Czech
Technical University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Large Language Models for Numerical Queries

Bachelor's Thesis of
Timofej Kiselev

Supervisor: **Ing. Jan Drchal, Ph.D.**
Field of study: **Open Informatics**
Subfield: **Artificial Intelligence and Computer Science**
May 2024

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Kiselev Timofej** Personal ID number: **507393**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence and Computer Science**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Large Language Models for Numerical Queries

Bachelor's thesis title in Czech:

Velké jazykové modely pro numerické dotazy

Guidelines:

Experiment with LLM models solving numerically oriented queries.
1. Review state-of-the-art methods and datasets to solve numerically oriented tasks (e.g., word problems).
2. Collect selected datasets. Extend them if needed to support symbolic expression generation.
3. Perform experiments using selected LLMs, evaluating the quality of the generated symbolic expression.

Bibliography / sources:

[1] Ronald, Krist. Metody zpracování přirozeného jazyka pro řešení slovních úloh. MS thesis. České vysoké učení technické v Praze. Vypočetní a informační centrum., 2022.
[2] Storks, Shane, Qiaozi Gao, and Joyce Y. Chai. "Recent advances in natural language inference: A survey of benchmarks, resources, and approaches." arXiv preprint arXiv:1904.01172 (2019).
[3] Lewkowycz, Aitor, et al. "Solving quantitative reasoning problems with language models." Advances in Neural Information Processing Systems 35 (2022): 3843-3857.
[4] Yu, Longhui, et al. "Metamath: Bootstrap your own mathematical questions for large language models." arXiv preprint arXiv:2309.12284 (2023).

Name and workplace of bachelor's thesis supervisor:

Ing. Jan Drchal, Ph.D. Artificial Intelligence Center FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **19.01.2024** Deadline for bachelor thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

Ing. Jan Drchal, Ph.D. Supervisor's signature	prof. Dr. Ing. Jan Kybic Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgements

I would like to thank my supervisor, Ing. Jan Drchal, Ph.D., for all the support and advice regarding with this thesis. I would also like to thank Ing. Herbert Ullrich for his help with the LaTeX editor and the CTUstyle template used in this thesis. I am also grateful to my family for taking care of both my physical and mental well-being during my studies. Finally I would like to thank my friend Alexandr Belčenko for the grammar-focused proofreading of this thesis.

Declaration

Author statement for undergraduate thesis:
I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date 24. May 2024

.....
Signature

List of AI tools used:
DeepL Write for the grammar check, and improved word choice through the selection of more suitable synonyms. ChatGPT for editing graphical outputs such as tables and figures, visual changes only. ChatGPT for the assistance with programming of regex and parsers. ChatGPT for the initial dataset reformatting. GAIR/Abel-7B-001 for an experiment. MetaMath-Mistral-7B for the fine-tuning of generator model and experiments. Longformer allenai/longformer-base-4096 for fine-tuning the verifier model and experiments.

Abstract

This thesis examines various methods that use code generation as a means of solving mathematical word problems. It presents a novel format that uses symbolic expressions and reformats existing datasets into the proposed format. It then trains a model that uses this format and evaluates its results, experimenting with various methods to improve its performance.

Keywords: Large language models, LLM, Math Word Problem, MWP, SymPy, Symbolic expressions, code generation, verifier model

Supervisor: Ing. Jan Drchal, Ph.D.

Abstrakt

Tato práce zkoumá rozlišné metody využívající generace kódu k řešení matematických slovních úloh. Prezentuje nový formát využívající symbolických výrazů a přepisuje existující datasety do tohoto formátu. Následně trénuje model k používání tohoto formátu a vyhodnocuje jeho výsledky a provádí experimenty s různými metodami, které slouží k zlepšení výkonu.

Klíčová slova: Velké jazykové modley, LLM, Matematické slovní úlohy, MWP, SymPy, Symbolické výrazy, generování kódu, ověřovací model

Contents

Figures

Tables

Chapter 1
Introduction

Mathematical reasoning and code generation are currently some of the most difficult challenges for Large Language Models. These tasks require a certain degree of general external knowledge, language understanding, and multi-step reasoning capabilities, each of which is also challenging. As a result, reasoning benchmarks are the most commonly used as a way to compare state-of-the-art models.

Mathematical reasoning encompasses a range of domains, including arithmetic, math word problems, geometry, and theorem proving [Ahn, Janice et al., 2024]. This work focuses on the math word problems category, with the two most prominent benchmarks being GSM8K and MATH. Currently, many works solve math word problems by generating executable code. This approach circumvents the issues associated with arithmetic by executing the generated code to perform calculations, rather than relying on a language model for these computations. Moreover, it has been demonstrated that fine-tuning language models on code and mathematics, enhances the performance of language models by enhancing language comprehension mechanisms such as entity tracking [Prakash et al., 2024].

This work proposes a similar approach, whereby the solution is generated in the form of alternating blocks of natural language reasoning and executable SymPy code (a Python library for symbolic mathematics). The proposed approach yields output that is readily comprehensible, explainable, and convertible into various formats, including LaTeX expressions. The solution can verify the answer provided in textual form, which can be created by the same model, another model, or human.

Question: The difference between the ages of two friends is 2 years. The sum of their ages is 74 years. Find the age of the older friend.
Answer: Let's assume the age of the older friend is x years. Then the age of the younger friend is (x-2) years. The sum of their ages is x + (x-2) = 74 years. Combining like terms, we get 2x - 2 = 74. Adding 2 to both sides, we get 2x = 76. Dividing both sides by 2, we get x = 38. Therefore, the age of the older friend is 38 years.
38 The answer is: 38
Solution: [|Text|] Let's assume the age of the older friend is x years.\n Then the age of the younger friend is (x-2) years. \n The sum of their ages is x + (x-2) = 74 years. \n Combining like terms, we get 2x - 2 = 74. \n Adding 2 to both sides, we get 2x = 76. \n Dividing both sides by 2, we get x = 38. \n Therefore, the age of the older friend is 38 years. \n #### 38 \n The answer is: 38 [|EndOfBlock|] [|Sympy|] # Define symbol \n x = sp.symbols('x') [|EndOfBlock|] [|Text|] Now, let's set up and solve the equation for x. [|EndOfBlock|] [|Sympy|] # Set up the equation \n x = sp.symbols('x') \n sp.Eq(x + (x - 2), 74) \n # Solve for x \n solution = sp.solve(equation, x) \n # Solve for x \n [|EndOfBlock|] [|Text|] Extract the numerical solution for x. [|EndOfBlock|] [|Sympy|] # Extract the numerical solution \n *age_of_older_friend* = solution[0] \n *print(age_of_older_friend)* [|EndOfBlock|]

Figure 1.1: Example of the proposed format.

1.1 Math Word Problems - MWPs

Math Word Problems (MWPs) are mathematical problems presented in natural language, usually in the form of a question that requires an answer. Therefore MWPs require more reasoning skills than pure arithmetic problems written in mathematical notation. MWPs require the ability to identify crucial background information in the given question, to discern relationships and connections among the available data, to ascertain what the question is asking, and to perform reasoning steps that would lead to an answer. Furthermore, MWPs frequently require some degree of general knowledge that is not explicitly stated in the question but is necessary for the solution. For instance, the number of days in a week or an approximation of the value of π. Solving these problems is often a multi-step process in which new information is obtained by performing mathematical operations on existing data.

1.2 Formats

There are a number of different formats in which MWPs can be presented. All of them consist of a question and an answer to that question. The answer can take different forms, including a single numeric value, a step-by-step solution in natural language, or a series of mathematical operations performed to reach the final answer. Given the difficulty of evaluating intermediate steps in reaching a solution, the output answer is generally considered correct if its final answer value matches the provided value, usually to some allowed margin of error (for example, 1e-3). This approach allows for the possibility of automatic evaluation. To enable automatic evaluation the provided answer almost always has the final answer value emphasized in a way that allows it to be algorithmically extracted. The presence of supplementary information that serves to guide the solving process can be considered a separate format. For instance, when the equation is accessible, it can be presented in the format Question-Equation-Answer, or when the step-by-step solution is available Question-Rationale-Answer [Ahn, Janice et al., 2024]. Such differences may be reflected in the format of the particular dataset or benchmark, should the authors of the latter choose to do so.

Other formats of significance to this work are those that utilise executable code. This offers a variety of possible formats such as Question-Code-Answer, Question-Rationale-Code-Answer, but the most notable for this work is a series of short interleaving blocks of code and text in natural language. This format is used in different ways by GPT-4 Code Interpreter [Zhou et al., 2024] and MathCoder [Wang et al., 2024]. Both models generate code blocks in Python, which are then executed and their outputs appended for subsequent steps.

1.3 Thesis outline

The thesis is divided into 3 parts. The **Part I** talks about the state of the art explaining benchmarks, models and datasets of the domain. **Part II** explains the creation process of the dataset created as part of this thesis. **Part III** is dedicated to the fine-tuning of the model, and experiments for it's evaluation.

- **Chapter 1** the problems and benefits of mathematical reasoning in large language models. Introduces math word problems and their formats.
- **Chapter 2** examines benchmarks for math word problems, arithmetic, and code generation.
- **Chapter 3** explores the state of the art, and briefly explains the approaches and innovations of the selected models.
- **Chapter 4** examines datasets relevant to this thesis and provides examples of their format.
- **Chapter 5** explains the proposed format for solving math word problems this thesis works with.
- **Chapter 6** provides a detailed overview of the dataset creation process and statistics for the resulting dataset.
- **Chapter 7** gives insight into the fine-tuning of models created in this thesis, and their direct comparison.
- **Chapter 8** explores arithmetic capabilities of the fine-tuned model on a benchmark with detailed analysis.
- **Chapter 9** introduces the concept of verifiers, explains existing verifiers, and the verifier used in this thesis.
- **Chapter 10** proposes a custom sampling method, which employs the use of a verifier to enhance the quality of the generated solution.
- **Chapter 11** evaluates the fine-tuned model and compares it to the state of the art models. Experiments with different sampling method and verifier versions.
- **Chapter 13** gives a brief overview of this thesis results.

Part I

State-of-the-art

Chapter 2
Benchmarks

Neither generative models for solving MWPs nor models for code generation LLM models can be benchmarked by matching generated samples against their corresponding reference solutions. This is because the problems can be solved correctly in numerous ways that are very different, and at the same time a minor deviation from the correct solution can result in an incorrect solution. As a consequence of this, a number of benchmarks were developed for these specific domains of research. The models for solving MWPs are primarily concerned with reaching the correct answer value, whereas coding benchmarks focus on passing all of the provided unit tests.

2.1 MWP benchmarks

The two most prominent benchmarks for MWPs are GSM8K [Cobbe et al., 2021] and MATH [Hendrycks et al., 2021]. Both of those contain not only testing data, but also training and are explored in more detail in chapter 4 about datasets. GSM8K [Cobbe et al., 2021] contains 1319 simple grade school questions for testing, while MATH [Hendrycks et al., 2021] contains 5000 more challenging questions for testing divided into different categories.

2.2 GSM1K

GSM1K [Zhang et al., 2024] represents an analogous benchmark to GSM8K. It has been meticulously crafted to resemble the original GSM8K benchmark as closely as possible, but with new problems, in order to address the issue of LLM overfitting on the GSM8K benchmark due to dataset contamination or other means. In order to prevent a similar issue, the dataset will not be publicly released until the specified conditions set forth in the original publications have been met. A sample of 50 problems from the total of 1,250 was made available to the public.

2.3 Arithmetic benchmarks

There isn't an universally accepted benchmark for testing the arithmetic ability of LLMs, and almost no work focuses on evaluating this particular ability of their model. Despite this fact there is one dataset created for this purpose called MATH401 [Yuan et al., 2023]. It consists of 401 arithmetic expressions for testing.

query: "$e^{i\pi} + 1 =$"
response: "0"

Figure 2.1: An example of MATH401 format

It tests the ability to solve 16 categories of 25 problems each, to test different arithmetic abilities and expressions of different difficulty. Categories are considered easy or hard. The information is obtained from the GitHub[1] page of the dataset authors and categories determined based on the information provided in the original paper [Yuan et al., 2023]. These categories are as follows:

- (category 0 of a single equation, HARD) The Euler equation,
- (EASY) Addition and subtraction of two integers within 10
- (EASY) Addition and subtraction of two integers within 100
- (EASY) Addition and subtraction of two integers within 1,000
- (HARD) Addition and subtraction of two integers within 1×10^{12}
- (Easy) Addition and subtraction within $-10 \sim 10$
- (Easy) Addition and subtraction within $-100 \sim 100$
- (EASY) Multiplication of two integers within 100
- (EASY) Multiplication of two decimal numbers within 10.
- (HARD) Multiplication of two integers within 100,000
- (HARD) Division of two integers within 100.
- (EASY) Exponentiation with integer base within 10 and integer exponent within $2 \sim 4$.
- (HARD) Exponentiation with decimal number base within 10 and decimal number exponent within $2 \sim 4$.
- (HARD) Addition, subtraction and multiplication of one integer within 10 and a common irrational number (π or ϵ)
- (HARD) Long arithmetic expressions containing addition subtraction multiplication and division with brackets, involving integers within 100.
- (HARD) Trigonometry functions including sin, cos and tan. With inputs of degrees and radians. π can also appear.
- (HARD) Logarithm of integers within 1000 of different bases: 2, ϵ, 10.

[1] `https://github.com/GanjinZero/math401-llm`

2.4 Code benchmarks

Although this work is primarily concerned with MWPs, it employs code generation, rendering it useful for examining models that generate code and understanding how they are compared and what the datasets they're trained on look like.

2.4.1 HumanEval

There are numerous benchmarks for code generation. The most widely used benchmark is HumanEval [Chen et al., 2021], which was released in the same paper where the pass@k metric was also introduced. HumanEval consists of 164 hand-written programming problems docstrings and a few unit tests (the average is 7.7 unit tests per problem).

2.4.2 Pass@k

Pass@k metric gained popularity for the code generation task in later years. For this metric, k samples are generated for each problem. The problem is considered solved if any sample passes the unit tests for the problem. To reduce variance, practical applications require a different calculation method, as explained in the original work [Chen et al., 2021].

Chapter 3
State-of-the-Art (SOTA) Overview

A multitude of models have been developed to address both MWPs and coding benchmarks, with varying degrees of success. This chapter provides a brief overview of some of these models. Some are considered the current state-of-the-art among generative LLMs in general, while others have been designed to address those specific problems. The list also includes models that are crucial for this particular work.

3.1 MWP SOTA

The results data in the table 3.1 other than "4-bit MetaMath-Mistral-7B w/o calculator" and "4-bit Abel-7B-001 w/o calculator" were obtained from the original papers of these models, or from the papers with code leader-board[1].

Model name	Size	GSM8K score	MATH score
GPT4-Code + CSV + Voting 3.1.1	Unknown	97.0	89.2
GPT4-Code 3.1.1	Unknown	92.9	87.5
Shepherd+Mistral-7B 3.1.5	7B	89.1	43.5
MathCoder-CL-34B 3.1.8	34B	81.7	45.2
Abel-70B-001 3.1.6	70B	83.62	28.26
MetaMath-70B 3.1.2	70B	82.3	26.0
Abel-7B-002 3.1.6	7B	80.44	29.46
Minerva 540B 3.1.7	540B	78.5	50.3
Minerva 62B 3.1.7	62B	68.5 (maj1@100)	64.9 (maj5@256)
MetaMath-Mistral-7B 3.1.4	7B	77.7	28.2
4-bit MetaMath-Mistral-7B w/o calculator 3.1.4	7B	74.12	×
Abel-7B-001 3.1.6	7B	59.74	×
4-bit Abel-7B-001 w/o calculator 3.1.6	7B	54.97	×
Mistral 7b 3.1.3	7B	52.2	13.1

Table 3.1: Models compared on GSM8K and MATH benchmarks. The table only includes the best currently available versions of those models with the highest-performing evaluation methods or those that are relevant to this work. The results of my tests using 4-bit quantization, while not employing the calculator to correct the arithmetic operations, are in the rows "4-bit Abel-7b-001 w/o calculator" and "4-bit MetaMath-Mistral-7B w/o calculator".

[1] `https://paperswithcode.com/sota/arithmetic-reasoning-on-gsm8k`

3.1.1 GPT4-Code

The GPT-4 code interpreter [Zhou et al., 2024] employs both natural language and code generation and execution, demonstrating high zero-shot accuracy on MWP benchmarks. Additionally, the original paper introduces a novel verification process, self-debugging and CSV. These are explained in greater detail in the chapter 9 verifiers. They also demonstrate that Python code chains improve computational capability more than natural language chains, and that employing code multiple times is beneficial as it allows self-debugging in their case.

3.1.2 MetaMath models

MetaMath [Yu et al., 2023] models of varying sizes, including 7B, 13B, and 70B. These models are based on LlaMA-2, fine-tuned using the MetaMathQA dataset. The MetaMathQA dataset is explored in more detail in chapter 4 datasets. The original paper notes that a model fine-tuned on data containing incorrect solutions as well, performed better. The authors hypothesize that this may be attributed to the presence of some correct intermediate reasoning steps, despite the final answer being incorrect. The MetaMath-7B model demonstrated superior performance compared to other SOTA models of the same size at the time of its release, while the MetaMath-70B model outperformed the currently available version of GPT-3.5-Turbo at the time the paper was released.

3.1.3 Mistral

Mistral [Jiang et al., 2023] is a 7B model that uses grouped-query attention and sliding window attention, enabling faster inference and the handling of long sequences. Despite its smaller size, it outperforms larger models such as LlaMA-2 34B in both mathematics and code generation, and approaches the performance of more specialized models such as Code-LLama 7B without compromising performance in other domains. The open-source nature of this model, coupled with its performance and size, provides a solid foundation for the further development of smaller 7B models in these areas.
A recent study [Zhang et al., 2024] draws to attention the fact that models real performance might be lower than the one indicated by benchmarks due to overfitting resulting from data contamination or other mechanisms. The GSM1K benchmark created by the authors indicates that Mistral models are among those exhibiting a difference in performance. Nevertheless, the GSM1K benchmark demonstrates that despite the presence of overfitting, these models are still capable of reasoning, and the Mistral models remain among the strongest open-source models.

3.1.4 MetaMath-Mistral-7B

The MetaMath-Mistral-7B [Yu et al., 2023] model was developed by fine-tuning the Mistral model on the MetaMathQA dataset. This resulted in a significant improvement in the model's performance on MWP benchmarks. This work further builds upon this model as explained in 7 model.

3.1.5 Shepherd + Mistral

The Shepherd+Mistral-7B [Wang et al., 2023] model is based on the Mistral model, which has been fine-tuned on the MetaMathQA dataset. In addition, it incorporates reinforce-

ment learning and verification techniques, which have led to a notable improvement in accuracy. The best result is selected from 256 generated outputs, ranked by the verification method. Reinforcement learning is employed to automatically annotate intermediate reasoning steps based on their potential to lead to the correct final answer.
Despite the model's impressive performance on the MATH and the GSM8K benchmarks, testing the model on the GSM1K [Zhang et al., 2024] benchmark revealed that it exhibited the greatest degree of overfitting. Further examination showed low per-character log-likelihood, suggesting that data contamination was not the cause. The GSM1K authors hypothesize that it may be caused by the reward modeling process, but more research is needed to determine this.

3.1.6 Abel

Abel [Chern et al., 2023] models have gone largely unnoticed, likely due to the sole source of information being their GitHub page. The Abel models are a group of models that focus on supervised finetuning on carefully selected data. It is highly likely that Abel-7B-002 is based upon Mistral-7B, while other models are based on Llama 2 models of various sizes. However, it is difficult to determine with certainty.

I have performed a test of the Abel-7B-001 model on GSM8K, which yielded an accuracy of 54.97, while the reported accuracy by the authors was 59.74. The difference is likely due to the fact that I did not use any calculator or other automated evaluation to fill the "«expr1 op expr2 = " sections with a correct value as it is commonly done with GSM8K, instead allowing the model to fill the result. Furthermore, it is possible that the observed difference could be partially attributed to quantization.

3.1.7 Minerva

Minerva [Lewkowycz et al., 2022] are models of various sizes based on the PaLM models, further trained on a large dataset containing scientific and mathematical data collected from arXiv (58GB of data) and internet web pages (60GB of data). Authors show that a method of sampling k times and selecting a solution using majority voting (known as maj1@k) outperforms greedy decoding. And that majority voting performance saturates faster than the performance of pass@k2.4.2. Authors also utilized SymPy to normalize final answers before comparing them to the target answer, improving the overall accuracy by around 1%.

3.1.8 MathCoder

MathCoder [Wang et al., 2024] is a model that utilizes interleaving blocks of natural language, code, and execution results in a manner similar to the GPT-4 Code Interpreter. It achieved SOTA results on MATH and GSM8K benchmarks among open-source LLMs. The work proposed a method of generating high-quality dataset with math problems and code-based solutions. The authors observed a gap in the difficulty levels of the problems in the GSM8K and MATH datasets. To address this, they employed other LLMs to generate new problems with a difficulty level that falls between those of the provided samples from GSM8K and MATH. To create a ground truth solution, they used their initial MathCoder models to generate solutions and kept only those where n = 3 solutions matched. Using this approach, they created the MathCodeInstruct dataset.

3.2 Coding SOTA

The results data in the table 3.2 were obtained from the original papers of these models, or from the papers with code leader-board[2].

Model name	Size	HumanEval pass@1
GPT-4	Unknown	76.5
CODE-T (code-davinci-002) 9.0.3	Unknown	65.8
Unnatural Code Llama 3.2.2	34B	62.2
InstructCodeT5+ 16B (CodeT) 9.0.3	16B	42.9
StarCoder 3.2.3	15.5B	33.6
Codex-S 3.2.1	12B	32.2
Mistral 7B 3.1.3	7B	30.5

Table 3.2: Models compared on HumanEval benchmark. This table includes some models or methods from the research area of code generation. Some of these models are explained in more detail.

3.2.1 Codex

Codex [Chen et al., 2021] is one of the earlier works to focus on code generation, introducing the Pass@k metric 2.4.2. In the same paper the authors introduced the HumanEval 2.4.1 benchmark. Codex is trained on 159GB of Python files collected from 54 million public GitHub repositories. Additionally, authors collected problems and solutions from several interview preparation and programming contest websites. Their findings indicate that the addition of new tokens for representing different lengths of white-spaces allows for the representation of code using 30% fewer tokens.

3.2.2 Code Llama

Code Llama [Roziere et al., 2023] is a foundation model for code generation tasks based on LlaMA-2 [Touvron et al., 2023]. The paper demonstrates that initializing the model with a model pre-trained on both general-purpose text and code data, outperforms a model with the same architecture trained solely on code. There are multiple Code Llama models in addition to the foundational Code Llama (Code Llama-Python, Code Llama-Instruct, and Unnatural Code Llama) of various sizes. In addition to prompt completion, the aforementioned models are capable of infilling based on the surrounding context. The Instruct version of the model was fine-tuned on a proprietary data set to improve its safety and helpfulness, while the Python version is specialized for the generation of Python code. The authors noted that generation of a larger amount of solutions and tests with a smaller 7B model is more efficient than generating fewer solutions and tests with a larger 34B model given the same compute budget.

3.2.3 StarCoder

Starcoder [Li et al., 2023] is a fine-tuned version of the StarCoderBase model which is trained using Masked Language Modeling and Next Sentence Prediction objectives. It is capable of infilling based on the surrounding context.

[2] https://paperswithcode.com/sota/math-word-problem-solving-on-math

Chapter 4
Datasets

This chapter provides a more detailed examination of various datasets of MWPs. A multitude of other datasets are available for both mathematical reasoning and coding tasks. It appears that there is no comprehensive source of information that would contain the majority of these datasets in a single location. To address this issue, I created a table containing the majority of the datasets that I could find, which is available in appendix A.

4.1 MATH

MATH [Hendrycks et al., 2021] is a dataset and benchmark that consists of 12,500 challenging MWPs. 7,500 problems for training and 5,000 for testing. The problems are categorised into the following areas: algebra, counting and probability, geometry, intermediate algebra, number theory, pre-algebra, and precalculus. Each problem is assigned a difficulty level from 1 to 5, with 1 being the simplest for people to solve and 5 being the most difficult. The problems are designed to be challenging even for most human individuals. The dataset format presents the solution in a step-by-step format, including LaTeX, with the final answer highlighted by being `\boxed{}`.

Problem: Let

$$f(x) = \begin{cases} ax + 3, & \text{if } x > 2, \\ x - 5, & \text{if } -2 \leq x \leq 2, \\ 2x - b, & \text{if } x < -2. \end{cases}$$

Find $a + b$ if the piecewise function is continuous (which means that its graph can be drawn without lifting your pencil from the paper).

Level: Level 5

Type: Algebra

Solution: For the piecewise function to be continuous, the cases must "meet" at 2 and -2. For example, $ax + 3$ and $x - 5$ must be equal when $x = 2$. This implies $a(2) + 3 = 2 - 5$, which we solve to get $2a = -6 \Rightarrow a = -3$. Similarly, $x - 5$ and $2x - b$ must be equal when $x = -2$. Substituting, we get $-2 - 5 = 2(-2) - b$, which implies $b = 3$. So $a + b = -3 + 3 = \boxed{0}$.

Figure 4.1: An example of MATH format

4.2 GSM8K

GSM8K [Cobbe et al., 2021] a dataset and benchmark that consists of 8,792 simple grade school MWP questions with answers. 7,473 problems for training and 1,319 for testing. The answer is presented in a step-by-step format, with the final answer value highlighted by the "####" token.
The dataset employs a calculator notation to circumvent issues with arithmetic computations. When the model uses calculation annotations a calculator will override the sampling with the correct value. Example of calculation annotations: «48/2=24» where the value 24 is entered by the calculator.
The dataset also contains a Socratic format containing the same questions, but in the Socratic format, the answer contains sub-questions.

question:
Natalia sold clips to 48 of her friends in April,
and then she sold half as many clips in May.
How many clips did Natalia sell altogether in April and May?
answer:
Natalia sold 48/2 = «48/2=24»24 clips in May.
Natalia sold 48+24 = «48+24=72»72 clips altogether in April and
May.
72

Figure 4.2: An example of GSM8K format

question:
Natalia sold clips to 48 of her friends in April,
and then she sold half as many clips in May.
How many clips did Natalia sell altogether in April and May?
answer:
How many clips did Natalia sell in May?
** Natalia sold 48/2 = «48/2=24»24 clips in May.
How many clips did Natalia sell altogether in April and May?
** Natalia sold 48+24 = «48+24=72»72 clips altogether in April and
May.
72'

Figure 4.3: An example of GSM8K Socratic format

4.3 MathCodeInstruct

MathCodeInstruct [Wang et al., 2024] was created in two steps. Initially in the first step, solutions were created that consisted of interleaved blocks of natural language, code, and execution results for the GSM8K and MATH datasets. Subsequently, in the second step, more problems were augmented by asking LLM to generate questions with difficulty levels between the provided GSM8K and MATH problems. Finally, solutions were generated for these new problems using MathCoder-Initial (262,039 rows), which was trained on the data created from the GSM8K and MATH datasets in the first step.

4.4 MetamathQA

MetamathQ [Yu et al., 2023] is a large MWP dataset that consists of 395,000 problems and answers for training. It aims to maximize diversity and enable strong generalization. The dataset was created by bootstrapping the GSM8K and MATH datasets through the augmentation and rewriting of both questions and answers. The following methods of bootstrapping were employed:

- Question Rephrasing - Bootstrapping in the forward reasoning direction by rephrasing the question.
- Question Self-Verification - Bootstrapping in the backward reasoning direction by rewriting question with answer into a declarative statement and a number in the resulting statement is masked with x while a question "What is the value of unknown variable x?" is appended at the end.
- FOBAR question - The answer is appended to the question while some variable is masked, forming a new question that asks: "If we know the answer to the above question is (answer to the original question), what is the value of the unknown variable x?".
- Answer augmentation - Answers are augmented by generating more reasoning paths using a few-shot chain-of-thought by appending the question to a few reasoning examples, and keeping reasoning paths with correct answers.

Part II

Dataset creation

Chapter 5
Proposed format

The format proposed in this work is based on the interleaving of blocks of Python code and natural language reasoning. It employs natural language blocks to extract values from the question and build connections between them. Instead of the more general Python coding skills required in other works, in this work, code blocks focus on generating specific Python code utilizing SymPy[1]. SymPy is an open-source Python library for symbolic mathematics. This approach bypasses arithmetic computations by performing the most complex calculations in code. This eliminates the potential for arithmetic errors, and allows the use of a variety of shortcuts provided by SymPy. As illustrated in the next example figure 5.1, first a step-by-step solution in natural language is generated. This solution may be wrong, and its purpose is to find out which information is important to reach the final answer and what intermediate steps and variables are required.

A solution is then generated by interchanging natural language reasoning blocks and blocks of symbolic computation code. New tokens ([|Text|], [|Sympy|], and [|EndOfBlock|]) have been added to separate text and code blocks. The content of these blocks varies according to the specific question being addressed. However, they mostly follow a similar structure. The initial text block reiterates the values of the significant variables in some way and mentions variables that will need to be calculated in order to achieve the final answer. The first SymPy block starts by defining symbols or variables. Then subsequent blocks set up equations reflecting relationships between variables and use SymPy functions such as solve, simplify, subs, or others to arrive at the final answer. This approach allows the focus to be placed on the relationships between variables, rather than on the calculations. Furthermore intermediate results do not matter, which allows the SymPy blocks to be executed once at the very end, rather than being executed at the end of each code block as is the case in other works. To receive the answer value, all code blocks are executed as a single Python program. This approach also serves to prevent text blocks from attempting to guess the result values of arithmetic operations. Instead, they rely on defined variables to represent the variables or expressions and to focus on the relationships between them. Intermediate results are stored in the form of symbolic expressions that can be automatically converted into various formats, including LaTeX, and are accurate without the numerical instability of floating-point calculations. Finally, the last SymPy blocks replace the remaining variables in the expression, and add a print statement at the end, marking the final result that can now be calculated by executing all the concatenated code blocks together. The integration of reasoning blocks and explanatory comments for code facilitates the comprehension of the reasoning behind it.

[1] `https://www.sympy.org/en/index.html`

Question: The town of Belize has 400 homes. One fourth of the town's homes are white. One fifth of the non-white homes have a fireplace. How many of the non-white homes do not have a fireplace?
Answer: One fourth of the town's homes are white, so there are 400/4 = 100 white homes. The remaining non-white homes are 400 - 100 = 300 homes. One fifth of the non-white homes have a fireplace, so there are 300/5 = 60 non-white homes with a fireplace. Therefore, the number of non-white homes without a fireplace is 300 - 60 = 240.
240
The answer is: 240
Solution:
[|Text|] Let's define the variables:
total_homes: total number of homes in Belize
white_homes_percentage: percentage of homes that are white
fireplace_percentage_non_white: percentage of non-white homes that have a fireplace We can calculate the number of white homes and then find the number of non-white homes that do not have a fireplace. [|EndOfBlock|] [|Sympy|]
```
# Given values
total_homes_value = 400 # total number of homes in Belize
white_homes_percentage_value = 25 # percentage of homes that are white
fireplace_percentage_non_white_value = 20 # percentage of non-white homes that have a fireplace
# Define symbols
total_homes, white_homes_percentage, fireplace_percentage_non_white,
non_white_homes_no_fireplace = sp.symbols('total_homes white_homes_percentage fireplace_percentage_non_white non_white_homes_no_fireplace')
```
[|EndOfBlock|] [|Text|]
Calculate the number of white homes in Belize. [|EndOfBlock|] [|Sympy|]
```
# Calculate number of white homes
white_homes = (white_homes_percentage / 100) * total_homes
```
[|EndOfBlock|] [|Text|]
Find the number of non-white homes by subtracting the number of white homes from the total number of homes. [|EndOfBlock|] [|Sympy|]
```
# Calculate number of non-white homes
non_white_homes = total_homes - white_homes
```
[|EndOfBlock|] [|Text|]
Determine the number of non-white homes that have a fireplace.
[|EndOfBlock|] [|Sympy|]
```
# Calculate number of non-white homes with a fireplace
non_white_homes_with_fireplace = (fireplace_percentage_non_white / 100) *
non_white_homes
```
[|EndOfBlock|] [|Text|]
Finally, compute the number of non-white homes without a fireplace.
[|EndOfBlock|] [|Sympy|]
```
# Calculate number of non-white homes without a fireplace
non_white_homes_no_fireplace = non_white_homes - non_white_homes_with_fireplace
non_white_homes_no_fireplace_value = non_white_homes_no_fireplace.subs({ total_homes: total_homes_value, white_homes_percentage: white_homes_percentage_value, fireplace_percentage_non_white: fireplace_percentage_non_white_value})
```
[|EndOfBlock|] [|Sympy|]
```
print(non_white_homes_no_fireplace_value)
```
[|EndOfBlock|]

Figure 5.1: An example of a longer MWP in the proposed format

When executed, the code outputs a value of 240, which is the final result. Coincidentally, in the current example, it's the same answer as the one generated in the textual answer section.

Listing 5.1: An example of concatenated SymPy code blocks script, which yields the final answer upon execution.

```
# Given values
total_homes_value = 400 # total number of homes in Belize
white_homes_percentage_value = 25 # percentage of homes that are white
fireplace_percentage_non_white_value = 20 # percentage of non-white homes that have a fireplace

# Define symbols
total_homes, white_homes_percentage, fireplace_percentage_non_white, non_white_homes_no_fireplace = sp.symbols(
'total_homes white_homes_percentage fireplace_percentage_non_white non_white_homes_no_fireplace')

# Calculate number of white homes
white_homes = (white_homes_percentage / 100) * total_homes

# Calculate number of non-white homes
non_white_homes = total_homes - white_homes

# Calculate number of non-white homes with a fireplace
non_white_homes_with_fireplace = (fireplace_percentage_non_white / 100) *  non_white_homes

# Calculate number of non-white homes without a fireplace
non_white_homes_no_fireplace = non_white_homes - non_white_homes_with_fireplace
non_white_homes_no_fireplace_value = non_white_homes_no_fireplace.subs({
total_homes: total_homes_value,
white_homes_percentage: white_homes_percentage_value,
fireplace_percentage_non_white: fireplace_percentage_non_white_value
})

print(non_white_homes_no_fireplace_value)

```

Chapter 6
MetaMath-GSM8K-SymPy dataset

This chapter outlines the methodology employed to create the dataset in the previously described format, and provides an analysis of the resulting data. The process can be divided into several steps, which are illustrated in the figure 6.1. Each step is described in more detail in its corresponding subsection.
Terminology used in this chapter:

- Reformatting / Rephrasing / Translating - The process of generating the solution section in the proposed format with the question and answer provided as input.
- Correct - Upon execution, the code functions as intended and yields the correct answer value.
- ME - Mathematical Error. Upon execution, the code functions as intended but the answer value is wrong.
- CE - Code Error. The code fails upon execution.

ChatGPT Reformatting	**Count**	**Percentage of GSM8K**
Total Rows	7,343	100%
Correct Reformatting	3,012	41%
Incorrect Reformatting	4,331	59%
Mathematical Errors	1,944	26.5% (of all reformatted), 44.89% (of all incorrect)
Code Errors	2,387	32.5% (of all reformatted), 55.11% (of all incorrect)

Table 6.1: Results of GSM8K reformatting with ChatGPT-3.5

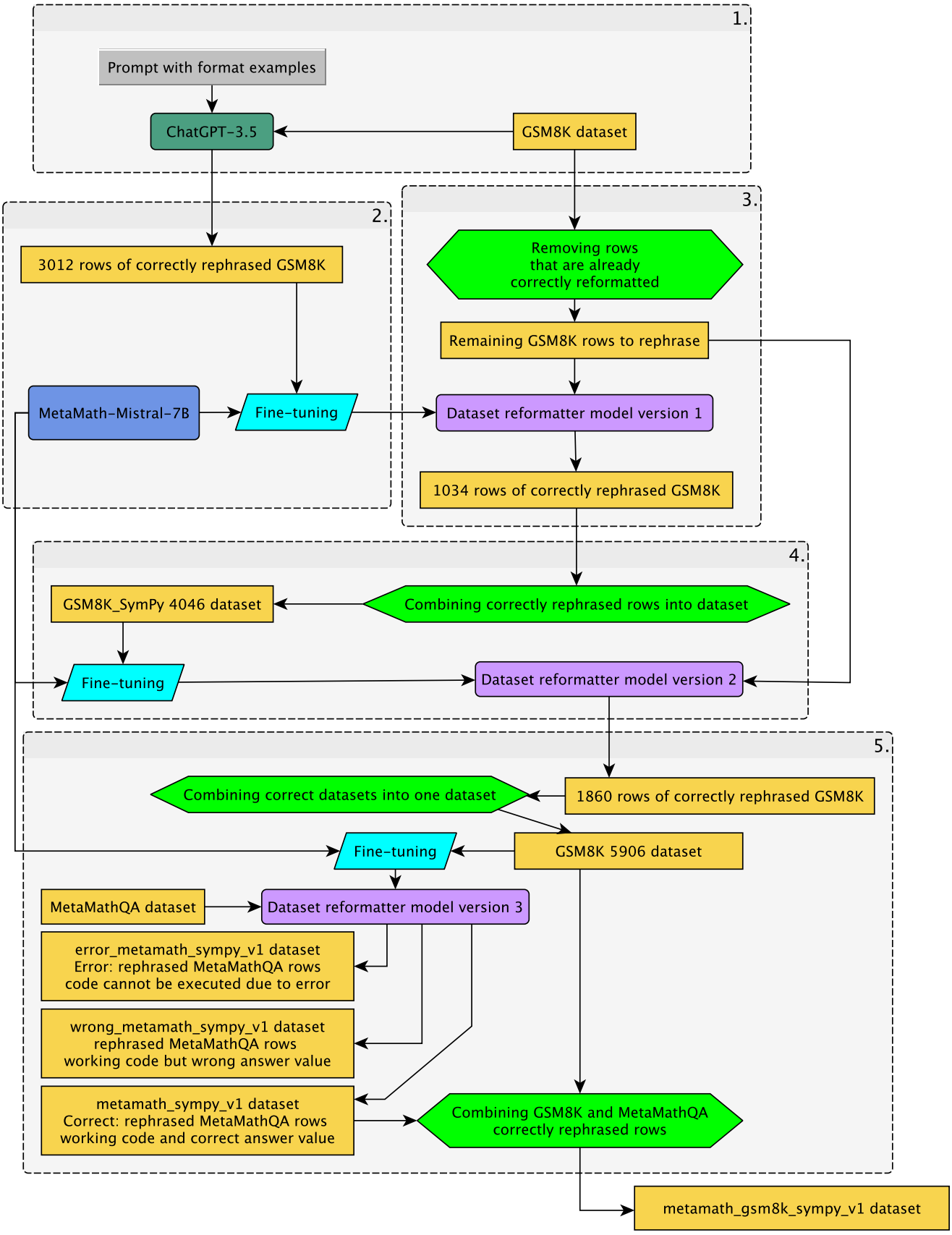


Figure 6.1: A diagram illustrating the creation process of the MetaMath-GSM8K-SymPy dataset

6.1 1. GSM8K reformatting using ChatGPT

In order to obtain the initial dataset for fine-tuning a reformatting model, it was necessary to gather some initial data by reformatting the dataset into the proposed format. This data was acquired by few-shot prompting ChatGPT-3.5[1] to reformat the GSM8K dataset in accordance with provided examples of the required format. The prompt is available in appendix B, and contains seven examples of correctly reformatted data on examples from the GSM8K and MetaMathQA datasets. The aforementioned examples were constructed manually, with due consideration given to the formatting preferences of ChatGPT and the issues with the outputs provided by ChatGPT. They were tested and calibrated on the manual reformatting of the first 100 examples from the GSM8K dataset in order to achieve adequate reformatting performance. The examples illustrate how the requisite format may vary depending on the question or answer differences. They also provide additional guidance on the use of SymPy. For instance, "Example 6" demonstrates that, despite the answer including a simplification of the expression in textual form, the solution should still simply apply the SymPy simplify function.
Upon evaluation of the 7,343 rows of the GSM8K train dataset reformatted by ChatGPT-3.5, it was found that 3,012 rows (41%) were correctly reformatted. The fact that the automatic evaluation process distinguishes between mathematical and code errors allows for further analysis illustrated in table 6.1. Of the incorrect reformattings, 1,944 failures (26.5% of all reformatted, 44.89% of all failures) were attributable to mathematical errors, yet the code executed correctly. In contrast, 2,387 failures (32.5% of all reformatted, 55.11% of all failures) were due to faulty generated code.

6.2 2. Fine-tuning a reformatting model version 1

Upon the completion of the initial step, the training set comprises 3,012 reformatted solutions. This training set is then used to fine-tune a MetaMath-Mistral-7B model in order to further reformat the dataset. The fine-tuning process is analogous to that employed for the MWP solver model, as detailed in chapter 7, except the absence of additional tokens, LoRA rank being $r = 32$, learning rate $lr = 2.5e-5$, and employing only first 1300 tokens during training. The resulting model is then applied to reformat the remaining 59% of the GSM8K dataset.

Reformatter-v1 Reformatting	**Count**	**Percentage of attempted rows**
Total Rows Attempted	2,722	100%
Correct Reformatting	1,034	37.99%
Incorrect Reformatting	1,688	62.01%
Mathematical Errors	224	8.96% (of all reformatted), 13.27% (of all incorrect)
Programming Errors	1,444	53.05% (of all reformatted), 86.73% (of all incorrect)

Table 6.2: Performance of Reformatter-v1

[1] `https://openai.com/index/chatgpt/`

6.3 3. Further GSM8K reformatting

Due to time constraints, only the first 2,722 of the remaining 4,331 rows of GSM8K were reformatted using this fine-tuned reformatting model (later reformatter-v1). Of those 2,722, a 1,034 examples were correctly reformatted, with a success rate of 37.98%. This represents a lower performance than previously achieved by ChatGPT-3.5 (41%). Of the remaining 1,688 incorrect translations, only 13.27% were attributed to mathematical errors, while 86.73% were due to code errors. This is a significant difference from ChatGPT-3.5 statistics where the distribution of errors was much closer to equal. This is likely caused by the reformatter-v1 model being a fine-tune of a MetaMath-Mistral-7B model, which has already undergone significant pre-training for mathematical reasoning, but encountered minimal SymPy code data during its previous training. This statistic can be found in the table 6.2.

6.4 4. Fine-tuning a reformatting model version 2

The fine-tuning of the reformatter-v2 is almost identical to the fine-tuning of reformatter-v1 except with a larger dataset, updated hyper-parameters, and being prompted with: "Solution: [|Text|] Let's define the variables: " instead of "Solution:" fixing the frequent issue of the first textual block being nonsense, exhibited by the previous model. The updated hyper-parameters had increased the number of tokens employed during the training process from 1300 to 1400, LoRA was set to also fine-tune the embedding layer, and new tokens were added. The tokens additional were "[|Text|]", "[|EndOfBlock|]", "[|Sympy|]" as well as the most common sub-sequences encountered in the dataset: "Let's define the variables", "import sympy as sp", "# Define symbols", "# Calculate total number of" and "# Total amount" in attempt to decrease the number of tokens required for the solution section. It achieved 42% accuracy, comparable to that of ChatGPT-3.5, while exhibiting a performance similar to that of reformatter-v1. Reformatter-v2 has been employed in the same way as in step three, rephrasing another 1860 rows of GSM8K, consequently coming together to create the GSM8K-SymPy dataset[2].

6.5 5. MetaMath-GSM8K-SymPy dataset

The GSM8K-SymPy dataset was then used for training a reformatter-v3 model on a larger dataset with better hyper-parameters, the most advantageous of which was the increase in LoRA rank from $r = 32$ to $r = 64$, and lower learning rate value of $lr = 2.5e - 6$. The dataset was cleaned by removing unnecessary parts that could be detected automatically, such as everything generated after the first "print()" statement. The majority of the additional tokens were removed, with only the "[|Text|]", "[|EndOfBlock|]", and "[|Sympy|]" tokens retained. This was done because it appeared that these additional tokens lead to a decrease in performance, due to the model starting to avoid using these formulations and resorting to less informative alternatives instead.
The reformatter-v3 model was then employed to reformat the MetaMathQA dataset. Throughout the process, the inference code was gradually optimized. Initially, a single solution was generated in approximately 180 seconds. However, with the model gen-

[2] `https://huggingface.co/datasets/tfshaman/gsm8k_sympy_v3`

erating answers in parallel, the time per solution was reduced to as little as 2.4 seconds. The MetaMathQA could be divided into eight categories based on their origin and the method of augmentation used by the authors. The aforementioned eight categories, namely GSM-AnsAug, GSM-FOBAR, GSM-Rephrased, GSM-SV, MATH-AnsAug, MATH-FOBAR, MATH-Rephrased, and MATH-SV, were also processed in parallel. The following table 6.3 presents the results of MetaMathQA reformatting, sorted by the aforementioned categories. It is evident that there is considerable variation in model performance among these categories.

Category	Correct	ME	CE	Correct Percentage	Total Rows
GSM AnsAug	62,890	6,826	10,284	78.61%	80,000
GSM FOBAR	7,829	4,468	27,703	19.57%	40,000
GSM Rephrased	62,391	6,181	10,991	77.99%	80,000
GSM SV	17,434	2,414	20,152	43.59%	40,000
MATH AnsAug	17,652	9,322	48,026	23.54%	75,000
MATH FOBAR	2,004	1,516	11,480	13.36%	15,000
MATH Rephrased	12,318	5,941	31,741	24.64%	50,000
MATH SV	3,284	1,573	10,143	24.64%	15,000
GSM SUM	150,544	20,326	69,130	63.73%	240,000
MATH SUM	35,258	18,352	101,390	22.75%	155,000
Total SUM	185,802	38,678	170,520	47.04%	395,000

Table 6.3: Analysis of the Reformatter-v3 model performance during the process of MetaMathQA dataset reformatting

As illustrated in the table, the Reformatter-v3 model struggles the most with the FOBAR-type questions. Furthermore, the performance of reformatting a more challenging MATH dataset is considerably lower than performanceo of reformatting the data derived from GSM8K, which is more closely resembling its original training data. It's also worth noting that mathematical errors (18.48%) are still considerably less frequent than code errors (81.52%). Upon examination of the exceptions returned in the event of code errors, it becomes evident that the most common exceptions are variations on the "name 'x' is not defined" error. This is due to the model never having previously encountered questions asking about the value of x, which often leads to errors. Despite this, 19% of GSM FOBAR and 13% of MATH FOBAR were reformatted correctly, indicating that the model is occasionally capable of reformatting those questions. Those correctly reformatted questions may provide sufficient training data for subsequent models to become more adept at working with similar variables in the future. It can be observed that the reformatter-v3 model has undergone a general improvement from its predecessors. This is evidenced by the fact that it now achieves 63.73% on GSM-derived data, and even with MATH-derived data taken into account, it still achieves a higher accuracy of 47.05% compared to its predecessors on entirely GSM data.

The reformatted data underwent further filtering and cleaning procedures, including some attempts at automatic code fixing, resulting in three publicly available datasets. MetaMath-SymPy[3], MetaMath-SymPy-Wrong[4] and MetaMath-Error[5].
By combining the MetaMath-SymPy with previously created GSM8K-SymPy a new dataset MetaMath-GSM8K-SymPy[6] was created. Following figures 6.2 and 6.3 demonstrate the distribution of tokens between questions, answers, and newly added solution parts. The tokenizer MetaMath-Mistral-7B uses was modified prior to analysis in the same way it is modified for fine-tuning, by adding "[|EndOfBlock|]", "[|Sympy|]" and "[|Text|]" tokens.

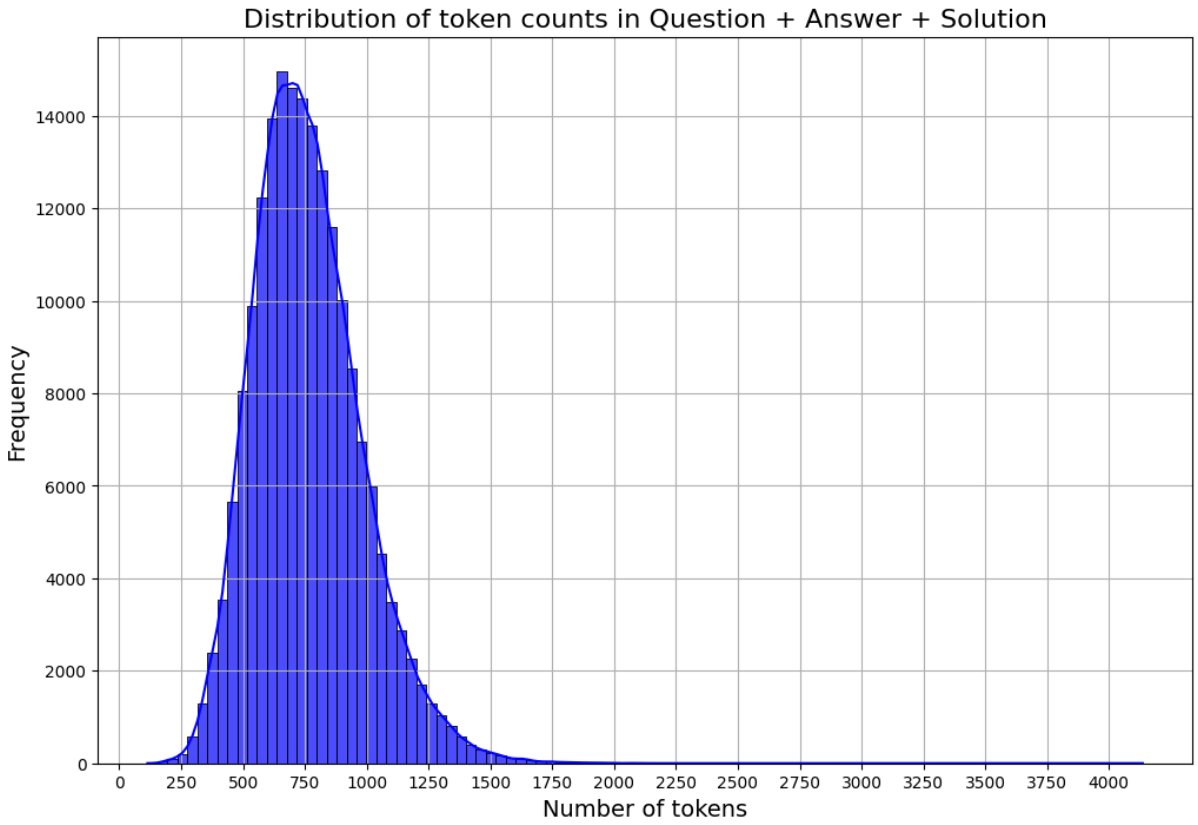


Figure 6.2: A distribution of tokens in questions, answers and solutions in the dataset together. Demonstrates that full text almost never exceeds 1500 tokens.

[3] https://huggingface.co/datasets/tfshaman/metamath_sympy_v1
[4] https://huggingface.co/datasets/tfshaman/wrong_metamath_sympy_v1
[5] https://huggingface.co/datasets/tfshaman/error_metamath_sympy_v1
[6] https://huggingface.co/datasets/tfshaman/metamath_gsm8k_sympy_v1

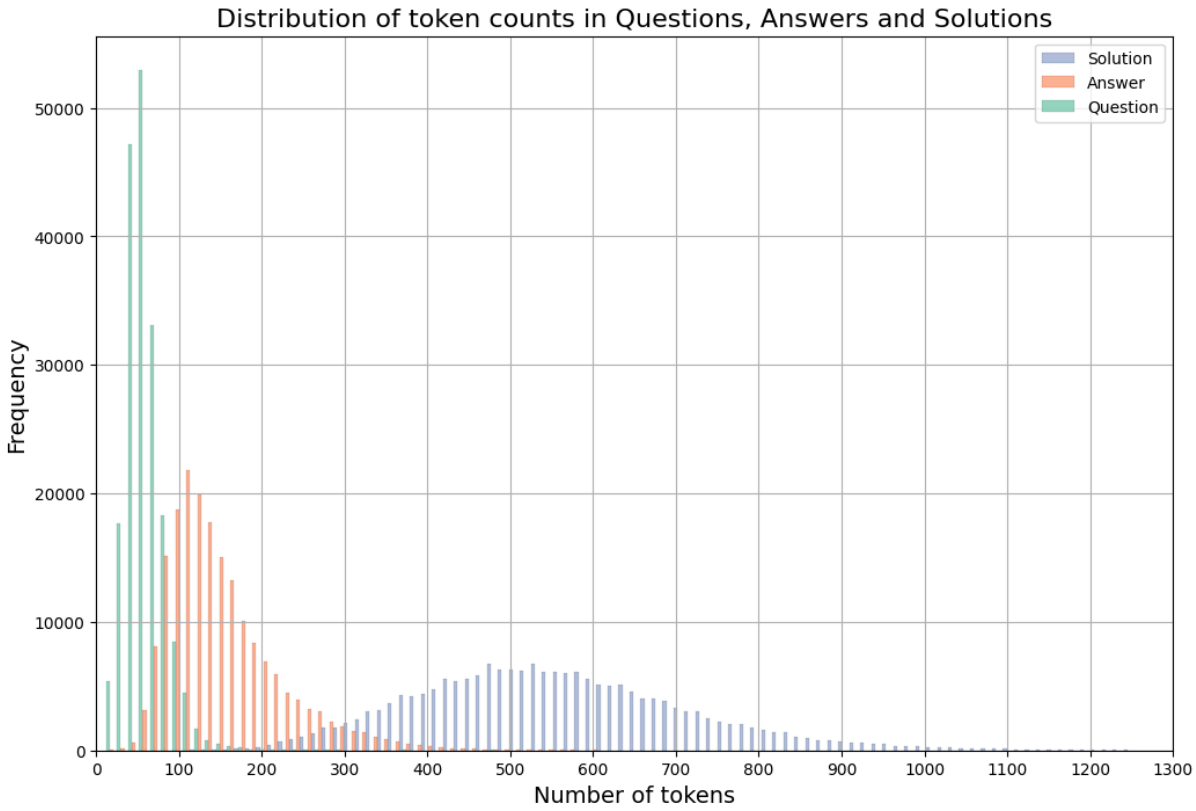


Figure 6.3: A distribution of tokens in questions, answers, and solutions. This demonstrates that questions are typically the shortest, usually requiring less than 100 tokens. The length of answers varies, with longer answers being less common, which resembles log-normal distribution. However, the distribution of solution lengths resembles a normal distribution with an average of approximately 550 tokens.

Category	Mean Tokens	Median Tokens
1, 2 blocks	236.10	226
3, 4 blocks	104.48	88
5, 6 blocks	87.66	77
7, 8 blocks	66.08	57
9, 10 blocks	35.03	0
11, 12 blocks	10.78	0
13, 14 blocks	3.10	0
15, 16 blocks	0.86	0
17, 18 blocks	0.21	0
19, 20 blocks	0.05	0

Table 6.4: Block-pair wise mean and median amount of tokens per pair

Because solution blocks come in pairs of textual and SymPy blocks, the analysis of block lengths was conducted on the pairs of blocks. The figure 6.4 demonstrates that the normal distribution behavior is primarily influenced by the initial pair of blocks, which contain the variables and symbols. Subsequent blocks exhibit a tendency to resemble a log-normal distribution, similar to the answers. Following figure 6.4 also demonstrates that there are rarely more than four non-print block pairs required. Given the format, it is evident that the last block is a short print statement.

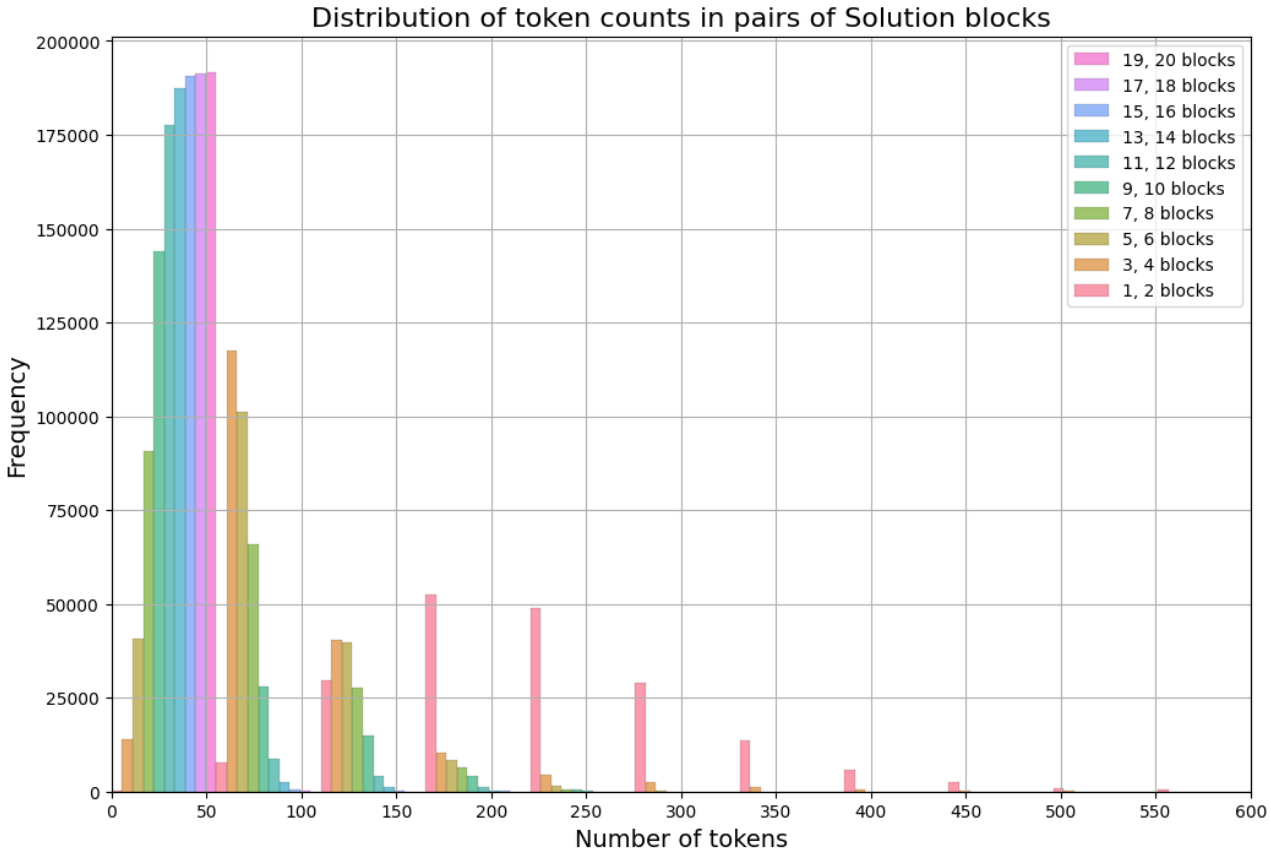


Figure 6.4: A distribution of tokens in pairs of blocks.

Part III

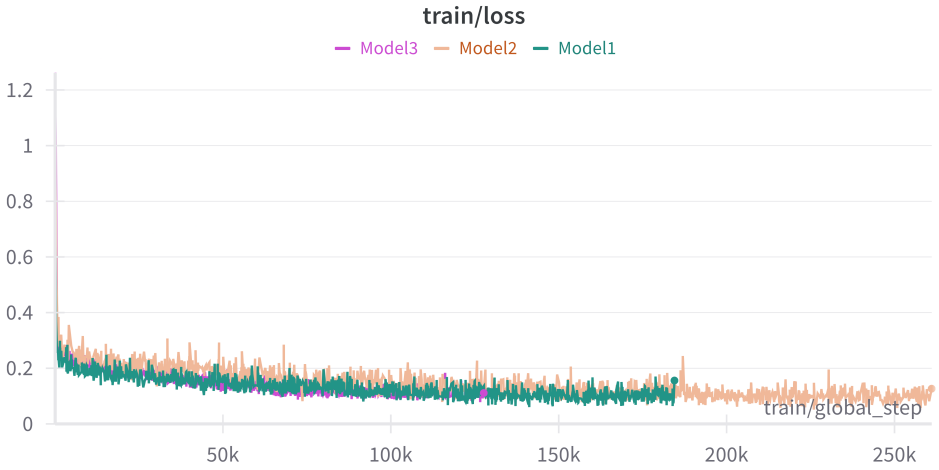
SymPy-Mistral

Chapter 7
Model

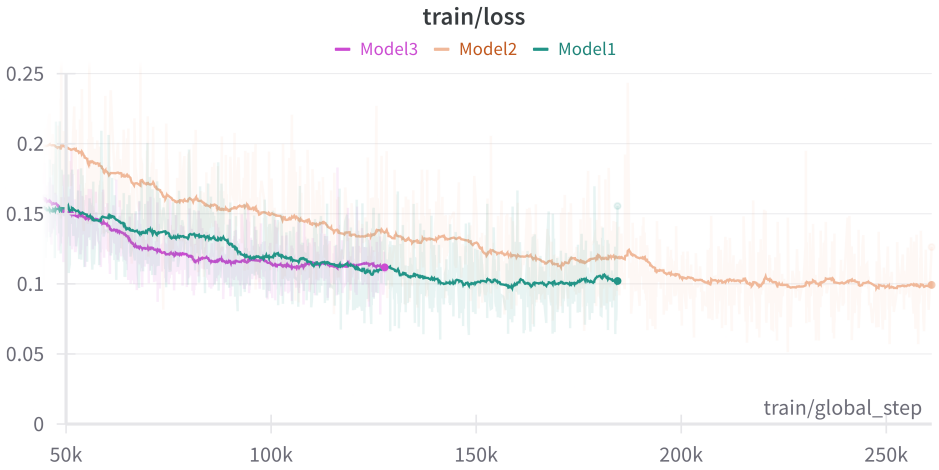
7.1 Fine-tuning models

The MetaMath-Mistral-7B model was fine-tuned on MetaMath-GSM8K-SymPy dataset to create three SymPy-Mistral models, which will be reffered to simply as ”model-1”, ”model-2” and ”model-3” in this chapter.
The MetaMath-Mistral-7B model was selected as the baseline for fine-tuning, due to its previous fine-tuning on the initial MetaMath dataset, which resulted in the acquisition of some mathematical reasoning abilities. Furthermore, its base model, the Mistral-7B, demonstrated favorable generation capabilities for a model of its code size, as evidenced by its mark on the HumanEval benchmark. Finally, the relatively small size of the model, 7B, was a significant factor, as computational resources were limited. Nevertheless, it was necessary to achieve further reduction achieved by applying 4-bit quantization. To further reduce the number of parameters required to be fine-tuned, a Low-Rank Adaptation (LoRA) [Hu et al., 2022] method was employed. LoRA reduces the number of parameters by decomposing a large matrix into smaller low-rank matrices. Despite these reductions, the batch size during fine-tuning was constrained to a very small size due to the limited CUDA memory.
Three models were trained with different hyper-parameters for a different number of training steps. Following figure 7.1 illustrates the training loss, number of training steps, and learning-rate.
All models were fine-tuned using LoRA r=64, a=64, lora-dropout=0.05, on initial 1,500 tokens, which are sufficient, as can be seen from the dataset analysis. Main differences are as follows:

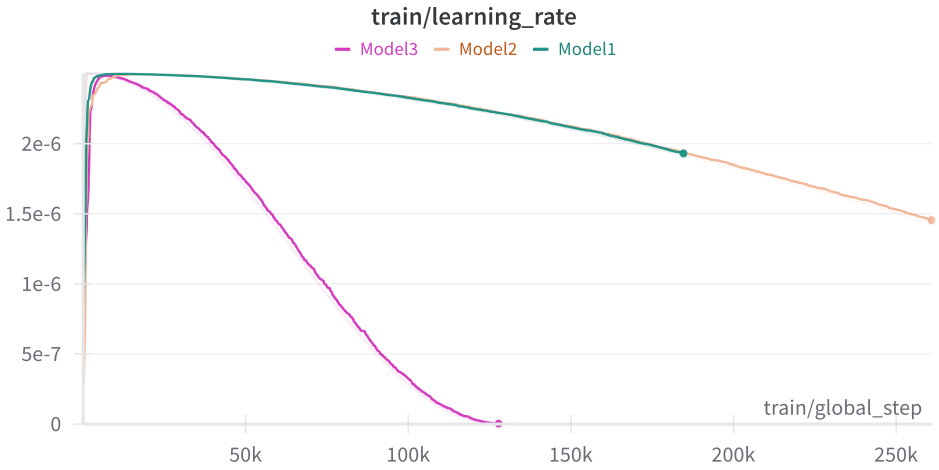
- Model-1 - Trained on Question, Answer, Solution. Batch size 1. Latest checkpoint 180,000 steps.
- Model-2 - Trained on Question, Solution. Batch size 1. Latest checkpoint 260,000 steps.
- Model-3 - Trained on Question, Solution. Batch size 3. Latest checkpoint 120,000 steps.



(a) : Exact training loss.



(b) : The training loss was smoothed and cut to start from 50,000 steps to illustrate the differences in training loss among the models in a more comprehensible manner.



(c) : Learning-rate change during training. All models started with from 500 to 1000 warm up steps.

Figure 7.1: Fine-tuning process

7.2 Comparison

Explanations for column names from this section:

- Checkpoint - Number of steps a model was fine-tuned for.
- Textual Answer % or Answer % - The percentage of rows in which the correct answer value was reached in the textual reasoning (Answer) section.
- Correct - The number of rows in which the code yielded the correct answer value.
- ME - Mathematical Error. The number of rows in which the code yielded the wrong answer value.
- CE - Code Error. The number of rows where the code cannot be executed without an exception, and is not successfully auto-corrected to fix the issue.
- Code Correct % - $\frac{Correct}{Correct+Wrong+Error}$
- Code + Answer % - Percentage of correct answer values, but in cases where there is an answer value in textual form available in the textual reasoning answer section, and the code is not operational, the textual answer value is considered the final answer value.
- No Errors - Similarly to Code Correct, but $\frac{Correct}{Correct+Wrong}$ instead.

Prior to training a verifier model and implementing a custom sampling strategy, it is essential to compare these three models. This was done at various checkpoints during the training process, as training loss does not sufficiently reflect the performance of the model on test data. The necessity to execute generated code makes testing various checkpoints on benchmarks it the only available method to measure the performance of the model. The following table 7.1 illustrates the performance of different models on benchmarks. Due to the way parallel evaluation was implemented at the time, the amount of examples used for evaluation varies slightly. All solutions were generated in a deterministic way without any sampling by greedy decoding.
In order to attempt to rectify some of the more common exceptions, an algorithmic code auto-correction was employed. For instance, it is not uncommon for generated code to crash while attempting to use an undefined *"variable_name"* when previously defined a *"variable_name_value"*. This can be automatically detected and corrected, increasing the performance by a few percent. Other common issues that auto correction attempts to address include:

- usage of sp.cosine or sp.sine instead of correct sp.cos, sp.sin
- incorrectly attempting to access the value of e by "e_value" instead if sp.E
- forgetting to substitute a known value in the expression of the final answer
- attempting to use .subs on something that is not a SymPy expression

As illustrated in the 7.1 table, there is a clear advantage to generating a complete reasoning process in natural language beforehand by generating the answer section before generating the solution section. This can be observed in the difference between the two rows that generate textual answers and the rest. It can be noted that model-1 that

Model and Checkpoint	Textual Answer %	Correct	ME	CE	Code Correct %
Model-1 160,000	×	575	422	263	45.63
Model-1 180,000	×	569	407	264	45.89
Model-1 140,000	68.06	683	319	238	**55.08**
Model-1 180,000	66.64	**696**	330	284	53.12
Model-2 60,000	×	643	435	**222**	49.46
Model-2 130,000	×	622	435	243	47.85
Model-2 200,000	×	636	401	263	48.92
Model-2 260,000	×	589	427	284	45.31
Model-3 60,000	×	439	395	466	33.77
Model-3 110,000	×	440	**275**	485	36.67
Model-3 120,000	×	426	300	553	33.31

Table 7.1: Model performance metrics across different checkpoints evaluated on GSM8K

generates textual answers beforehand reaches the same quality earlier than model-2 that generates without it. Despite reaching comparable levels of train loss, model-2 outperforms model-1 when it is prohibited for model-1 to generate textual answers. However, when model-1 is permitted to generate textual answers in advance, it significantly outperforms model-2. It can be observed that, in all three models, the increase in performance is accompanied by a reduction in the number of mathematical errors and an increase in the number of both correct outputs and code errors. Furthermore, it can be observed that the textual answer still performs better than the code, but worse than the initial MetaMath-Mistral model. The relationship between the textual answer and code solution is explored in more detail in the next table 7.2.

Model and Checkpoint	Textual Answer %	Correct	ME	CE	Code Correct %
Model-1 140,000	model generated 68.06	683	319	238	55.08
Model-1 180,000	model generated 66.64	696	330	284	53.12
Model-1 160,000	ground truth answer	773	**249**	268	59.92
Model-1 180,000	MetaMath-Mistral 74.12	**878**	269	**167**	**66.81**

Table 7.2: The relationship between textual answer and code solution evaluated on GSM8K.

As illustrated in the table 7.2, the quality of the textual answer has a direct impact on the performance of the model in the subsequent solution section. Nevertheless, the relationship between the quality of the textual answer section and the subsequent solution section is not straightforward. Providing the model with the ground truth textual answers, where all of the answers are correct, does not result in a nearly as significant improvement of the solution section as does providing the model with textual answers generated by MetaMath-Mistral 7B, the model-1 is based on. The reason for this phenomenon is unclear. The training data originated from the same sources as the ground truth data, not from the MetaMath-Mistral 7B. Therefore, the hypothesis that the model was trained on this data is not applicable. An alternative hypothesis is that the discrepancy can be attributed to a different checkpoint. However, previous analysis suggests that the differences between different checkpoints are not as pronounced as in this case. It is possible that the answers generated by MetaMath-Mistral contain some additional information beneficial for the solution. In this instance, the MetaMath-Mistral 7B textual answers were generated without the use of a calculator, in a manner analogous to how model-1 would generate them, it achieved 74.12% accuracy in textual form.

Checkpoint	Answer %	Correct	ME	CE	Code %	Code + Answer %	No Errors
140,000	68.06	683	319	**238**	55.08	63.7	68.16
180,000	66.64	696	330	284	53.12	63.89	67.84
160,000	**100**	773	249	268	59.92	**78.21**	75.64
180,000	74.12	**878**	269	**167**	**66.81**	72.07	**76.54**

Table 7.3: The relationship between textual answer and code solution evaluated on GSM8K by model-1. Compared to the ground truth textual answers (100) and textual answers generated by MetaMath-Mistral model (74.12).

The table 7.3 illustrates two potential methods of addressing the code error cases that do not necessitate additional computations. The strategy of disregarding code error cases or replacing them with textual value responses is not unreasonable, as errors present a better alternative to mathematical errors. When code errors are encountered, they can be rectified, whereas mathematical errors return a valid answer value and therefore have no way of being distinguished from the correct answers.
In all cases, with the exception of the ground truth case, when the rows containing code errors are entirely disregarded, the accuracy is higher than both the percentage of correct answer values in the answer section and in the solution section.
For subsequent evaluations, a model-1 with checkpoint 180,000 will be employed, as the textual answer section proved to be essential for performance. From now on the model-1 180,000 will be called SymPy-Mistral for simplicity.

Chapter 8
Arithmetic

To demonstrate the advantages of performing arithmetic computations via executable code a small MATH401-llm dataset [Yuan et al., 2023] has been used, due to the lack of a universally agreed upon arithmetic benchmark. The dataset is described in greater detail in the chapter dedicated to various benchmarks 2.3. Despite the considerably smaller size the SymPy-Mistral model achieves SOTA results on simple arithmetic benchmark comparable to and in some cases even surpassing much larger models, that get the result by generating the next token. The fact that the SymPy-Mistral 7B model outperforms numerous larger models provides compelling evidence that the generation of executable code code can effectively address the issues associated with arithmetic computations. Following figure 8.1 contains a comparison of SymPy-Mistral model with other LLMs on the MATH401-llm arithmetic dataset. The results data for the models other than SymPy-Mistrlal were obtained from the GitHub[1] page of the dataset authors and their original paper [Yuan et al., 2023].

Name	Size	E	±	×	÷	^	Tri	log	Dec	Neg	Irr	Big	Long	Easy	Hard	All
GPT-4	?	✓	**99**	67	**100**	50	**68**	**76**	67	67	**100**	48	**96**	**100**	**67**	**84**
ChatGPT	?	✓	97	65	80	50	44	56	67	67	64	40	68	**100**	49	74
InstructGPT	175B	×	83	59	80	36	8	16	64	64	36	4	24	92	22	57
CodeX	175B	✓	36	27	8	10	8	0	25	25	12	0	0	40	4	22
Galactica	120B	✓	69	43	24	44	16	0	57	57	28	0	24	78	12	45
LLaMA	65B	✓	44	35	8	22	8	0	41	41	20	0	4	52	5	28
OPT	175B	✓	33	35	4	12	0	4	25	25	8	0	0	41	2	22
GPT-Neox	20B	✓	51	48	4	40	4	0	43	43	20	0	8	66	4	35
GLM	130B	✓	39	31	8	22	0	0	29	29	24	0	8	46	5	26
BloomZ	176B	×	23	37	12	30	8	0	43	43	20	0	8	39	6	22
Bloom	176B	×	21	37	12	30	0	0	37	37	16	0	0	37	4	20
TO++	11B	×	6	3	0	6	8	0	3	3	4	0	0	7	2	4
Flan-T5	11B	×	1	13	4	0	0	0	11	11	8	0	0	6	2	4
SymPy-Mistral (My)	7B	✓	84.57	**86.4**	90	**80**	28	52	**81.33**	**84**	68	**72**	92	94.5	**66.66**	80.54

Figure 8.1: Column E indicates whether the model is capable of solving the Euler equation. The following columns demonstrate the accuracy of the model when applied to the combination of expressions containing the specified operators or functions. The "Dec", "Neg", "Irr", "Big", and "Long" columns demonstrate the accuracy on decimal, negative, irrational, big numbers, or long expressions, whereas "Easy" and "Hard" the accuracy on combination of the easy and hard expressions. Which expressions are considered to fall under each category can be found in the original paper [Yuan et al., 2023] or in the dataset overview in this section 2.3.

[1] `https://github.com/GanjinZero/math401-llm`

8.1 Detailed analysis

The capacity to differentiate between instances where the model fails to resolve the expression due to mathematical error vs. failing to generate a code that would provide the solution allows for the separate consideration of those cases. The case of code error is generally a better result than giving a valid mathematically incorrect result, as it allows for the problem to be addressed by a different method or for the parameters to be modified and another attempt to be made.

Name	Size	Errors	E	±	×	÷	^	Tri	log	Dec	Neg	Irr	Big	Long	Easy	Hard	All
SymPy-Mistral	7B	✓	✓	84.57	86.4	90	80	28	52	81.33	84	68	72	92	94.5	66.66	80.54
SymPy-Mistral	7B	×	✓	91.35	93.1	95.74	93	35	76.47	93.84	93.33	85	94.73	95.83	97.42	77.9	88.73

Figure 8.2: Results of SymPy-Mistral model on MATH401-llm arithmetic dataset [Yuan et al., 2023], with and without errors accounted for. The table facilitates the differentiation of these cases by comparing them separately, and in one case ignoring instances where the model failed to provide functioning code.

The model frequently fails due to an attempt to solve the equation by modifying it directly, rather than using the solve function or attempting to use SymPy in a manner that is not intended (using an incorrect function, misspelling the function name, attempting to extract an element from a list that is not present in the output of the function). This indicates that the model requires further training on more SymPy data. Alternatively, the model occasionally fails due to the omission of some portion of the result, or in cases of large inputs or exponential, it may become stuck in the initial text section by repeating the numbers infinitely. One potential solution to this particular failure would be to set a limit to the length of this particular section separately from the maximum length of the entire output. The model struggles the most with trigonometry, logarithms, and irrational numbers. These instances require the use of less common SymPy functions. This supports the idea that the inclusion of more SymPy examples with a greater variety of use cases and functions would be beneficial.

```
# Given values
equation_value = 447310251389 - 473614891549
# Define symbol
x = sp.symbols('x')
# Solve for x
solution = sp.solve(equation_value, x)
# Extract the numerical solution
x_value = solution[0]
print(x_value)
```

Figure 8.3: An example of wrongly used SymPy. In this instance, SymPy attempts to solve the *equation_value* as if it were an equation, which it is not. In order to transform it into an equation with the symbol x, a *sp.Eq(equation_value, x)* is required.

```
# Given values
equation_value = 447310251389 - 473614891549
# Define symbol
x = sp.symbols('x')
# Create an equation
equation = sp.Eq(equation_value, x)
# Solve for x
solution = sp.solve(equation, x)
# Extract the numerical solution
x_value = solution[0]
print(x_value)
```

Figure 8.4: The correct usage of SymPy on the same instance for comparison.

8.2 Settings

These results were achieved by prompting model with the following prompt format:
"Question: **query** x. What is the value of x?\n\nAnswer:"
where query is the query from the dataset as follows.
"Question: $e^{i\pi} + 1 = x$. What is the value of x?\n\nAnswer:"
This transforms these expressions into MWP format containing a question, that the model requires for working due to being trained on MWPs.

The usual custom sampling is employed on samples generated in groups of 100 new tokens at a time, with the following parameters, without the assistance of the verifier. The model variant is the SymPy-Mistral checkpoint 180,000, with the generation of the answer section enabled.

Listing 8.1: Arithmetic generation parameters

```
temperature=2.5,
num_beams=2,
repetition_penalty=1.3,

```

Chapter 9
Verifiers

Verifiers are methods employed to evaluate the completion generated by the model, with the objective of enhancing its performance. As the number of generated samples increases, the probability of at least one correct solution being among them also grows as indicated by the pass@k metric with higher temperature [Chen et al., 2021]. Therefore, if there were a verifier method to determine the correct sample from the pool of generated solutions, it would often significantly enhance the performance of the model. These approaches are commonly employed in both models for code generation and solving MWPs, and have been demonstrated to be highly effective. This chapter provides an overview of existing techniques and the technique used in this work.

9.0.1 Verifiers in MWPs

In the paper "Training Verifiers to Solve Math Word Problems" [Cobbe et al., 2021], which introduced the GSM8K dataset, the authors proposed verifiers as a means of evaluating and ranking model completions. The paper compared two possible verifiers: a solution-level verifier, which makes a scalar prediction for a whole generated solution, and a token-level verifier, which makes a scalar prediction after each token in the solution. In the original paper the token-level verifier outperformed solution-lever verifier, despite exhibiting slower training. Additionally, it exhibited less overfitting.

The authors also found that a combination of a large generator model (175B) with a small verifier model (6B) significantly outperforms the combination of a small generator model (6B) with a large verifier model (175B). This directly contradicts the results of a more recent study [Liu et al., 2023]. It also demonstrated that their sequence-to-sequence verifier that supervises each token (token-level verifier) outperforms the solution-level verifier. However, in their case, the combination of a smaller generator model (125M) with a larger verifier model (1.3B) demonstrated significantly higher accuracy than the combination of a larger generator model (1.3B) with a smaller verifier model (125M).

I propose a few possible explanations for this phenomenon:

- The datasets employed for training of these models are not identical, and the sizes of the datasets differ significantly. The initial work utilized the GSM8K dataset, which consisted of 7,473 training samples. In contrast, the later work used the TinyGSM synthetic dataset, which contained 12.3M GSM8K-style problems paired with Python code. The difference in the size of the datasets may provide an explanation for the continued improvement of a model with a greater number of parameters in the case of a larger dataset.

- The models employ different formats to achieve their results. The initial work generated step-by-step solutions in natural language, whereas the subsequent work generated Python code. A solution that follows a more rigid format of a programming language in contrast to natural language may influence the requisite size of the verifier to learn a heuristic to effectively rank the solutions.

- There is a significant difference in model sizes. In the case of the initial work, the smaller model encompasses 6B parameters, whereas the larger model encompasses 175B parameters. In the subsequent work, the smaller model encompasses 125M parameters, while the larger model encompasses 1.3B parameters. This contrast is significant, as even the model of 1.3B parameters is still considerably smaller than the smaller model of 6B parameters. The discrepancy may be attributed to the differing capabilities of models with varying sizes.

Further research is necessary to explain the phenomenon in question.

9.0.2 GPT-4 Code: CSV and self-debugging

The combination of GPT4-Code's self-debugging and explicit code-based self-verification (CSV) [Zhou et al., 2024] represents an example of a verifier that employs code generation capabilities. The self-debugging process involves the model rewriting the block of faulty code based on the exception received upon execution. However, self-debugging is not capable of verification of the reasoning steps or the final answer, so the CSV is required. CSV, in contrast to the use of a separate model, makes use of GPT4-Code's code generation capabilities by prompting model to verify the solution with an additional code block that outputs a Boolean value of either "True" or "False." When the CSV returns a value of "False", the model can provide an improved alternative. The outcome of self-verification is also incorporated into the determination of the reliability of the solution for the weighted majority voting strategy. In contrast to the simple majority voting strategy, which relied solely on the frequency of the answers, this strategy assigns weights to the solutions based on their reliability.

9.0.3 CodeT

CodeT [Chen et al., 2022] is a verifier for code generation. It employs a pre-trained language model for code generation to generate unit tests for the problem at hand. If multiple code solutions pass the same tests, they are considered to have the same functionality. Solutions that pass more tests are considered to be better. This is used to create consensus sets and to rank them by the number of solutions and tests they contain in order to select the best solution from the best consensus set. As little as 10 test cases have shown to be sufficient to enhance the pass@1 metric by 9.5% when using code-davinci-002 as a generator model.

9.0.4 Reinforcement learning

In reinforcement learning, a critic model that evaluates the quality of the policy model outputs is effectively a verifier of the policy model. Verification and reinforcement learning are explored in the MATH-SHEPHERD [Wang et al., 2023] where intermediate reasoning steps are assigned score based on their potential to lead to the correct final answer.

9.0.5 Verifiers in this work

Due to time and computational constraints, the requirements for the verifiers were set to be minimalistic and quickly available, yet effective enough to be of practical utility. Furthermore, it was necessary for the verifier to be compatible with the custom sampling, which is explained in the chapter 10. A decision has been made to utilise a small encoder model with a classification head as a solution-level verifier. Its embeddings can be employed in the custom sampling process, and a classification head can be leveraged to assess the solution's quality. The verifier was fine-tuned from longormer-base-4096 [Beltagy et al., 2020], which was chosen as a base for the verifier because of its support of sequences of sufficient length and meets the requirements of size and architecture.
Longormer [Beltagy et al., 2020] is an open-source transformer-based model scalable for processing long documents by combining sliding window and global attention. The longormer-base-4096 supports sequences of length up to 4,096. In this work the longformer-base-4096 encoder is used as for fine-tuning with a custom classification head for 2 classes. In total the size of the final verifier is 148,660,994 parameters.
The Longormer verifier was fine-tuned for classification on the data from the MetaMath-GSM8K-SymPy dataset, with the correct class being data from the MetaMath-GSM8K-SymPy dataset and the incorrect class being data from the example combination of the MetaMath-SymPy-Wrong and MetaMath-Error datasets. Three models were created in this manner.

- Verifier-l1 - Fine-tuned to classify based on only the solution section.
- Verifier-l2 - Fine-tuned to classify based on the question and solution sections.
- Verifier-l3 - Fine-tuned to classify based on the question, answer and solution sections.

These verifiers were evaluated by classifying outputs created by the SymPy-Mistral on GSM8K test set.

Verifier	Percentage of correctly classified
Verifier-l1	51.75
Verifier-l2	61.22
Verifier-l3	65.19

Table 9.1: Classification Accuracy of Verifiers.

From this, the effectiveness of the verifiers might appear questionable. However the verifiers primary objective is not to ascertain the correctness of an output; rather, its goal is to identify which output of a group of highly similar outputs is superior. Demonstrating this on a single set of outputs for the same problem would be challenging. Instead, it will be demonstrated indirectly by demonstrating the statistical difference in classification certainty of true positive, false positive, true negative, and false negative examples.

The table 9.2 demonstrates the statistical difference of classification certainty for each model. A predicted value, achieved by applying softmax function to verifier outputs and taking the value corresponding to the class "correct" was employed as a probability of sequence correctness. These values were multiplied by 100 to get percentages for clarity. In the case of positive outcomes, the value range is between 50 and 100, while in the case of negative outcomes, the value range is between 0 and 50. In the optimal scenario, true positives are always assigned a value of 100, while false positives are assigned a value of 50, and true negatives are always assigned a value of 0, while false negatives are assigned a value of 50. Verifiers are tasked with comparing sequences that are relatively similar.

Mean Score	Verifier-l1	Verifier-l2	Verifier-l3
True Positive	51.57	78.59	90.78
False Positive	51.55	73.88	89.43
True Negative	49.34	10.21	11.99
False Negative	49.36	20.05	14.91
Median Score	Verifier-l1	Verifier-l2	Verifier-l3
True Positive	51.48	81.14	96.29
False Positive	51.54	73.72	95.02
True Negative	49.54	04.01	6.68
False Negative	49.58	11.89	10.55
Mean Difference of True and False Positives	0.02	4.71	1.35
Mean Difference of True and False Negatives	-0.02	-9.84	-2.92
Median Difference of True and False Positives	-0.06	7.42	1.27
Median Difference of True and False Negatives	-0.04	-7.88	-3.87

Table 9.2: The mean and median scores of true positive, false positive, true negative, and false negative examples demonstrate that answers correctly classified as positive are statistically assigned greater value than false positives, whereas answers correctly classified as negative are statistically assigned lower value than false negatives.

Consequently, a sequence that has diverged from being correct only slightly will likely still receive a value similar to that of a correct sequence it diverged from. However, if the verifier is at least slightly more uncertain whether the sequence is correct or not and thus the value is a bit closer to 50, while the values of correct sequences are further from 50, the verifier is useful.

Initially, the mean and median of the predicted values in true and false, positive and negative cases were calculated. Subsequently, differences of true and false positives and negatives were calculated by subtracting the false mean or median values from the true mean or median values. For the positive case, the higher the achieved value, the better; conversely, for negatives, the lower the value, the better. The results demonstrate that verifier-l1 is unlikely to be useful as the difference between true and false values is negligible. Verifier-l2 and verifier-l3 demonstrate promising results, with verifier-l2 achieving substantially higher differences between true and false values. Nevertheless, this does not necessarily imply that it will perform better, as verifier-l2 has a lower percentage of correctly classified samples than verifier-l3 (61.22 vs. 65.19), as illustrated in table 9.1. In the case of verifier-l3, the samples that were misclassified by verifier-l2 with a values close to 50 were likely correctly classified by verifier-l3, but still with value close to 50, causing the observed effect. This approach is not useful for comparing the actual performance of the verifiers. It is only useful for deciding whether or not they are likely to work.

Chapter 10
Custom sampling

In order to enhance the generation process, a custom sampling process was created that leverages the verifier. The sampling process can be simplified into several steps, with greedy best sampler and parameters N = 5, K = 200, similarly to how illustrated on the right of the figure 10.2:

- 1. For given question start generating 5 sequences in parallel by generating 200 tokens at a time in each of them.
- 2. When reaching the end of the SymPy code block in each sequence, ask the verifier model what is the probability of each sequence to be correct.
- 3. Based on these verifier scores, sampler will reject the unlikely sequences by replacing them with copies of sequences with higher probabilities.
- 4. Continue generating 200 tokens at a time, if encountered print() statement remove the sequence into the output list, the rest continues from the step 2.

Two diagrams were created in order to explain the process in greater detail. Diagram 10.1 shows the general pipeline, whereas diagram 10.2 shows a simplified example of the process.
On the right of the diagram 10.2 is a visualization of a similar example as described above, without the texts, showing only the tokens of the beginning and ending blocks. The texts are replaced by dots, indicating the different lengths of the generated blocks and the divergence of the sections during the process. In the diagram example the sequences start with an id for visualisation purposes. For example the fifth sequence starts with id 5., which is extended after the sampler step, for example 5.2., to show that the current second sequence has diverged from the copy of the initial fifth sequence by the previous second sequence with id 2. being rejected by the sampler filter with the copy of previous fifth sequence that remained and is now with id 5.5.. On the left are the steps from the diagram 10.1 rewritten linearly, with repetitions instead of loops. The examples on the right roughly follow the steps on the left.
Diagram 10.1 illustrates that each row of GSM8K test data is processed individually through a pipeline. This pipeline first copies the same row N times, and then uses the SymPy-Mistral model to generate K tokens for each of them repeatedly until each contains two [|EndOfBlock|] tokens. This process results in the generation of the first code block. Subsequently, these N sequences are sent to the verifier model, which assigns a score to each of them. Any generated content subsequent to the second [|EndOfBlock|] token is then removed. This is accomplished subsequent to the application of the verifier, which serves to verify the potential future continuation from this point. Table 6.4 indicates that the block pairs following the first two are typically shorter, or approximately equal to

100 tokens in length. At approximately K = 200, the verifier is provided with at least a block look-ahead of additional information, which allows it to reject sequences that are likely to continue in a clearly wrong direction. Subsequently, all sequences that already contain print statements and therefore should not be generated further were removed into the output list. The number of sequences is then reduced to match the new number of sequences. All remaining sequences are then getting the code generated until that point executed, and the latest code expression extracted. These sequences, along with their scores and extracted expressions, are then passed to the sampling filter. The sampling filter has four options:

- Greedy best - Greedy best sampler considers only the verifier scores and replaces all N sequences with the copies of the sequence with the highest score.

- Greedy best popular - Greedy best popular filters in two steps. Initially, all unique expressions in the code execution outputs are evaluated, and the sequences with the most frequent expression for the current step are selected. This process is analogous to majority voting. Subsequently, in the second step, the sequence with the highest verifier score is selected all sequences are replaced with it as in greedy best.

- Cosine similarity - In this step, the embeddings of the verifier are used as vectors to compute the cosine similarity between all pairs of sequences. The algorithm starts by selecting the sequence with the highest verifier score, and finds a sequence with the lowest cosine similarity, and selects it by removing the other sequence from both similarity lists. In this way, it selects N sequences. Some sequences may be selected repeatedly, while some may never be selected. The reasoning behind this approach is that selecting the sequences with the most variety increases the chances that some of them will reach the correct answer.

- Binary tournament - Binary tournament randomly selects N pairs of sequences and keeps only the one with the higher verifier value. This allows the diversity of sequences to be maintained while still eliminating the weakest sequences.

After the sampler filter is applied, additional K tokens are generated, and the process is repeated each 2 [|EndOfBlock|] tokens until the token limit is reached or until no sequences remain due to all of them reaching the print statement.
Custom sampling requires model to generate different sequences to work. This is achieved by generating with parameters as for example used for most test in chapter 11: *num_beams* = 3, *temperature* = (from 0.8 to 2.5), *repetition_penalty* = (from 1.2 to 1.5).
The diagram 10.2 illustrates the sampling process using the greedy best popular filter. The process is presented in a linear, step-by-step format, with concrete examples displayed on the right side. While the examples do not illustrate the entirety of the process, they serve to illustrate the key stages. The process terminates when the final result is determined. It can be seen that the output list contains different solutions. After the custom sampling process, a verifier is applied once more to select the output with the highest score, or the output with the highest score among the most popular output expressions. In the aforementioned example, the application of a verifier to select the highest-scored output among the most popular will result in the selection of the output with a score of 0.87 and an output value (pi/2 + 1.2)/10. These results are then evaluated in the next chapter 11.

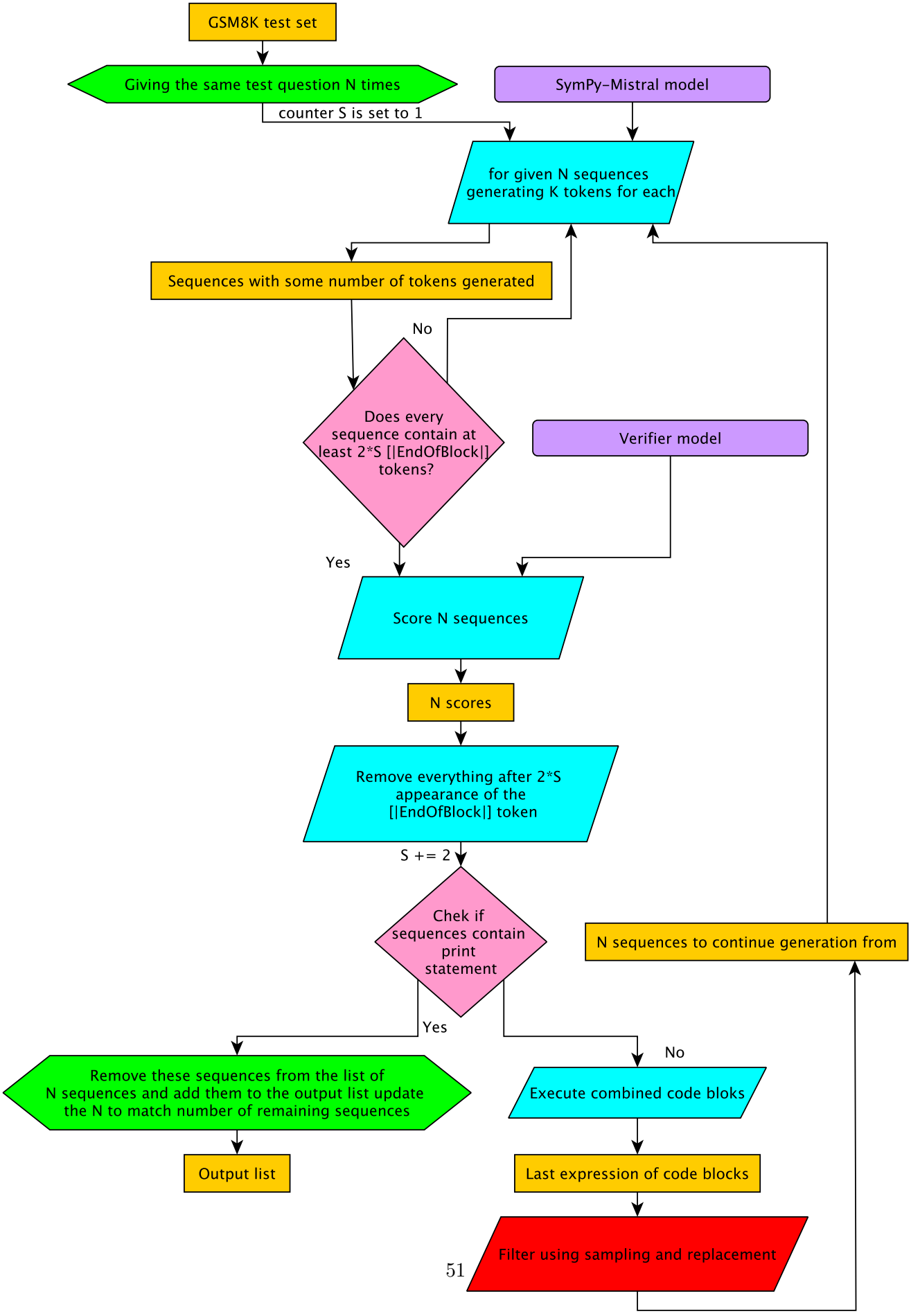


Figure 10.1: A diagram illustrating custom sampling process.

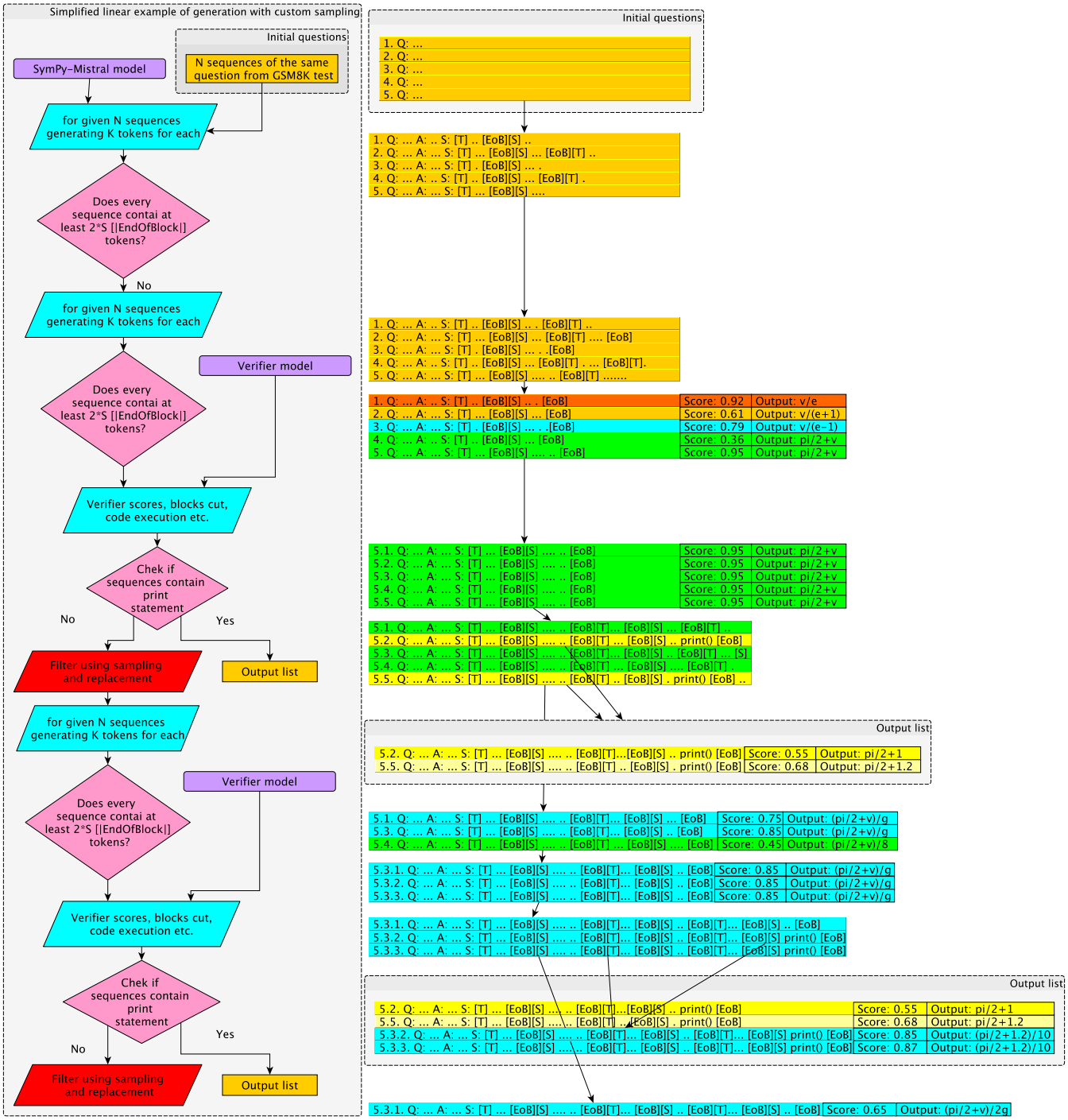


Figure 10.2: A diagram illustrating simplified example of the custom sampling process. Q, A and S indicate question, answer, and solution sections. [T], [S], [EoB] are simplified versions of [|Text|], [|Sympy|], [|EndOfBlock|] tokens. Equivalent sequences are represented by the same color.

Chapter 11
Evaluation

This chapter provides an in-depth evaluation of the model and a comparison of verifiers and sampling filters, referred to in this chapter as samplers for short. The terminology and column names used in this chapter is the same as in chapter 7, except the addition of Top and Verifier columns. That is:

- Checkpoint - Number of steps the model was fine-tuned for.
- Textual Answer % or Answer % - The percentage of rows in which the correct answer value was reached in the textual reasoning (Answer) section.
- Correct - The number of rows in which the code yielded correct answer value.
- ME - Mathematical Error. The number of rows in which the code yielded wrong answer value.
- CE - Code Error The number of rows where the code cannot be executed without an exception, and is not successfully auto-corrected to fix the issue.
- Code Correct % - $\frac{Correct}{Correct+Wrong+Error}$
- Code + Answer % - Percentage of correct answer values, but in cases where there is an answer value in textual form available in the textual reasoning answer section, and the code is not operational, the textual answer value is considered the final answer value.
- No Errors - Similarly to Code Correct, but $\frac{Correct}{Correct+Wrong}$ instead.
- Top - Two selection methods exist for selecting the top final answer from all generated samples, designated as "Best" and "Majority." In the "Best" method, the output with the highest Verifier score is selected. In contrast, in the "Majority" method, the output with the highest Verifier score among the most common answer values is selected.
- Verifier - Two values separated by a slash symbol are used to indicate the verifier model employed for sampling during generation and the verifier model used for selecting the final output. For example Top = Best, Verifier = l3/l2 signifies that verifier l3 was utilized during the generation process for the sampling procedure, while verifier l2 was employed to select the output with the highest score as the final result.

The tests were conducted on the GSM8K test set in accordance with the sampling process pipeline, as described in chapter 10. The outputs were generated with a temperature parameter of 0.8, unless specified otherwise. Different combinations of verifiers were employed to compare their performance. The pipeline was applied twice with N = 5 otherwise

written as $2 * (N = 5)$, generating 10 outputs in total, which were subsequently used to select the final answer. This is not equivalent to N = 10, as for example Greedy Best sampler keeps very low variety of outputs, but these 2 N outputs. However, these two N outputs are covered separately, creating two sets of five outputs that are highly similar within their respective sets. Nevertheless, the differences between sets can be significant. Baseline results for comparison will be used results achieved in the chapter 7, namely:

- baseline-1 - MetaMath-Mistral geberated textual answer without calculator (textual answer accuracy of 74.12%)
- baseline-2 - Greedy decoding with answer section generation (code accuracy of 54.12%)
- baseline-3 Greedy decoding with MetaMath-Mistral textual answer (code accuracy of 66.81%)

11.1 Binary Tournament, N value and temperature

Table 11.1 shows data for the Binary Tournament sampling experiments. The binary tournament sampler outperformed both code baselines 2 and 3 in case of majority vote selection, but never achieved the same quality of textual answers.
The binary tournament comparison also tested the effects of varying the number of generated samples as well as the effect of temperature increase. The l3 sampling was performed with the usual $2 * (N = 5)$, but with the temperature set to 2.5. The l2 sampling was performed with $N = 8$, but with the standard temperature of 0.8. The results of the l3 sampling demonstrated a substantially lower accuracy of textual answers than those of l2 sampling, which was unexpected given that the l2 verifier was not trained with textual answers, while l3 was. This discrepancy is likely due to the temperature being set too high, as the model appeared to be highly sensitive to temperature changes. Interestingly, the accuracy of code outputs declined only when the final answer was selected based on the highest score when sampling with l2 verifier and higher temperature. However, when selecting by majority vote, the accuracy of code increased even surpassing the alternative which used l3 verifier sampling and lower temperature. This suggests that having more varied outputs caused by higher temperature might increase the amount of arithmetic errors in the textual answer section, as well as an elevated probability of encountering an adversarial example among outputs that would lead to the verifier making a wrong choice. However, the observed improvement in performance with majority voting, and a significant decline in the number of outputs that result in error due to code error, indicates that it also increases the probability of the model generating more samples with working code, including both correct and incorrect answer values. However, it is challenging to ascertain whether this is a beneficial outcome, as the increase in mathematical errors with only an insignificant improvement in majority voting code accuracy (66.94% vs 67.48%) is not a favorable outcome. It is possible to regenerate or fix code errors by other means, as demonstrated by the ”Code + Answer %” section (71.72% vs 68.61%), where there is no improvement, but a decrease. In contrast, mathematical errors cannot be corrected because it is not known if they are correct or incorrect. The data suggests that higher temperatures are detrimental for binary tournament. Nevertheless, this may not be the case for larger N or different samplers, as majority voting may perform better with a higher variance of outputs when there is a greater number of outputs.
In order to approximate the effect of larger N, an additional experiment was conducted

by combining the outputs collected in previous cases. The results of this additional experiment were then tested on 18 outputs (8 sampled with low temperature, 10 sampled with high temperature). The selection of the highest-ranked output demonstrated a slight improvement in comparison to the high-temperature case, but a decline in comparison to the low-temperature case. However, the majority vote selection demonstrated an improvement in comparison to both cases. Despite the fact that there were fewer code error cases, there were also fewer answers with mathematical errors. This coupled with higher "Code + Answer %" supports the idea that larger N coupled with majority voting might be beneficial, but further experiments with different temperatures and larger N would be necessary to test this hypothesis.

Top	Verifier	Correct	ME	CE	Code %	Answer %	Code + Answer %	No Errors
Best	l3/l2	810	455	54	61.41	51.78	62.55	64.03
Best	l3/l3	810	454	55	61.41	52.99	62.55	64.08
Majority	l3/l2	889	372	58	67.4	57.54	68.54	70.5
Majority	l3/l3	890	371	58	67.48	58.07	68.61	70.58
Best	l2/l2	878	303	138	66.57	72.25	71.34	74.34
Best	l2/l3	873	305	141	66.19	72.18	71.34	74.11
Majority	l2/l2	883	297	139	66.94	**72.48**	71.72	74.83
Majority	l2/l3	883	297	139	66.94	72.4	71.72	**74.83**
Best	(l3+l2)/l2	831	447	41	63	54.74	63.84	65.02
Best	(l3+l2)/l3	824	449	46	62.47	54.28	63.53	64.73
Majority	(l3+l2)/l2	933	343	43	**70.74**	64.82	72.1	73.12
Majority	(l3+l2)/l3	933	343	43	**70.74**	63.91	**72.1**	73.12

Table 11.1: Comparison of binary tournament sampler with SymPy-Mistral model on GSM8K test benchmark.

11.2 Greedy Best

The Greedy Best Sampler was subjected to a single test, with the l2 verifier for generation. It also outperformed both coding baselines, while not reaching the quality of a textual answer. The code accuracy is slightly better than in the case of binary sampling when tested with the same parameters, yet the results are not significantly different. Of interest is that it seems that in case of greedy best sampling there is almost no difference between best ranked selection and majority vote selection. This is logical, given that all the outputs are essentially identical, and therefore the majority voting cannot introduce anything new.

Top	Verifier	Correct	ME	CE	Code %	Answer %	Code + Answer %	No Errors
Best	l2/l2	880	320	119	66.72	71.95	70.66	**73.33**
Best	l2/l3	892	308	119	**67.62**	**72.48**	**71.42**	**74.33**
Majority	l2/l2	890	311	118	67.48	72.55	71.27	74.1
Majority	l2/l3	890	311	118	67.48	72.55	71.19	74.1

Table 11.2: Comparison of greedy best sampler with SymPy-Mistral model on GSM8K test benchmark.

11.3 Greedy Best Popular

The Greedy Best Popular sampler was also subjected to a higher temperature of 2.5, as this approach was deemed the most likely to withstand the negative effects it might bring. The results in the table 11.3 support this assumption, as it outperforms all three baselines

as well as both Binary Tournament and Greedy Best sampling methods. The textual answer also achieved a slight improvement over the baseline. This evidence demonstrates that high temperature can be beneficial.
Despite the fact that it is a greedy approach, whereby only the final pair of blocks in the outputs differs, the higher temperature caused these final blocks to differ sufficiently for the best-ranked and majority vote selections to be distinguishable. The approach achieved approximately the same number of answers with mathematical errors as the binary tournament, while also reducing the number of code errors.
The difference in the textual answer accuracy of the l2 and l3 verifiers employed during the sampling process, and l3 verifier, demonstrates that training the verifier on answer sections can also enhance its textual answer accuracy.

Top	Verifier	Correct	ME	CE	Code %	Answer %	Code + Answer %	No Errors
Best	l2/l2	889	307	123	67.4	72.93	71.87	74.33
Best	l2/l3	887	310	122	67.25	73.09	71.95	74.1
Majority	l2/l2	897	301	121	68.01	72.55	72.4	74.87
Majority	l2/l3	897	301	121	68.01	72.86	72.71	74.87
Best	l3/l2	922	302	95	69.9	74.07	73.24	75.33
Best	l3/l3	919	311	89	69.67	74.07	72.86	74.72
Majority	l3/l2	930	298	91	**70.51**	74.15	**74**	**75.73**
Majority	l3/l3	930	298	91	**70.51**	**74.15**	**74**	**75.73**

Table 11.3: Comparison of greedy best popular sampler with SymPy-Mistral on GSM8K test benchmark.

11.4 Cosine Similarity

The Cosine Similarity sampler outperforms both coding baselines, yet exhibits a lower textual answer accuracy. It achieves results similar to those of the Greedy Best sampler, but slightly outperforms in the case of majority vote selection due to having more varied outputs.

Top	Verifier	Correct	Wrong	Error	Code %	Answer %	Code + Answer %	No Errors
Best	l2/l2	902	323	94	68.39	72.02	71.65	73.63
Best	l2/l3	896	327	96	67.93	71.8	71.27	73.26
Majority	l2/l2	913	309	97	**69.22**	72.02	**72.63**	**74.71**
Majority	l2/l3	913	309	97	**69.22**	**72.18**	72.55	**74.71**

Table 11.4: Comparison of cosine similarity sampler with SymPy-Mistral model on GSM8K test benchmark.

11.5 Comparison of samplers

The results with those specific combinations of verifiers were selected to represent their samplers due to their highest code accuracy for an equal number of output samples.

Sampler	Top	Verifier	Correct	ME	CE	Code %	Answer %	Code + Answer %	No Errors
Cosine Similarity	Best	l2/l2	902	323	94	68.39	72.02	71.65	73.63
Greedy Best Popular	Best	l3/l2	922	302	95	69.9	74.07	73.24	75.33
Greedy Best	Best	l2/l3	892	308	119	67.62	72.48	71.42	74.33
Binary Tournament	Best	l2/l2	878	303	138	66.57	72.25	71.34	74.34
Cosine Similarity	Majority	l2/l3	913	309	97	69.22	72.18	72.55	74.71
Greedy Best Popular	Majority	l3/l3	930	298	91	**70.51**	**74.15**	**74**	**75.73**
Greedy Best	Majority	l2/l3	890	311	118	67.48	72.55	71.19	74.1
Binary Tournament	Majority	l2/l3	883	297	139	66.94	72.4	71.72	74.83

Table 11.5: Comparison of samplers with SymPy-Mistral model on GSM8K test benchmark.

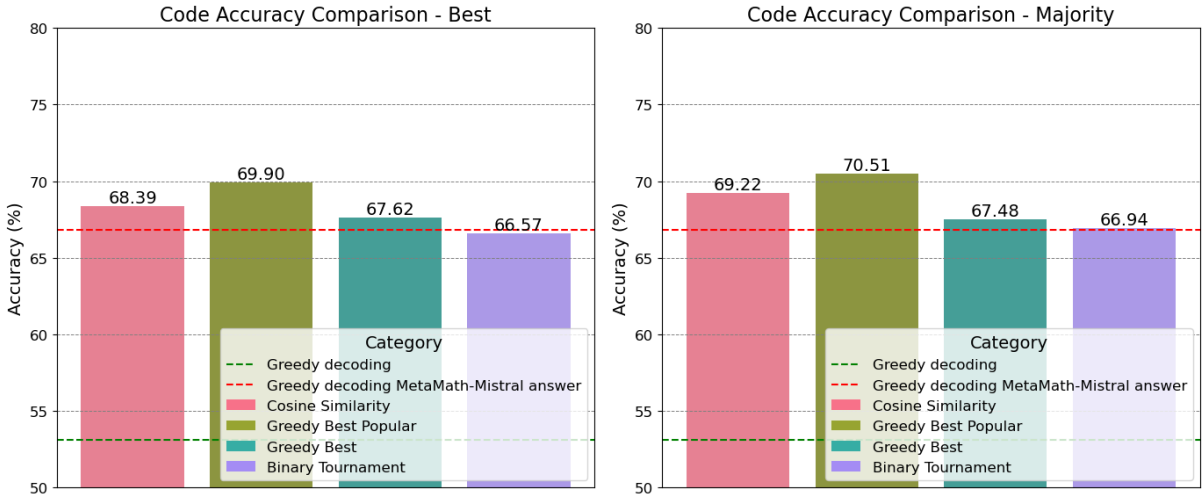


Figure 11.1: A comparison of the code accuracy among different samplers with different selection types.

Figure 11.1 illustrates the code accuracy of different samplers. In all cases, the performance of the samplers exceeds that of the baseline-2. However, in the instance of the highest-ranked selection, the binary tournament fails to outperform the baseline-3, whereas other samplers do. Only in the case of the greedy best sampler does the highest-ranked selection outperform the majority vote selection.

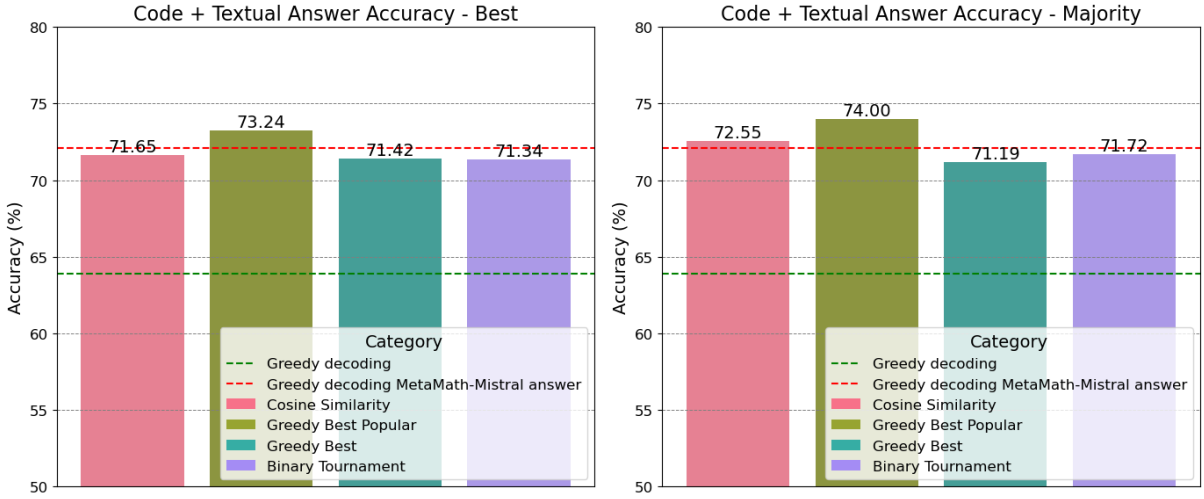


Figure 11.2: A comparison of the code + textual answer accuracy among different samplers with different selection types on GSM8K test benchmark.

In order to ascertain the accuracy of the "Code + Textual Answer", the results of the same tests with the values of their corresponding accuracy values 63.89% and 72.07% are utilized. Figure 11.2 demonstrates that only the Greedy Best Popular sampler effectively compensates for errors by utilizing textual answers in both cases, significantly outperforming the baseline. While Cosine Similarity outperforms the baseline in the majority vote case, it only manages to surpass it slightly.

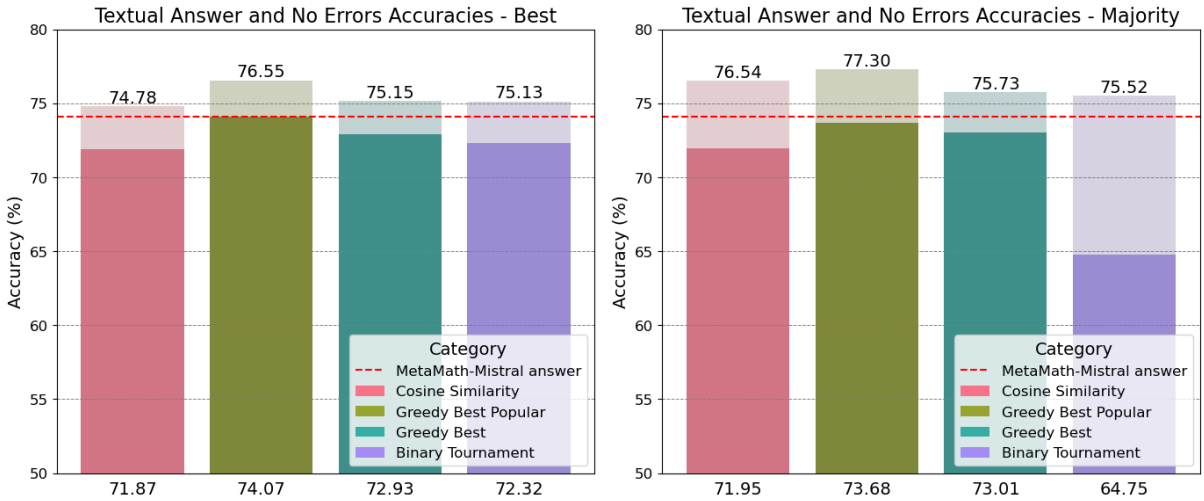


Figure 11.3: A comparison of textual answer accuracy among different samplers with different selection types on GSM8K test benchmark.

The overall accuracy of textual responses declined, with the exception of the greedy best popular sampler, which was the only one to maintain the quality of the textual response. The binary tournament appeared to decline significantly, as demonstrated by figure 11.3 in the case of the majority vote. However, this is misleading due to the example being selected based on the coding accuracy. All the accuracies of textual answers achieved with lower temperature would be 72.4, which outperforms cosine similarity.
The aforementioned figure 11.3 also illustrates the code accuracy, excluding code errors.

As a result the majority vote selection mostly outperforms the highest ranked selection. The order of samplers performance is in case of majority vote as follows:

- 1. Code - Greedy Best Popular, Code + Answer - Greedy Best Popular, Answer - Greedy Best Popular
- 2. Code - Cosine Similarity, Code + Answer - Cosine Similarity, Answer - Greedy Best
- 3. Code - Greedy Best, Code + Answer - Binary Tournament, Answer - Binary Tournament
- 4. Code - Binary Tournament, Code Answer - Greedy Best, Answer - Cosine Similarity

The Greedy Best Popular approach was identified as the most effective sampling method, consistently outperforming all other sampling and baseline methods. Cosine Similarity was identified as the second most effective option, suggesting that it might be a good idea to test it with the l3 verifier and higher temperature settings, as it seems to be a promising approach as well. Greedy Best was the third and Binary Tournament the least effective, despite using a more accurate textual answer ranking. These last two approaches do not appear to offer much scope for improvement.

11.6 Comparison of verifiers

In order to facilitate a meaningful comparison, in each case the best code accuracy example for selected combination of verifiers was selected. The resulting data is presented in the figure 11.4, with verifiers used in generation sampling arranged horizontally in rows, and verifiers used in the selection arranged vertically in columns.
The available data indicates that there is no significant difference between l3 and l2 verifiers during the selection process. However, the l3 verifier outperforms the l2 verifier during the generation sampling process.

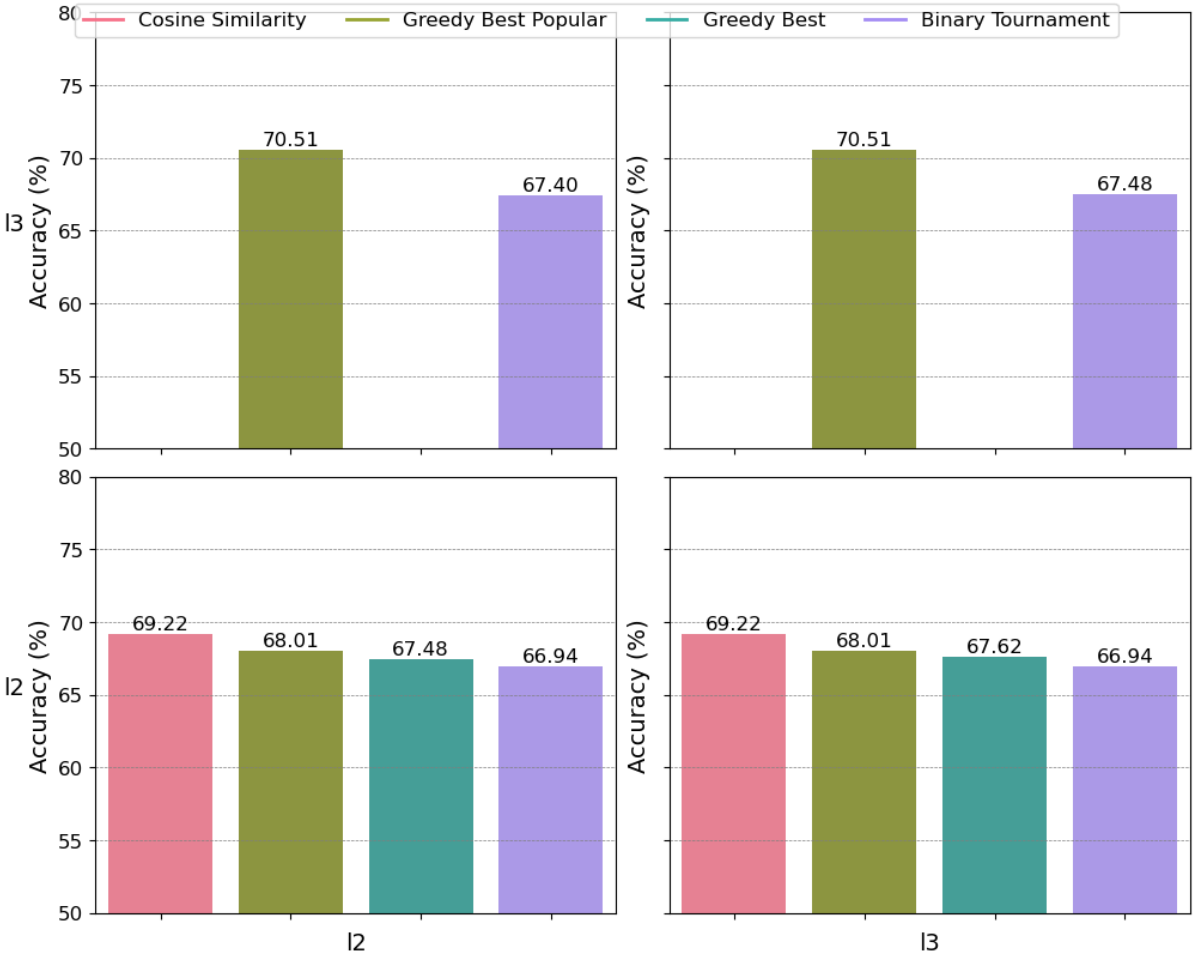


Figure 11.4: All verifier combinations compared. Sampling verifiers horizontally. Selecting verifiers vertically. Compared on GSM8K test benchmark.

Chapter 12
Future work

This chapter briefly mentions potential improvements for the approach.

- Adding irrelevant context to the questions in the dataset similarly to GSM-IC [Shi et al., 2023]. The authors have shown that LLMs are sensitive to irrelevant information. I suspect that my approach is even more susceptible to distraction by irrelevant information because SymPy-Mistral acknowledges each variable by defining separate variables and symbols for each of them in the initial blocks.
- Reformatting more of the MetaMathQA dataset with the newly available model as more high quality training data usually leads to better performance.
- Filtering the MetaMath-GSM8K-SymPy dataset, by removing the samples lowest-ranked by the verifier. This approach could potentially result in a feedback loop, where a model trained on solutions approved by the verifier would reformat additional solutions in a similar manner, and the newly trained verifier would then repeat the process. A few steps of this approach could eliminate the poor quality data and select more data with the structure that both SymPy-Mistral and the verifier work better with.
- Try the same approach, but with a token-level verifier, since it's been shown that token-level verifiers work better than solution-level verifiers. [Cobbe et al., 2021].
- Using larger verifier. Some works suggest that larger verifiers with smaller generators are better, while others claim the opposite as described in chapter 9.0.1. This thesis only tested the smaller verifier with larger generator approach.
- Implement more sophisticated automatic code fixing approaches, similar to self-debugging, to improve the ability of the model to generate SymPy code.
- Append relevant SymPy documentation hints, when the model generates faulty code to improve the ability of the model to generate SymPy code.
- Test different K value - the amount of newly generated tokens in each sampling step, as the amount of look-ahead available may affect the performance of the verifier.
- Developing a new sampling method that combines the advantages of Cosine Similarity and Greedy Best Popular samplers. Both samplers showed promising performance, and Cosine Similarity has potential for improvement.

Chapter 13
Conclusion

This thesis provides an overview of the current state of the art in the field of solving math word problems using large language models and examines some of the code generation approaches. It proposes a new format for solving MWPs by combining mathematical reasoning in natural language text with blocks of executable code and proposes a novel approach of using SymPy code, a Python library for symbolic mathematics, instead of general-purpose code used in other methods. To the best of my knowledge, this is the first work to demonstrate the advantages of a code generation approach over large language models that do not use code generation in the context of solving purely arithmetic expressions. Additionally, I have compiled links to all of the mathematical datasets and most of the code datasets that I have managed to locate. This work goes over various verifiers in both fields of math word problems and code generation and demonstrates how a minimalistic verifier can be created and employed. One of the most significant contributions of this work is the provision of several open-source datasets that are reformatted from GSM8K and MetaMathQA datasets (MetaMath-SymPy[1], MetaMath-SymPy-Wrong[2], MetaMath-Error[3] and MetaMath-GSM8K-SymPy[4]) for solving math word problems in the proposed format of interweaving blocks of textual reasoning and SymPy code. This thesis presents an analysis and methodology for the creation of the dataset. Additionally, in this thesis, I have fine-tuned and evaluated a SymPy-Mistral model based on the MetaMath-Mistral-7B model using the introduced MetaMath-GSM8K-SymPy dataset. Furthermore, a custom sampling method was developed to enhance the performance of the model. A series of experiments was performed with various samplers and combinations of the verifier model. The Greedy Best Popular and Cosine Similarity samplers were identified as the most effective. The resulting model exhibited comparable accuracy of final answers to that of the original model, but in a format that offers better interpretability and the ability to perform automatic analysis of the results.

I hope that this work will prove beneficial in future research in the field of using large language models to solve mathematical word problems through code generation, by introducing novel approach based on symbolic expressions. Furthermore, it is my hope that this work will help to bridge the divide between the mathematical reasoning and code generation fields, as both employ analogous methodologies and share numerous similarities.

[1] https://huggingface.co/datasets/tfshaman/metamath_sympy_v1
[2] https://huggingface.co/datasets/tfshaman/wrong_metamath_sympy_v1
[3] https://huggingface.co/datasets/tfshaman/error_metamath_sympy_v1
[4] https://huggingface.co/datasets/tfshaman/metamath_gsm8k_sympy_v1

Bibliography

[Ahn, Janice et al., 2024] Ahn, Janice, Verma, Rishu, Lou, Renze, Liu, Di, Zhang, Rui, and Yin, Wenpeng (2024). Large language models for mathematical reasoning: Progresses and challenges. In Falk, Neele, Papi, Sara, and Zhang, Mike, editors, *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop*, pages 225–237, St. Julian's, Malta. Association for Computational Linguistics.

[Beltagy et al., 2020] Beltagy, I., Peters, M. E., and Cohan, A. (2020). Longformer: The long-document transformer. *arXiv:2004.05150*.

[Chen et al., 2022] Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.-G., and Chen, W. (2022). Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

[Chen et al., 2021] Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. (2021). Evaluating large language models trained on code.

[Chern et al., 2023] Chern, E., Zou, H., Li, X., Hu, J., Feng, K., Li, J., and Liu, P. (2023). Generative ai for math: Abel. `https://github.com/GAIR-NLP/abel`.

[Cobbe et al., 2021] Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. (2021). Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

[Hendrycks et al., 2021] Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., and Steinhardt, J. (2021). Measuring mathematical problem solving with the math dataset. *NeurIPS*.

[Hu et al., 2022] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. (2022). LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

[Jiang et al., 2023] Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., Casas, D. d. l., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., et al. (2023). Mistral 7b. *arXiv preprint arXiv:2310.06825*.

[Lewkowycz et al., 2022] Lewkowycz, A., Andreassen, A., Dohan, D., Dyer, E., Michalewski, H., Ramasesh, V., Slone, A., Anil, C., Schlag, I., Gutman-Solo, T., et al. (2022). Solving quantitative reasoning problems with language models. *Advances in Neural Information Processing Systems*, 35:3843–3857.

[Li et al., 2023] Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al. (2023). Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

[Liu et al., 2023] Liu, B., Bubeck, S., Eldan, R., Kulkarni, J., Li, Y., Nguyen, A., Ward, R., and Zhang, Y. (2023). Tinygsm: achieving> 80% on gsm8k with small language models. *arXiv preprint arXiv:2312.09241*.

[Prakash et al., 2024] Prakash, N., Shaham, T. R., Haklay, T., Belinkov, Y., and Bau, D. (2024). Fine-tuning enhances existing mechanisms: A case study on entity tracking. *arXiv preprint arXiv:2402.14811*.

[Roziere et al., 2023] Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al. (2023). Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

[Shi et al., 2023] Shi, F., Chen, X., Misra, K., Scales, N., Dohan, D., Chi, E., Schärli, N., and Zhou, D. (2023). Large language models can be easily distracted by irrelevant context. *arXiv preprint arXiv:2302.00093*.

[Touvron et al., 2023] Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. (2023). Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

[Wang et al., 2024] Wang, K., Ren, H., Zhou, A., Lu, Z., Luo, S., Shi, W., Zhang, R., Song, L., Zhan, M., and Li, H. (2024). Mathcoder: Seamless code integration in LLMs for enhanced mathematical reasoning. In *The Twelfth International Conference on Learning Representations*.

[Wang et al., 2023] Wang, P., Li, L., Shao, Z., Xu, R., Dai, D., Li, Y., Chen, D., Wu, Y., and Sui, Z. (2023). Math-shepherd: Verify and reinforce llms step-by-step without human annotations. *CoRR, abs/2312.08935*.

[Yu et al., 2023] Yu, L., Jiang, W., Shi, H., Yu, J., Liu, Z., Zhang, Y., Kwok, J. T., Li, Z., Weller, A., and Liu, W. (2023). Metamath: Bootstrap your own mathematical questions for large language models. *arXiv preprint arXiv:2309.12284*.

[Yuan et al., 2023] Yuan, Z., Yuan, H., Tan, C., Wang, W., and Huang, S. (2023). How well do large language models perform in arithmetic tasks?

[Zhang et al., 2024] Zhang, H., Da, J., Lee, D., Robinson, V., Wu, C., Song, W., Zhao, T., Raja, P., Slack, D., Lyu, Q., et al. (2024). A careful examination of large language model performance on grade school arithmetic. *arXiv preprint arXiv:2405.00332*.

[Zhou et al., 2024] Zhou, A., Wang, K., Lu, Z., Shi, W., Luo, S., Qin, Z., Lu, S., Jia, A., Song, L., Zhan, M., and Li, H. (2024). Solving challenging math word problems using GPT-4 code interpreter with code-based self-verification. In *The Twelfth International Conference on Learning Representations*.

Appendices

Appendix A
Dataset Tables

The tables present openly available datasets on mathematical reasoning and code that could be located at the time of writing this work. The tables include URLs linking to the datasets' locations. However, it should be noted that some of them may no longer be accessible. Those links that were not available at the time of writing this work have been crossed out. The tables also contain information about the license and type of the dataset. It is important to note that not all of those datasets are in English.

A.1 Math Datasets

Name	Type	License	Where to find
MAWPS	MWPs	Unknown	~~github~~
SVAMP	MWPs	MIT	github
Math23K	MWPs	Unknown	ai.tencent.com
ALG514	AWPs	Unknown	groups.csail.mit.edu
MATHQA	MC	apache-2.0	huggingface or github.io
MetaMathQA	MWPs	MIT	huggingface
Ape-210k	MWPs	Unknown	~~github~~ or github
HWMP	MWPs	MIT	github
MATH	MWPs	MIT	github
GSM8K	MWPs	MIT	github or huggingface
TALSCQ-EN	MC	MIT	github or huggingface
asdiv-a	MWPs	Unknown	github
draw	AWPs	Unknown	~~aka.ms~~ or github
DeepMind math	GEN	apache-2.0	github
MathCodeInstruct	MWPs	apache-2.0	huggingface
GSM-IC	MWPs	Unknown	github

MWP dataset types are following:

- MWPs - Math Word Problems
- MWP-C - Math Word Problems with Code solutions
- AWPs - Algebra Word Problems
- MC - Multiple-choice
- GEN - Generates questions

A.2 Code Datasets

Name	Type	License	Where to find
Human Eval	PTC	MIT	github
Humaneval-X	MTC	apache-2.0	huggingface
MBPP	PTC	Apache-2.0 or cc-by-4.0	github or huggingface
APPS	PTC	MIT	huggingface or github
CoNaLa	PTC	MIT	github.io or huggingface
CONCODE	JTC	MIT	github
code_contests	MTC	Apache-2.0 or cc-by-4.0	github or huggingface
CodeNet	MTC	Apache-2.0	github
description2code	MTC	Unknown	github
Django	PCEP	MIT	github
CodeXGLUE	MTC	MIT	github
XLCoST	MTC	MIT	github
PLBART	MTC	MIT	github
DS-1000	PTC	CC-BY-SA-4.0 license	github.io or github
JuICe	PTC	Unknown	github
DataScienceProblems	PTC	MIT	github
Lyra	O	GPL-3.0	github
PyTorrent	PTC	Unknown	github
PythonProgrammingPuzzles	PTC	MIT	github
150k Python Dataset	PTC+	Unknown	sri.inf.ethz.ch
Natural Language to Python Code	PTC	Unknown	kaggle
Python Code Data	PTC	CC0: Public Domain	kaggle
Meta Kaggle Code	O	Apache-2.0	kaggle
Code Parrot	MTC	Other	huggingface or huggingface
Code Clippy	MTC	Unknown or MIT	the-eye.eu or huggingface
Code Pile	MTC	MIT	github
The Stack	MTC	Other	huggingface
BigCode Dataset	MTC	Apache-2.0 license	github

Code dataset types are following:

- PTC - Python Text description Codes
- JTC – Java Text description Codes
- MTC – Multilingual Text description Codes
- PCEP – Python Code English Pseudocode
- O – other

Appendix B
ChatGPT prompt

An example of prompt given to ChatGPT-3.5 to reformat dataset. The prompt is slightly modified by changing spacing and new lines to appear readable in this work.

```
You are an expert reformatting MWP dataset.
Given MWP and it's solution write a solution using only modules sympy and math.
You will be given few examples of how to solve your task.
Afterwards answer with a single word "Understood" to receive your next data.

Example 1:
Question: "John builds a toy bridge to support various weights.
It needs to support 6 cans of soda that have 12 ounces of soda.
The cans weigh 2 ounces empty.  He then also adds 2 more empty cans.
How much weight must the bridge hold up?"
Answer: "The weight of soda was 6*12=<<6*12=72>>72 ounces
It had to support the weight of 6+2=<<6+2=8>>8 empty cans
The weight of the empty cans is 8*2=<<8*2=16>>16 ounces
So it must support 72+16=<<72+16=88>>88 ounces
#### 88"

Solution:
First create symbols for each defined variable, and variables storing their value.
Then using step by step reasoning solve the problem.
After each text reasoning field include a line of code that does it.
Separate text and code fileds using tags [|Text|], [|Sympy|], [|EndOfBlock|].

Example of the solution:
[|Text|]
Let's define the variables:

w_s: weight of soda in one can (in ounces)
n: number of cans of soda
w_ec: weight of an empty can (in ounces)
n_ac: number of additional empty cans

The total weight the bridge must hold up is the sum of
the weight of soda and the weight of empty cans.
[|EndOfBlock|]
[|Sympy|]
import sympy as sp

# Given values
w_s_value = 12  # weight of soda in one can (in ounces)
n_value = 6 # number of cans of soda
```

```
w_ec_value = 2  # weight of an empty can (in ounces)
n_ac_value = 2 # number of additional empty cans

# Define symbols
w_s, n, w_ec, n_ac = sp.symbols('w_s n w_ec n_ac')
[|EndOfBlock|]
[|Text|]
The total number of empty cans is given by n + n_ac.
[|EndOfBlock|]
[|Sympy|]
# number of empty cans
n_ec = n + n_ac
[|EndOfBlock|]
[|Text|]
The weight of soda is given by w_s * n.
The weight of empty cans is given by w_ec * n_ec.
[|EndOfBlock|]
[|Sympy|]
# weight of soda
w_soda = w_s * n
# weight of empty cans
w_empty_cans = w_ec * n_ec
[|EndOfBlock|]
[|Text|]
Therefore, the total weight is w_soda + w_empty_cans.
[|EndOfBlock|]
[|Sympy|]
answer = w_soda + w_empty_cans
answer_value = answer.subs({w_s: w_s_value, n: n_value, w_ec: w_ec_value, n_ac: n_ac_value})
[|EndOfBlock|]
[|Text|]
Therefore the answer is answer_value
[|EndOfBlock|]
[|Sympy|]
print(answer_value)
[|EndOfBlock|]

Example 2:
Question: Amelia has $60 to spend on her dinner at a restaurant.
The first course costs $15 and the second course $5 more.
The cost of the dessert is 25% of the price of the second course.
How much money will Amelia have left after buying all those meals?

Answer: The second course cost Amelia $15 + $5 = $<<15+5=20>>20.
The dessert cost 25/100 * $20 = $<<25/100*20=5>>5.
That leaves Amelia with $60 - $15 - $20 - $5 = $<<60-15-20-5=20>>20.
#### 20

Solution:
[|Text|]
Let's define the variables:

first_course: cost of the first course
second_course: cost of the second course
dessert_percentage: percentage of the second course cost for dessert
budget: total budget Amelia has
```

```
The second course costs $5 more than the first course,
and the dessert costs 25% of the second course.
The total cost is the sum of the first course, second course, and dessert costs.
[|EndOfBlock|]
[|Sympy|]
import sympy as sp
# Given values
budget_value = 60 # total budget Amelia has
first_course_value = 15 # cost of the first course
second_course_value = first_course_value + 5 # cost of the second course
dessert_percentage_value = 25 # percentage of the second course cost for dessert

# Define symbols
first_course, second_course, dessert_percentage, budget = sp.symbols('first_course
    second_course dessert_percentage budget')
[|EndOfBlock|]
[|Text|]
The cost of the second course is the sum of the first course cost and $5.
[|EndOfBlock|]
[|Sympy|]
# cost of the second course
second_course = first_course + 5
[|EndOfBlock|]
[|Text|]
The cost of the dessert is 25% of the second course cost.
[|EndOfBlock|]
[|Sympy|]
# cost of the dessert
dessert_cost = (dessert_percentage / 100) * second_course
[|EndOfBlock|]
[|Text|]
Therefore, the total cost is the sum of the first course, second course, and dessert costs.
[|EndOfBlock|]
[|Sympy|]
# total cost
total_cost = first_course + second_course + dessert_cost
total_cost_value = total_cost.subs({first_course: first_course_value,
    dessert_percentage: dessert_percentage_value})
[|EndOfBlock|]
[|Text|]
Amelia will have money left after buying all those meals,
which is the remaining budget after deducting the total cost.
[|EndOfBlock|]
[|Sympy|]
# remaining budget
remaining_budget = budget - total_cost
remaining_budget_value = remaining_budget.subs({budget: budget_value,
    total_cost: total_cost_value})
[|EndOfBlock|]
[|Text|]
Therefore, the answer is remaining_budget_value.
[|EndOfBlock|]
[|Sympy|]
answer_value = remaining_budget_value
print(answer_value)
```

```
[|EndOfBlock|]

Example 3:
Question:
If Rex wants to take 40 hour-long lessons before his driver's
license test and is currently having two-hour sessions twice a week,
how many more weeks does he need to continue taking lessons after 6 weeks to reach his goal?

Answer:
In one week, Rex has two sessions of two hours each,
so he has a total of 2 x 2 = 4 hours of lessons in one week. After 6 weeks,
he has taken a total of 6 x 4 = 24 hours of lessons.
If he wants to take 40 hours of lessons in total,
he still needs 40 - 24 = 16 more hours of lessons.
If he continues with two-hour sessions twice a week,
he will have 2 x 2 = 4 more hours of lessons in one week.
So, he will need 16 / 4 = 4 more weeks to reach his goal.
#### 4
The answer is: 4

Solution:
[|Text|]
Let's define the variables:

hours_per_session: duration of each lesson session in hours
sessions_per_week: number of lesson sessions per week
total_lessons_goal: total number of lessons Rex wants to take
weeks_already_completed: number of weeks Rex has already taken lessons
total_hours_taken: total number of hours Rex has already taken lessons
[|EndOfBlock|]
[|Sympy|]
import sympy as sp

#Given values
hours_per_session_value = 2 # duration of each lesson session in hours
sessions_per_week_value = 2 # number of lesson sessions per week
total_lessons_goal_value = 40 # total number of lessons Rex wants to take
weeks_already_completed_value = 6 # number of weeks Rex has already taken lessons

# Define symbols
hours_per_session, sessions_per_week, total_lessons_goal,
    weeks_already_completed, total_hours_taken = sp.symbols(
    'hours_per_session sessions_per_week
    total_lessons_goal weeks_already_completed total_hours_taken')
[|EndOfBlock|]
[|Text|]
In one week, Rex has sessions_per_week sessions of hours_per_session each,
so he has a total of sessions_per_week * hours_per_session hours of lessons in one week.
[|EndOfBlock|]
[|Sympy|]
# total hours per week
total_hours_per_week = sessions_per_week * hours_per_session
[|EndOfBlock|]
[|Text|]
After weeks_already_completed weeks,
he has taken a total of weeks_already_completed * total_hours_per_week hours of lessons.
```

```
[|EndOfBlock|]
[|Sympy|]
# total hours taken so far
total_hours_taken_so_far = weeks_already_completed * total_hours_per_week
[|EndOfBlock|]
[|Text|]
If he wants to take total_lessons_goal lessons in total,
he still needs total_lessons_goal - total_hours_taken_so_far more hours of lessons.
[|EndOfBlock|]
[|Sympy|]
# remaining hours needed
remaining_hours_needed = total_lessons_goal - total_hours_taken_so_far
[|EndOfBlock|]
[|Text|]
If he continues with sessions_per_week sessions of hours_per_session each per week,
he will have total_hours_per_week more hours of lessons in one week.
[|EndOfBlock|]
[|Sympy|]
# additional hours per week
additional_hours_per_week = total_hours_per_week
[|EndOfBlock|]
[|Text|]
So, he will need remaining_hours_needed / additional_hours_per_week more weeks
to reach his goal.
[|EndOfBlock|]
[|Sympy|]
# additional weeks needed
additional_weeks_needed = sp.ceiling(remaining_hours_needed / additional_hours_per_week)
[|EndOfBlock|]
[|Text|]
Therefore, the answer is additional_weeks_needed.
[|EndOfBlock|]
[|Sympy|]
answer_value = additional_weeks_needed.subs({
hours_per_session: hours_per_session_value,
sessions_per_week: sessions_per_week_value,
total_lessons_goal: total_lessons_goal_value,
weeks_already_completed: weeks_already_completed_value
})
[|EndOfBlock|]
[|Sympy|]
print(answer_value)
[|EndOfBlock|]

Example 4:
Question:
Simplify $(2-3z) - (3+4z)$.

Answer:
We distribute the negative sign to both terms inside the parentheses:
$(2-3z) - (3+4z) = 2 - 3z - 3 - 4z$
Combining like terms, we have:
$2 - 3z - 3 - 4z = (2-3) + (-3z-4z) = \\boxed{-1 - 7z}$The answer is: -1-7z

Solution:
[|Text|]
```

```
Let's define the variable:

z: a variable

The expression to simplify is $(2-3z) - (3+4z)$.
[|EndOfBlock|]
[|Sympy|]
import sympy as sp
# Define symbol
z = sp.symbols('z')
[|EndOfBlock|]
[|Text|]
Distribute the negative sign to both terms inside the parentheses.
[|EndOfBlock|]
[|Sympy|]
# Distribute the negative sign
simplified_expression = sp.expand((2 - 3z) - (3 + 4z))
[|EndOfBlock|]
[|Text|]
Combine like terms.
[|EndOfBlock|]
[|Sympy|]
# Combine like terms
simplified_expression = sp.simplify(simplified_expression)
[|EndOfBlock|]
[|Text|]
Therefore, the simplified expression is simplified_expression.
[|EndOfBlock|]
[|Sympy|]
# Result
answer = simplified_expression
[|EndOfBlock|]
[|Sympy|]
print(answer)
[|EndOfBlock|]

Example 5:
Question: Determine the distance between the points (0,15) and (8,0).
Answer: We can use the distance formula, which states that the distance
between two points $(x_1, y_1)$ and $(x_2, y_2)$ is given
by $\\sqrt{(x_2-x_1)^2 + (y_2-y_1)^2}$. In this case,
the distance between (0,15) and (8,0) is
$\\sqrt{(8-0)^2 + (0-15)^2} = \\sqrt{64+225} = \\sqrt{289} = \\boxed{17}$.
The answer is: 17

Solution:
[|Text|]
Let's define the variables:

x1, y1: coordinates of the first point (0, 15)
x2, y2: coordinates of the second point (8, 0)

The distance between two points $(x_1, y_1)$ and $(x_2, y_2)$
is given by the distance formula: $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.
[|EndOfBlock|]
[|Sympy|]
```

```
import sympy as sp
# Define symbols
x1, y1, x2, y2 = sp.symbols('x1 y1 x2 y2')
[|EndOfBlock|]
[|Text|]
Apply the distance formula to find the distance between the points (0,15) and (8,0).
[|EndOfBlock|]
[|Sympy|]
# Distance formula
distance_formula = sp.sqrt((x2 - x1)**2 + (y2 - y1)**2)
# Substitute the coordinates of the points
distance_expression = distance_formula.subs({x1: 0, y1: 15, x2: 8, y2: 0})
# Simplify the expression
distance_value = sp.simplify(distance_expression)
[|EndOfBlock|]
[|Text|]
Therefore, the distance is given by distance_value.
[|EndOfBlock|]
[|Sympy|]
# Result
answer = distance_value
[|EndOfBlock|]
[|Sympy|]
print(answer)
[|EndOfBlock|]

Example 6:
Question:
Ralph has $54.00 worth of products in his cart.
At the register, he asks if he could have a 20% discount on an item with a small issue.
This item is $x to start.  They agree.
Ralph also has a 10% coupon on his purchase,
which he uses after the 20% discount on the item with the small issue.
All of his items will cost 45.

Answer:
To solve this problem, we need to determine the value of x,
which represents the original price of the item with a small issue.
Let's break down the information given:
Total cost of the items in Ralph's cart: $54.00
Discount on the item with a small issue: 20%
Coupon discount on the total purchase: 10%
Final cost of all items after discounts: $45.00
We can set up the equation as follows:
Total cost of the items
- Discount on the item with a small issue
- Coupon discount on the total purchase = Final cost of all items
$54.00 - (20% * x) - (10% * ($54.00 - (20% * x))) = $45.00
Let's simplify and solve for
x: $54.00 - (0.20x) - (0.10 * ($54.00 - (0.20x))) = $45.00
$54.00 - 0.20x - (0.10 * ($54.00 - 0.20x)) = $45.00
$54.00 - 0.20x - (0.10 * $54.00 - 0.10 * 0.20x) = $45.00
$54.00 - 0.20x - ($5.40 - 0.02x) = $45.00
$54.00 - 0.20x - $5.40 + 0.02x = $45.00
$48.60 - 0.18x = $45.00
To isolate x, we subtract $48.60 from both sides of the equation:
```

```
$48.60 - $48.60 - 0.18x = $45.00 - $48.60 -0.18x = -$3.60
To solve for x, we divide both sides of the equation
by -0.18: x = -$3.60 / -0.18 x = $20.00
The value of x is $20.00.
#### 20
The answer is: 20

Solution:
[|Text|]
To solve this problem, we need to determine the value of x,
which represents the original price of the item with a small issue.
Let's break down the information given:
Total cost of the items in Ralph's cart: $54.00
Discount on the item with a small issue: 20%
Coupon discount on the total purchase: 10%
Final cost of all items after discounts: $45.00
[|EndOfBlock|]

[|Sympy|]
# Given values
total_cost = 54.00 # total cost of the items in Ralph's cart
discount_percentage = 20 # discount on the item with a small issue
coupon_percentage = 10 # coupon discount on the total purchase
final_cost_after_discounts = 45.00 # final cost of all items after discounts
# Define symbol
x = sp.symbols('x')
[|EndOfBlock|]
[|Text|]
We can set up the equation as follows:
Total cost of the items
- Discount on the item with a small issue
- Coupon discount on the total purchase = Final cost of all items
$54.00 - (20% * x) - (10% * ($54.00 - (20% * x))) = $45.00
[|EndOfBlock|]
[|Sympy|]
# Set up the equation
equation = total_cost - (discount_percentage / 100)
    * x - (coupon_percentage / 100) * (total_cost
    - (discount_percentage / 100) * x) - final_cost_after_discounts
[|EndOfBlock|]
[|Text|]
Let's simplify the equation.
[|EndOfBlock|]
[|Sympy|]
# Simplify the equation
simplified_equation = sp.simplify(equation)
[|EndOfBlock|]
[|Text|]
Now, let's solve for x.
[|EndOfBlock|]
[|Sympy|]
# Solve for x
solution = sp.solve(simplified_equation, x)
[|EndOfBlock|]
[|Text|]
Extract the numerical solution for x.
```

```
[|EndOfBlock|]
[|Sympy|]
# Extract the numerical solution
x_value = solution[0]
print(x_value)
[|EndOfBlock|]


Example 7:
Question:
Natalia sold clips to 48 of her friends in April,
and then she sold half as many clips in May.
How many clips did Natalia sell altogether in April and May?
Answer:
Natalia sold 48/2 = <<48/2=24>>24 clips in May.
Natalia sold 48+24 = <<48+24=72>>72 clips altogether in April and May.
#### 72

Solution:
[|Text|]
Let's define the variables:

w_s: weight of soda in one can (in ounces)
n: number of cans of soda
w_ec: weight of an empty can (in ounces)
n_ac: number of additional empty cans

The total weight the bridge must hold up is the sum of the
weight of soda and the weight of empty cans.
[|EndOfBlock|]
[|Sympy|]
import sympy as sp
import math

# Given values
w_s_value = 12 # weight of soda in one can (in ounces)
n_value = 6 # number of cans of soda
w_ec_value = 2 # weight of an empty can (in ounces)
n_ac_value = 2 # number of additional empty cans

# Define symbols
w_s, n, w_ec, n_ac = sp.symbols('w_s n w_ec n_ac')
w_s, n, w_ec, n_ac = sp.symbols('w_s n w_ec n_ac')
[|EndOfBlock|]
[|Text|]
The total number of empty cans is given by n + n_ac.
[|EndOfBlock|]
[|Sympy|]
# number of empty cans
n_ec = n + n_ac
[|EndOfBlock|]
[|Text|]
The weight of soda is given by w_s * n.
The weight of empty cans is given by w_ec * n_ec.
[|EndOfBlock|]
[|Sympy|]
```

```
# weight of soda
w_soda = w_s * n
# weight of empty cans
w_empty_cans = w_ec * n_ec
[|EndOfBlock|]
[|Text|]
Therefore, the total weight is w_soda + w_empty_cans.
[|EndOfBlock|]
[|Sympy|]
# total weight
total_weight = w_soda + w_empty_cans
total_weight_value = total_weight.subs({
    w_s: w_s_value,
    n: n_value,
    w_ec: w_ec_value,
    n_ac: n_ac_value})
[|EndOfBlock|]
[|Text|]
Therefore the answer is total_weight_value
[|EndOfBlock|]
[|Sympy|]
answer_value = total_weight_value
print(answer_value)
[|EndOfBlock|]

Now you should be ready to work.
If you understood confirm your task by answering with a single word "Understood"
```

Appendix C
Acronyms

MWP Math Word Problem

LLM Large Language Model

SOTA State-of-the-art

ME Mathematical Error

CE Code Error

Sampler Sampling filter