# Reproducing MathPrompter: Mathematical Reasoning using Large Language Models

**Aron Winkler**
aron.winkler@student.uni-tuebingen.de

## Abstract

In recent years, breakthroughs in natural language processing (NLP) and language modelling in particular have enabled researchers and consumers to tackle mathematical problems through language models. It has become clear, however, that language models do not deal well mathematical problems in zero-shot contexts, even if those problems seem very easy to humans. Recently, a series of various research contributions has progressed the state of the art considerably, especially through the use of very heavy large language models (LLMs). Following these developments, Imani et al., 2023 reached quasi-SOTA performance with despite using relatively more limited resources. These authors' clever use of prompt engineering and result validation with simple computational resources are the target of this replication attempt. Code for this work can be found on GitHub[1].

## 1 Introduction

Since their introduction to NLP and their swift increase in popularity in recent years, Large Language Models (LLMs) have been applied to a wide set of domains. Due to their affinity with zero- and few-shot learning contexts (Brown et al., 2020), LLMs have been experimented with in the context of mathematical problems defined in natural language, with varying degrees of success.

LLMs have been observed to struggle even with simple arithmetic when they saw large popularity and availability with GPT3 through ChatGPT[2]. When prompted with a mathematical question, the model would often dream up an incorrect solution, even when to a human the task would have seemed trivial. More quantitative evaluations of the phenomenon have been carried out as well.

The MultiArith dataset (Roy and Roth, 2015) has been used in various studies for the purposes of gauging the performance of LLMs and associated prompting strategies on simple arithmetic problems. It provides simple mathematical exercises and the solutions to them, with all solutions consisting of a single integer. LLMs have had their mathematical prowess evaluated on the dataset, however zero-shot techniques have been reported to only achieve around 17% accuracy on the dataset (Kojima et al., 2023). Such results lend credence to the impression that LLMs struggle greatly when tasked with mathematical reasoning, at least in a zero-shot setting.

Kojima et al., 2023 contributed a chain-of-though (CoT) approach to formulating mathematical prompts, which breaks down the problem into a series of intermediate steps via an ingenius double-prompting strategy:

1. The LLM is promted with the mathimatical problem, and the task to break it down into intermediate steps

2. The LLM is prompted again with the intermediate steps from the previous stem, and the task of computing the answer

This simple process of separating the task into two distinct steps (labelled "reasoining extraction" and "answer extraction" in the original) boosts accuracy over MultiArith to 53%.

Imani et al., 2023 improved further, reaching as far as 92.5% accuracy on MultiArith. Though some of this is likely due to a model upgrade (Kojima et al., 2023 used OpenAI's *text-davinic-002*, while Imani et al., 2023 used *text-davinci-003*, which is itself of obsolete at the writing of this report), much of the increase is likely due to the new strategy being employed.

[1] https://github.com/waron97/enlp_mathematical_reasoning

[2] https://openai.com/

## 2 MathPrompter

The improvement in accuracy over previous studies was achieved with MathPrompter (Imani et al., 2023), a system that shares some similarities with CoT prompting, but forgoes attempting to get a direct solution from the model. MathPrompter's execution steps can be broken down as follows:

1. MathPrompter receives a question Q as input, then maps it to a new prompt Qt, resulting from Q having its numeric values mapped to variables. For example *"May has 10 apples, but sells 5. How many apples does May have now?"* becomes *"May has A apples, but sells B. How many apples does May have now?"* The original variable values {A: 10, B: 5} are also recorded.

2. Qt is mapped to two prompts (P1, P2) to be forwarded to the LLM, one eliciting a python function and the other a mathematical expression:

   P1: "{Qt} Write a mathematical equation and generate the answer format starting with 'Answer = '"

   P2: "{Qt} Write a python function that returns the answer"

3. The underlying LLM is prompted with P1 and P2 individually, returning (hopefully) something along the lines of

   ```
   def foo(A, B):
       return A - B
   ```

   for the Python-function prompt.

4. The LLM completions are passed to Python's *eval* method, with the problem variables initialized to random values. If both prompt completions yield the same result for the same random values, then the completion is accepted.

5. The accepted solution from step 4 is again used with *eval*, with the problem variables set to the original values ({A: 10, B: 5} from step 1). The whole process is repeated 5 times, i.e. MathPrompter computes 5 "accepted" results, and reports the majority solution to the user.

A flowchart of MathPrompter can be viewed in the original paper for a more easy-to-digest representation.

## 3 Reproduction process and results

The execution cycle illustrated above is meant to reflect fairly accurately the information provided about the functioning of MathPrompter in the original paper. Since it is of course not possible to provide all nuances in written form, a sitation naturally arises in which not all implementation details are clearly reported. The authors also did not publish any code, so such questions are left for the reproducers to explore. What follows here is a general overview of the gaps that this reproduction attempted fill in, which may differ from the original implementation.

Firstly, referencing step (1) from section 2, mapping Q to Qt involves extracting variables from the mathematical prompt - which can be a tricky task if the objective is a truly general solution. However, for the context of MultiArith, a simple regular expression targeting groups of digits was sufficient, although it is stil necessary to pay attention to edge cases such as the expression "mp3 player", where the script had initially identified a variable. Real-world implementations of MathPrompter should, however, dedicate more developer attention to handling edge cases in variable extraction.

Moving on to steps (2) and (3), a point of uncertainty is represented by the fact that Imani et al. 2023 do not delineate precisely how the final prompts to the LLM are derived. In summary, if after evaluating the two completions for P1 and P2, the resulting expressions fail to converge to the same result given the random variables (e.g. in the case of "A + B - C" and "A - B / C"), it is unclear whether MathPrompter exits with failure or attempts to generate new completions by prompting the LLM again - and if so, how many re-prompts is it allowed per cycle. In this study, the instructions were taken to mean that in case of non-convergence, MathPrompter is allowed to request new completions from the LLM in step (3). For the purposes of this study, the limit to this has been set to 5 attempts - after 5 attempts, if no converging completion pair is found, the system declares no result for that cycle (this only concerns step (4), but remember that everything from step (2) though (4) is repeated an additional 5 times).

Another point of contention is the utilization of python's *eval* function during result evaluation, which is accessed in both steps (3) and (4). The *eval* method is a useful but not omnipotent tool to check code output - among other things, assign-

ments and function definitions are not allowed. It's not stated to what extent the original implementation worked around these limitations. Particuarly problematic with regard to this are multiline python functions returned by the LLM: functions with only a return statement as their body are simple to map to a format that *eval* can process, but multiple intermediate steps before a return statement are not. In that scenario, it would probably be more efficient to not rely on *eval* at all and instead set up a proper execution environment for the function, but this would require significant changes to the experiment setting. As a consequence, this reproduction of the MathPrompter execution cycle ignored errors arising from functions consisting of more than a return statement, declaring non-convergence of completions in this scenario instead.

Lastly, it's worth mentioning that getting access to the MultiArith dataset is at the same time straightforward and difficult. When searching for MutliArith on a search engine, multiple results come up - in that sense, acquiring the data is in itself trivial. However, these sources of data do not come from the original publishers of the dataset, and there is no real proof of authenticity on them. Furthermore, the paper which is quoted in other research when attributing MultiArith (Roy and Roth, 2015) contains no reference to this dataset in its body. Having an official release of MultiArith would be desirable for future work down the line.

For further clarification of the practical reproduction steps taken in this experiment, it should be sufficient to refer to the GitHub repository linked above.

With only the aforementioned assumptions implemented, MathPrompter achieves 73.83% accuracy on MultiArith, almost 20 percentage points less than the reported 92.5%. This drop could be attributed to a number of factors, some of which are explored in section 4, but here it is worth pointing out one that is fairly trivial to solve: integer division.

After manual review, it was observed that the prompts eliciting a Python function from the LLM often contained integer division (Pythons "//" operator), whereas the simple mathematical formulations of the solution did not. With the original variable values, this would not have caused an issue since all gold solutions are integers - however, with solution validation being carried out with random values, this difference often meant different outputs for completions that were identical save for integer division. It is possible that the original study did not have this issue (OpenAI models are known to be changed under the hood, even though both the original study and this reproduction explicitly used *"text-davinci-003"*), or that it was regarded so minor that it did not get a mention in the paper.

By replacing integer division operators in the Python completions with simple division operators, MathPrompter is recorded to have an accuracy of 82.3% in this experiment, which is closer to the originally reported value.

A futher after-the-fact analysis was to check whether not requiring P1 and P2 to converge to the same prompt would make a difference in the results (previosuly, answers where both prompts failed to agree on a single value were simply discarded). The main motivation for this was that in a real-world scenario, some use cases could prefer showing a partial solution to the end user, even if confidence in its correctness was low. To confirm this, all unsolved questions left after accounting for integer division were re-examined by allowing a valid result even if only one of the two prompts lead to its computation. This, however, did not lead no any substantial change in the results: at most 1 or 2 questions benefited from this alteration across the runs. Crucially, this should have allowed the mathematical expression to provide a result when the Python solution involved multiple lines. Interestingly, this means that when the Python solution had intermediate steps (and was thus unprocessable), the mathematical expression from the other prompt was also wrong.

## 4 Final observations

Wrapping up on the issue of multiline Python solutions returned by the LLM, which were a discussion point at several points of this work, superficial analysis revealed that, at least in this experiment, not being able to gain results from them did not impact the results, since the corresponding simple mathematical formulations were also incorrect. This remains nonetheless a weak point of MathPrompter as it was described by Imani et al., 2023, since defining solutions in a programming language naturally aligns with breaking complex problems down in more steps than a single return statement can cleanly encapsulate.

Aside from mapping integer division to standard division, there are therefore no other low-hanging fruit that would allow to drive the reproduced ac-

curacy closer to the reported accuracy of MathPrompter.

It is also a possibility that this is an effect of the "dumbening" of GPT models that has been generally reported by both the public and researchers (Chen et al., 2023). Though perhaps a factor, such a steep drop in task accuracy might be too big a difference to be attributed to this only - more likely, a combination of underlying model changes and some details having been overlooked in this reproduction attempt caused the difference in performance. It is, at any rate, not hard to imagine that a few minor improvements could get MathPrompter performance above 90% - in this sense, this study can be understood as being a successful replication attempt.

In addition to the original paper, this study also measured two other factors: average number of LM prompts made per question, and average question execution time. While real-world scenarios might do without the amount of re-prompting that was done in this experiment, it is still relevant to know for real-world use cases what number of API calls might be made in the standard case.

For the average number calls, the measured value was 30.06. While 30 model prompts per question is exceedingly high, it is worth considering that real-world scenarios might not be set up in a way that allows up to 50 API calls for a single question (which was the case in this study). Still, developers should be careful when using MathPrompter by setting up proper exit conditions, in case the model fails to output a well-formed solution - at least within a reasonable number of attempts.

Average execution time for questions was 26.84 seconds in this experiment. As before, real-world implementations might be set up to exit earlier in case of failure to generate a valid result, bringing this number to a more manageable range. It is however paramount that proper attention is payed to balancing execution time and quality, since waiting 26 seconds on average for a prompt is too slow even for LLM standards.

Lastly, addressing the lack of an interactive application with which to test out MathPrompter has also been the target of this experiment. The Github repository contains instructions for how to set up such an environment.

## 5 Discussion

In summary, this reproduction attempt failed to achieve the reported 92.5% accuracy of MathPrompter on MultiArith, falling short at around 82.3%. This is the best result this study achieved, however it is not unimaginable that further optimisations that didn't get a mention in the original paper would boost accuracy to the reported levels.

Aspects of MathPrompter that should interest developers but that were not discussed by the originals authors were also highlighted here, such as expected API calls per question and execution time. Publishing such results should enable implementers to make more sound configuration options, both for customer satisfaction, and to lessen the strain on their wallets.

Using LLMs to solve mathematical problems, and to process numeric values in general, is an interesting and active field of research that still has avenues for growth. While early results are promising, one should keep in mind that MultiArith exercises are all of the sort that humans would expect to see in the first years of elementary school, therefore an above 90% accuracy on it should be more the default than something to be in awe of. Perhaps, in the future, research could consider venturing into the area of more complex mathematical problems, as there might be more to gain there.

## References


Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners.

Lingjiao Chen, Matei Zaharia, and James Zou. 2023. How is chatgpt's behavior changing over time?

Shima Imani, Liang Du, and Harsh Shrivastava. 2023. MathPrompter: Mathematical reasoning using large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 5: Industry Track)*, pages 37–42, Toronto, Canada. Association for Computational Linguistics.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2023. Large language models are zero-shot reasoners.

Subhro Roy and Dan Roth. 2015. Solving general arithmetic word problems. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1743–1752, Lisbon, Portugal. Association for Computational Linguistics.