems (MBPP) [2]. WizardCoder uses a novel training paradigm called Evol-Instruct, by which it is prompted during training to incrementally add complexity to code generation training examples, yielding new and more complex auxiliary training prompts for code generation. Due to GPU memory limitations, we analyze a smaller version of this model (WizardCoder-7B).

Approach & Findings: Through manual analysis of WizardCoder-7B's failures on various code generation prompts, we found that edge case handling, task generalization, and mathematical acumen are three large pain points. In later sections, we hypothesize the causes of these systemic failures in the context of WizardCoder-7B's architectural limitations and training paradigm (including potential pitfalls of Evol-Instruct).

2 Materials & Methods

Dataset: We used the Mostly Basic Python Problems (MBPP) dataset [2], which contains nearly 1,000 crowd-sourced python coding problems requiring entry-level programming and algorithms skills. The tasks range in difficulty from simple data manipulation to optimization problems that could require more complex but common techniques like dynamic programming. Each record in the dataset consists ofa text description of the prompt and test cases that generated solutions will be evaluated on for correctness. Note that success on the test cases deos not imply that the generated code is correct. For this reason, we directly analyze flaws in the code produced by WizardCoder on this dataset.

Analysis Approach: We first sought to run a pre-trained version of WizardCoder-7B on the MBPP dataset and collect generated code for each prompt, as well as aggregate/prompt-wise pass/fail metrics; from this point we aimed to manually analyze the code generated for a subset of the prompts, with the goal of identifying systemic failures partially explainable by model architecture limitations or shortcomings of the Evol-Instruct paradigm. To meet these ends, we first obtained the WizardLM repository, which contains both the MBPP dataset and the code to evaluate WizardCoder on it. The repository conveniently contains the problems in the MBPP dataset along with instructions to evaluate a given model on the data. That being said, we did have to convert the supposedly preprocessed MBPP dataset to the `jsonl` format expected by scripts that aggregate the model's predictions. We then obtained `TheBloke/WizardCoder-Python-7B-V1.0-AWQ` (low-bit quantized model for lower memory/faster inference) from HuggingFace Hub and ran inference with

`mbppplus_gen_vllm.py`. We then collected the generated responses using `mbppplus_process_preds.py` and evaluated the quality of the results with `evalplus` [8], a library (with an executable version) for computing pass metrics on both HumanEval and MBPP datasets. The code generation task data and WizardCoder's solutions to them are available in `report/data`. The detailed steps of this whole process are available in the WizardLM repository. We link the Python notebook we used to generate and aggregate WizardCoder-7B's predictions on MBPP at the end of this analysis.

Aggregate Performance: We attained `0.61 pass@1` on the original MBPP dataset and `0.46 pass@1` on the extended MBPP Plus dataset (results may vary slightly). While these results were consistent with the authors' claims about small model performance, it suggests that WizardCoder faces some limitations in code generation for more difficult challenges. This result is not surprising though; the version of WizardCoder used has only $\sim$6.75 billion parameters, and larger versions of WizardCoder have attained significantly higher `pass@1` rates on MBPP. That being said, one would naturally inquire about the challenge complexity threshold past which systemic failure is at play. We anticipate that this threshold is not one-dimensional - in particular, that there are likely multiple mostly independent categories of task difficulty. We analyze failure cases from some of these proposed categories in the next section.

3 Results

Failure Categorization: Through manual analysis, we found many cases where WizardCoder failed to account for edge cases, craft a sufficiently general solution function, and nontrivially apply even basic mathematical relationships. We elaborate on each of these three non-exhaustive failure categories and the corresponding flaws in WizardCoder-7B's responses to prompts in MBPP.

Edge Case Handling: On many occasions, WizardCoder struggles to handle relatively simple edge cases in various MBPP code generation tasks. By "edge-case" we mean an input one needs to handle with reasoning not fully contained by the general approach to the problem in question. We detail some of these instances now. Consider problem 797, where the model is asked to compute the sum of all odd natural numbers in a given range $[l, r]$, and WizardCoder's solution below:

```
def sum_in_range(l, r):
    return sum(range(l, r+1, 2))
```

When l is even, we need to exclude l by using the range $[l+1, r]$ instead; but the model does not write code that accounts for this case, instead using the same range regardless of the parity of l. We see similar issues in WizardCoder's solution to problem 784, where the model is asked to determine if all characters in a given string are the same:

```
def all_Characters_Same(string):
    return len(set(string)) == 1
```

The model's generated code fails to account for the case where `string` is empty, and it would incorrectly return `false` when the statement is vacuously true. Such behavior also arises in problem 754, where one is given three lists and asked to find the indices at which all three lists have equal elements at that index:

```
def extract_index_list(l1, l2, l3):
    common_elements = []
    for i in range(len(l1)):
        if l1[i] == l2[i] == l3[i]:
            common_elements.append(l1[i])
    return common_elements
```

The code fails to check explicitly if the index in question is out of bounds for some lists but not others, which will cause a run-time error on lists of differing lengths. One could also view this as a failure to generalize for variable length lists. Similar issues exist in WizardCoder's solutions to problems (including but not limited to) 735, 755, and 762. Note that there are no immediate commonalities between the example prompts mentioned beyond the fact that they all have some boundary cases that may need to be explicitly addressed outside of the general solution. Such is the difficulty with detecting and handling edge cases automatedly, as there is little to no structure relating them across an arbitrary group of computing problems. That beind

said, it's possible that Evol-Instruct was not applied specifically to address edge cases during the fine-tuning process; this could imply that there is room for improvement, and we discuss it in a later section.

Generalization: On the opposite end of the spectrum, WizardCoder also struggles with creating sufficiently general solutions to many code generation tasks (and possibly accurately generalizing from base cases). One of these instances is the generated solution to problem 9, which asks one to write a Python function that determines the minimum positive number of rotations (cyclically shift all characters to the right by one) of an input string `s` that will `s` again:

```
def find_Rotations(s):
    n = len(s)
    if n == 0:
        return 0
    elif n == 1:
        return 0
    elif n == 2:
        if s[0] == s[1]:
            return 0
        else:
            return 1
    else:
        for i in range(n):
            if s[i] == s[0]:
                return i
    return n
```

It's worth noting that WizardCoder attempted to handle edge cases, like when `len(s)` is 0, 1, or 2. But in the general case, the model simply finds the minimum index i for which the i-th character is equal to the first character. This is obviously not correct, as when $i = 0$ this condition will pass, causing the code to always output 0. Interestingly enough, it seems that this was an attempt at generalizing the code written for the case where `s` is of length two. We see similar behavior in WizardCoder's solution to problem 476, which asks one to compute the sum of the smallest and largest values in an array:

```
def big_sum(arr):
    return arr[0] + arr[-1]
```

Again, note how the model does not generalize to arbitrary lists and instead writes a solution that only works with sorted lists. This is particularly interesting given that the example in the prompt is `[1, 2, 3]`, and it lends more evidence to the claim that WizardCoder potentially does not learn all it can from evolved versions of simpler tasks generated via Evol-Instruct. Similar behavior arises in WizardCoder's solution to problem 439, which asks one to concatenate a list of integers into the corresponding number:

```
def multiple_to_single(lst):
    result = 0
    for num in lst:
        result = result * 10 + num
    return result
```

Again, WizardCoder produced a solution that would only work for a list of 1-digit numbers, indicating a generalization failure seen in previous problems. Other instances of this can be observed in problems 69, 125, and 259. The failures to generalize described here could result from the flawed nature/application of Evol-Instruct during training/fine tuning. The WizardCoder authors describe the prompt used in Evol-Instruct as follows:

```
Please increase the difficulty of the given programming test question a
bit.
You can increase the difficulty using, but not limited to, the following
methods:
{method}
{question}
```

where `method` was one of the five types given below:

```
Add new constraints and requirements to the original problem, adding
approximately 10 additional words.
```

```
Replace a commonly used requirement in the programming task with a less
common and more specific one.
If the original problem can be solved with only a few logical steps,
please add more reasoning steps.
Provide a piece of erroneous code as a reference to increase
misdirection.
Propose higher time or space complexity requirements, but please refrain
from doing so frequently.
```

There are no prompts to remove constraints from existing code generation challenges and make them more general - for instance, "remove the requirement that the input to the programming task is sorted". Thus, there could be room for improvement in engineering Evol-Instruct prompts to facilitate better generalization by WizardCoder. We discuss this more in a later section.

Mathematical Acumen: As one could guess based on the aforementioned failures, WizardLM struggles with even basic applications of simple mathematical relationships. One of these is WizardCoder's solution to the problem 72, which asks one to determine if a number x can be represented as the difference $a^2 - b^2$ of two square numbers:

```
def dif_Square(n):
    for i in range(1, int(n**0.5)+1):
        for j in range(1, int(n**0.5)+1):
            if i**2 + j**2 == n:
                return True
    return False
```

It's clear here that WizardCoder-7B is attempting to apply methods from its fine-tuning data for determining if x is a *sum* of squares, but it fails to make the conceptual conversion from sum of squares to difference of squares. One may conclude that WizardCoder has no easily accessible perception of the difference between these two expressions at all. If we consider "sum", "difference", "+", and "-" as separate tokens for instance, it could be that WizardCoder does not have latent representations which sufficiently separate "sum" and "+" from "difference" and "-". We see a similar pattern with WizardCoder's solution to problem 233, which asks one to find the *lateral* surface area of a cylinder:

```
def lateralsuface_cylinder(radius, height):
 return 2 * math.pi * radius * height + 2 * math.pi * radius ** 2
```

Again, it appears that WizardCoder fails to convert what would be considered a "standard" formula into something suitable for the task at hand. This could be attributable to the model not assigning enough attention to the contextual relationships between tokens/token sets like "lateral" and "surface area". Signs of similar behavior emerge in WizardCoder's solution to problem 14, which asks one to compute the volume of a triangular prism:

```
def find_Volume(length, width, height):
    return length * width * height
```

Again, WizardCoder naively attempts to use the well-known relationship between a rectangular prism's side lengths and its volume. But it fails to adapt this formula into something that would work for a triangular prism on the basis of it being half of a rectangular prism. Again, this could likely be attributed to a lack of importance assigned to contextual relationships between tokens like "triangular" and "prism". A sufficient amount of attention paid to this by the model could allow it to differentiate and potentially discern a relationship between the volume of a triangular prism and that of a rectangular prism, which is necessary for the generation of a correct solution. Though somewhat humiliating, all of these quantitative shortcomings may not be surprising; WizardCoder was not jointly fine-tuned on math problems and code generation problems (there actually is a separate model for the former from the same group called WizardMath [9]), and in general, quantiative reasoning is a well-known struggle for large language models. Exceptions to this phenomenon like Minerva [6] had to be fine-tuned specifically on research papers and other formally/mathematically rigorous data just to reach 78% accuracy on math grade-school word problem datasets like GSM8K [4]. So it may be delusional to expect even basic mathematical aptitude from a much smaller model tuned only on code generation tasks. That said, the Evol-Instruct paradigm could be used here to fuse aspects of math-related and coding-related prompt evolution. We discuss ideas of this nature shortly.

4 Discussion

Architectural Limitations: As Kaplan et al. have studied [5], LLM model test loss scales inversely with model parameters. Furthermore, the authors report that larger models need to process fewer tokens to attain the same test loss. This is no surprise, as a larger number of parameters grants access to a larger function space, allowing for modelling of more relationships between tokens or richer modelling of relationships between existing tokens. For this reason alone, it's safe to say that all else fixed, choosing a larger model would have resulted in better performance and fewer systemic errors than what was encountered with WizardCoder-7B. Additionally, the model we used was a quantized model meant for low-precision, computationally cheap inference. Using a full-precision model could have increased aggregate `pass@1` rates as well, although systemic errors may well have persisted. However, such improvements are not always feasible since researchers and curious users are limited by computing power and GPU memory. It's likely far more beneficial to analyze the limitations of the training approach used, since innovations on this front (especially with Evol-Instruct) could introduce significant improvements even with small models.

Limitations of Evol-Instruct: Recall the methods that WizardCoder is allowed to use in prompt evolution:

```
Add new constraints and requirements to the original problem, adding
approximately 10 additional words.
Replace a commonly used requirement in the programming task with a less
common and more specific one.
If the original problem can be solved with only a few logical steps,
please add more reasoning steps.
Provide a piece of erroneous code as a reference to increase
misdirection.
Propose higher time or space complexity requirements, but please refrain
from doing so frequently.
```

This approach is limited by the quality of the prompts, which likely must strike a delicate balance: too general, and the model will not apply the methods to evolve instructions properly; too specific, and the model may evolve prompts in a manner that adversely impacts generalization to other coding tasks. This would justify trying multiple different sets of methods of varying generality during the fine tuning process, and possibly performing some kind of ablation study to determine evolution instruction quality. Additionally, the Evol-Instruct paradigm is bound by the model's ability to learn from the evolved prompts. This takes us back to model-size bottlenecks which we cannot do much about beyond increasing model size or developing some novel, significantly more learning-conducive architectural network component.

5 The Way Forward

Here, we outline avenues of future research based on our findings and their speculated causes. Most of these ideas center around the training process and ideate variants/novel applications of Evol-Instruct.

Targeted Evol-Instruct: As mentioned before, it could be worth methodically trying different instruction evolution methods in the evolution prompts used by Evol-Instruct. One example of this approach involves deliberately evolving prompts centered around the systemic errors described in previous sections. For instance, suppose we want to fine-tune WizardCoder's base model in hopes of minimizing the chance that it creates a solution missing an edge case. We would then include among `methods` a prompt like "modify the programming task so it contains an edge case not handled by the general approach". One may ask how WizardCoder would evolve prompts effectively without already knowing how to handle edge cases; to this end, it could be useful to annotate training prompts with edge cases so that the model can discern that they are initially prioritized for evolution, then prioritize these tasks during the prompt evolution rounds.

Joint Training on Math/Coding Prompts: It's possible that jointly fine-tuning WizardCoder's base model on math problems and coding problems could produce transferable improvements to both sets of tasks. This could boost the performance of math-intensive coding challenges. Preliminary work by Toshniwal et al. [13] explored this initiative to an extent using a tuning dataset consisting of math problems allowing for coded solutions. This approach attained competitive `pass@1` rates of 0.81 on GSM8K and 0.416

on the MATH dataset. Exploring an approach in this vein could remedy the quantitative reasoning failures observed in WizardCoder's generated solutions.

Bottom-Up Training on Modular Prompts: Practical coding tasks are largely modular; they can be split up into easier subtasks, and then the solutions can be combined to solve the original task. Li at al. made use of this observation in MoTCoder [7], which generates code for prompt decompositions in a highly similar fashion. The use of this approach, however, begs the following question: why not just gather many common subtasks and train on useful combinations of them in a bottom-up fashion? This bottom-up approach could involve evolving prompts to make WizardCoder good at common subtasks, then evolving subtasks by combining them to obtain increasingly complex prompts. This approach could improve edge case detection and generalization if every coding challenge can be represented by its constitutent subtasks; however, it may fall short for inference on coding challenges that are not inherently modular or have dense subtask dependencies.

Input Fuzzing/Mutation for Edge Case Training: In light of WizardCoder-7B's seemingly systemic edge case detection failures, one may wonder if such problems could have been averted if the model repeatedly trained on the same code generation task for some time but with mutating inputs. This could be beneficial, as the test cases used in fine tuning may not cover the full range of cases the model would need to cover in a completely correct solution. Concretely, this would look as follows. Suppose we take the problem of determining if all characters in a string are the same (assume it's in the training/tuning dataset). We would repeatedly mutate the input to this prompt by adding/deleting elements from the string, which could yield edge cases like an empty input; we would then compute the loss over each instance of the coding problem, so that the model could learn that its original approach does not handle the empty string edge case. It does not seem that such an approach has been covered extensively in the literature, so this could be a risky but potentially lucrative research avenue.

6 Code & Data Availability

WizardCoder-7B Model Evaluation: The code for evaluating WizardCoder-7B on the MBPP dataset can be found in the notebook linked here. It contains all of the bash scripts and python code necessary to download all the relevant repositories/models/data, create the appropriate environments, and run the WizardCoder scripts for evaluation on MBPP and aggregation of generated code per prompt. Note that one may attain slightly different `pass@1` rates, but the numbers should not vary significantly.

Data from Our Analysis: The code generation responses and pass/fail data can be found in this repository under the directory `report/data`. It contains aggregate results from WizardCoder-7B predictions, as well as the code generated for each MBPP problem (under `report/data/wizardcoder-results-by-task`). Note that multiple generated responses annotated with "fail" seem (and likely are) correct upon inspection. There may be discrepancies surrounding what `evalplus` deemed a "passing" solution, but this should not matter much as we picked failures to analyze based on manual code review and not the evaluation result annotation.

7 Conclusion

Code generation LLMs have exciting applications in the present and future. To fully realize their potential in this arena, we aimed to advance understanding of some categories of code LLM failures (edge case detection, sufficiently generalized solutions, quantitative reasoning), potential causes related to model architecture / training paradigms, and what can be done in the future to improve. The insights from this analysis should serve as food for thought regarding how LLMs can fail at deceptively simple tasks, why they may fail in this manner, and future research that will increase the robustness of such models to increasingly sophisticated code generation tasks.

8 Acknowledgements

I exclusively made use of class notes / resources, code generation LLM literature, and WizardLM repository documentation for this analysis.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[4] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021.

[5] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.

[6] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reasoning problems with language models, 2022.

[7] Jingyao Li, Pengguang Chen, and Jiaya Jia. Motcoder: Elevating large language models with modular of thought for challenging programming tasks. *arXiv preprint arXiv:2312.15960*, 2023.

[8] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[9] Haipeng Luo, Qingfeng Sun, Can Xu, Pu Zhao, Jianguang Lou, Chongyang Tao, Xiubo Geng, Qingwei Lin, Shifeng Chen, and Dongmei Zhang. Wizardmath: Empowering mathematical reasoning for large language models via reinforced evol-instruct, 2023.

[10] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

[11] paperswithcode.com. Papers with code - apps benchmark (code generation).

[12] Peng Shu, Huaqin Zhao, Hanqi Jiang, Yiwei Li, Shaochen Xu, Yi Pan, Zihao Wu, Zhengliang Liu, Guoyu Lu, Le Guan, et al. Llms for coding and robotics education. *arXiv preprint arXiv:2402.06116*, 2024.

[13] Shubham Toshniwal, Ivan Moshkov, Sean Narenthiran, Daria Gitman, Fei Jia, and Igor Gitman. Openmathinstruct-1: A 1.8 million math instruction tuning dataset. *arXiv preprint arXiv:2402.10176*, 2024.