Next-Gen Programming Maths Prompt Translator

Boris Petreski
Technical University of Munich
Munich, Germany
boris.petreski@mytum.de

Ludvig Sanell
Technical University of Munich
Munich, Germany
ludsan22@student.hh.se

Casper du Jardin Kejser
Technical University of Munich
Munich, Germany
202305847@post.au.dk

***Abstract*—The intregation of Artificial Intelligence (AI) into software development is reshaping programming practices, particularly in the area of automated code generation. In this work, we introduce the Maths Prompt Translator, an AI-powered system built upon the CodeT5 model, designed to convert natural language arithmetic queries into correct Python code. A custom dataset was developed to support both training and evaluation, ensuring a diverse representation of arithmetic tasks. The system demonstrated strong performance across standard arithmetic operations while highlighting opportunities for further refinement in handling more complex queries. These findings underscore the potential of AI-driven approaches to bridge the gapp between theoretical concepts and practical software engineering applications.**

I. Introduction

The integration of Artificial Intelligence (AI) into software development, particularly through large language models (LLMs), has ushered in a new era of automated code generation, prompting extensive research into systems that can translate natural language descriptions to solve different problems. Despite the growing focus on AI in academic curricula, many still lack the practical exposure required to use these technologies effectively in real-world applications.

In response to this gap, our Next-Gen Programming: Maths prompt translator project was built to provide a structured, hands-on laboratory experience. The Maths Prompt Translator is a system built upon the CodeT5 model. Although CodeT5 was not originally designed to handle datasets comprising mathematical queries, our system has been engineered to transform natural language maths prompts into functional Python code through fine-tuning and adaptation.

The primary objectives of this work are the following. First, the project aims to demonstrate the possibility and effectiveness of integrating AI components into the software development process by automatically generating code from textual descriptions.

Secondly, it seeks to bridge the gap between theoretical AI concepts and their practical application, thereby equipping us with the necessary skills to program and use the landscape of AI-driven programming.

This paper is organised as follows: the remainder of the document details the methodology behind our approach, presents an in-depth analysis of the experimental results focusing on category-specific performance, and provides graphical representations to visualize the findings. Ultimately, the work presented here aims to contribute both to the understanding of AI-driven code generation and to the practical applications of automated programming systems.

II. Background

The rise of Artificial Intelligence (AI) has transformed software engineering, particularly through the development of large language models (LLMs) capable of generating code from natural language. Models such as CodeT5 exemplify how AI can automate complex programming tasks, thereby enhancing productivity and reducing errors in code development.

Despite these advancements, a significant gap persists between theoretical instruction and practical application. While students often acquire extensive theoretical knowledge of AI, opportunities for hands-on experience in applying these concepts to real-world programming challenges remain limited. Therefore, the practical course Next Gen Programming allows for a great opportunity of applying theoretical knowledge in a real-world experience.

This project within the course called Maths Prompt Translator is based on the CodeT5 model, which has been fine-tuned to quickly and efficiently translate natural language mathematical problems into accurate Python code. This approach not only aligns with the trend of adapting AI for specific challenges but also serves as a practical demonstration of integrating AI into the software development process.

The sections below detail the methodologies employed, present the outcomes, and discuss the broader implications of our findings, thereby illustrating the potential of AI-driven automated programming.

III. Goal

The primary goal of this project has been to develop an AI-powered system capable of translating natural language queries into accurate and executable code. To solve this objective, this work has been guided by the following key aims:

1) **Demonstrate possibility:** Establish that deep learning models, when properly fine-tuned, can be effectively integrated into the software development process to automatically generate functional code from textual descriptions.

2) **Bridge Theory and Practice:** Address the gap between theoretical AI instruction and real-world application, enabling us to apply AI-driven methods in practical programming scenarios.

3) **Set result benchmarks:** Focus on handling mathematical queries and evaluating benchmark results such as metrics for accuracy, precision, recall, F1 score, and error analysis.

Through these objectives, this project aims not only to advance the understanding of AI-powered code generation but also to provide a practical solution for streamlining the software development process.

IV. Development process

A. Choosing the model

Selecting an appropriate base model was a critical first step in designing and fine-tuning a system for translating natural language arithmetic queries into Python code. To solve this, we initially compiled a list of models that we would test, which are summarised in Table I.

Model
GPT-2 Medium
GPT-2 Large
GPT-2 XL
CodeT5 Large
CodeT5 Base
CodeT5 Small
CodeGen Medium
CodeGen Small

TABLE I
Initially tested models

These models were first evaluated using a small, task-specific dataset that contained mathematical problems alongside their corresponding Python code outputs. Our evaluation focused on selecting a model that demonstrated robust natural language understanding while not already possessing the capability to translate text into code without further fine-tuning. In other words, the ideal candidate should require adaptation to perform the desired task, thereby providing a meaningful challenge and opportunity for improvement.

Our initial testing revealed that the CodeGen models, both small and medium, were already too capable of generating code outputs similar to our desired outcome, and thus did not meet our requirement for further development. Consequently, the decision narrowed to a choice between the GPT-2 models and the CodeT5 variants. Although both models showed competent language understanding, we initially selected the CodeT5 Large Model. Its performance in generating code was slightly less proficient compared to its counterparts, indicating a greater scope for enhancement through fine-tuning.

Additionally, the CodeT5 Large was directly available on Hugging Face, which made its integration and experimentation easier [1].

CodeT5 is a pre-trained encoder-decoder model that builds upon the T5 architecture, an approach that has proven highly effective for both natural language understanding and generation tasks [2] [3]. These characteristics makes CodeT5 an optimal choice for this project, offering both a solid baseline understanding and ample room for performance improvement through targeted fine-tuning.

B. Deciding upon dataset

A robust and relevant dataset is essential for training an AI system that translates natural language queries into Python code. Given the lack of pre-existing datasets meeting our specific objectives, we opted for the challenge of developing a custom dataset. Initially, a small set of approximately 50 examples was manually made to capture the desired linguistic style and structure of arithmetic problems. This small set of data was insufficient to train a deep-learning model effectively. Therefore, we expanded it using an AI-assisted approach.

Specifically, the manually created examples were used as prompts for ChatGPT [4] along with other specifications to generate additional examples. To ensure a diverse range of problem descriptions and to incorporate more challenging queries, multiple distinct datasets were generated by varying the prompts and specifications. These individual datasets were then merged, resulting in a final dataset comprising 11,484 unique questions paired with their corresponding Python code outputs.

In accordance with practices in machine learning, the complete dataset was split using a 5/1 rule allocating $\frac{5}{6} = 83.333\%$ of the examples to the training set and reserving $\frac{1}{6} = 16.667\%$ for testing. This division is closely

aligned with the literature to balance model training and unbiased performance evaluation.

Table II provides a summary of the final dataset composition.

Dataset Partition	Number of Examples
Training Set = (83.333%)	9,570
Testing Set = (16.667%)	1,914

TABLE II
SUMMARY OF THE FINAL DATASET COMPOSITION.

Additionally, in order to ensure fairness and a robust evaluation of the model, the dataset was partitioned such that the distribution of arithmetic categories in the training set mirrored that of the testing set. Specifically, the proportions of questions covering pure addition, subtraction, division, multiplication, and miscellaneous operations were kept consistent across both splits. Our custom-built dataset, specifically designed to contain different easily mathematical queries and their corresponding Python code, provides a solid foundation for training and evaluating the Maths Prompt Translator efficiently.

C. Training the AI-model

When the dataset was created, the model training could begin. This step started somewhat early in the development cycle, even before the dataset was finalised. This was done in order to get a grasp of the project and also to be able to continue with the code for testing. When the dataset was finalized, the training could also be finalized. In order to train the model, the code for this had to be created. For this project, Python was chosen as the language to use for training, since it is a common language to use for AI applications, and it is also known for its ease of use when it comes to mathematical applications [5].

A ".ipynb" file was also created since it allows a Python notebook to be run section by section instead of the whole program all at once [6]. This allowed for easier debugging since it always showed which section failed. During testing, it was discovered that a switch from CodeT5 Large to CodeT5 Base would also be optimal. This was because, with the dataset created, CodeT5 Large would be under-saturated. In other words, it means that it was not provided with enough data to be optimal. The base model proved both sufficient and optimal for this use case.

D. Training Process

The model was fine-tuned on the training set over the course of five epochs. During training, both the training loss and the validation loss were monitored to assess convergence and generalisation performance. Table III summarises both the training- and validation loss values recorded at each epoch.

Epoch	Training Loss	Validation Loss
1	0.0014	0.000758
2	0.0001	0.000184
3	0.0016	0.000124
4	0.0012	0.000090
5	0.0003	0.000076

TABLE III
TRAINING AND VALIDATION LOSS OVER 5 EPOCHS.

Figure 1 illustrates a graphical representation of the results of the training and validation loss over the epochs. While the training loss exhibits some fluctuations between epochs, potentially due to batch-level variance or possibly due to the smaller size of the dataset, the validation loss consistently decreases, reaching a minimum of 0.000076 by epoch 5. This steady decline in the validation loss shows improved generalization performance as the model is exposed to more examples during training.

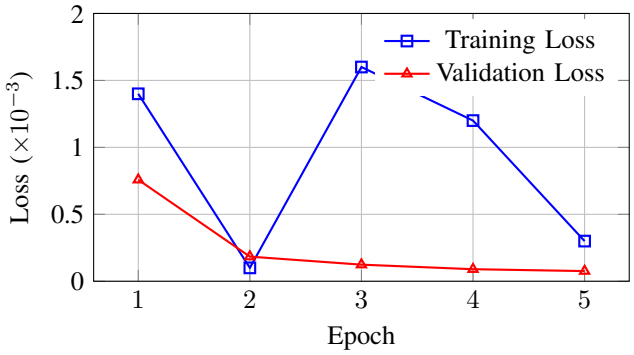

Fig. 1. Evolution of training and validation loss over epochs.

The training results revealed several key points:

- **Rapid Initial Convergence:** The sharp drop in training loss from epoch 1 to epoch 2 (from 0.0014 to 0.0001) indicates that the model quickly captures the fundamental patterns in the data.

- **Fluctuations in Training Loss:** The increase in training loss at epoch 3 to 0.0016, followed by a subsequent decrease by epoch 5, suggests the presence of batch-level variance. Despite these fluctuations, the overall trend is a steady reduction in loss after the third epoch.

- **Consistent Decline in Validation Loss:** The validation loss steadily decreases over the epochs, culminating in a very low final value. Interestingly, the validation loss remains consistently on a downward trend, even during epochs where the training loss fluctuates. This suggests that the model's generalization capability was not adversely affected by temporary overfitting to specific training batches.

In summary, the training process was characterized by rapid initial learning, minor fluctuations in training loss, and

a consistent decrease in validation loss. These observations confirm that the model has converged effectively and is well-tuned to generalise to new queries.

E. Fine-Tuning the model:

The fine-tuning of the model consisted of using more datasets, like APPS and MBPP and different wordings of the problems, improving the performance

V. Testing and Results

To evaluate the performance of the Maths Prompt Translator, we selected $\frac{1}{6}$ of the entire dataset at random for testing, though with the same distribution of the different problem categories. This stratified sampling strategy ensured that the test set maintained the same category distribution as the training set, thereby accurately reflecting the model's generalisation capabilities. In total, 1,914 questions were used in the testing phase.

After training, the testing of the model showed promising results by successfully translating the vast majority of arithmetic queries into Python code. The overall performance on the 1,914 test cases was as follows:

- **Accuracy:** 99.27% (1,900 out of 1,914 correct)
- **Precision:** 1.0000
- **Recall:** 0.9927
- **F1 Score:** 0.9963

A. Error Analysis

A detailed review of the outputs revealed that only 14 questions were flagged as incorrect. Many of these discrepancies were minor and pertained to superficial issues rather than fundamental arithmetic mistakes. The errors can be categorised as follows:

- **Operand Misplacement:** For instance, in one case the expected output was

  ```
  def solve(): return 72 / 71
  ```

  however the model produced

  ```
  def solve(): return 71 / 72
  ```

  indicating an inversion of operands. This was an outlier as it only happened once from all the testing.

- **Formatting Variations:** In another case, subtle formatting differences were observed. For example, when processing the query *"Add the square of 9 to the difference between the product of 5 and 6 and the cube of 4"*, the expected output was

  ```
  def solve(): return (9 ** 2 + (5 * 6 - 4 ** 3))
  ```

  whereas the model produced

  ```
  def solve(): return (9 ** 2 + 5 * 6 - 4 ** 3)
  ```

  Here, the omission of extra parentheses may lead to ambiguities in expression parsing.

- **Syntax Discrepancies in Complex Expressions:** Errors also arose in queries requiring the use of library functions. For instance, for the query *"What is the square root of 225?"*, the expected output was

  ```
  import math
  def solve(): return math.sqrt(225)
  ```

  but the model produced

  ```
  def solve(): return sqrt(225)
  ```

  Here, the necessary 'math.' prefix was omitted, resulting in a syntax discrepancy.

These examples illustrate that, while the vast majority of errors are minor and related to formatting rather than incorrect evaluation. Generally, the model does not compromise the arithmetic correctness of the code. If such superficial issues are disregarded, only a single question out of 1,914 would be considered truly incorrect, i.e. an instance of incorrect operand misplacement.

B. Category-Specific Performance

To further assess the model's performance, the test set was divided into five distinct arithmetic categories. The distribution of questions was as follows:

- **Pure Addition:** 427 questions
- **Pure Subtraction:** 440 questions
- **Pure Division:** 462 questions
- **Pure Multiplication:** 509 questions
- **Miscellaneous:** 76 questions

A detailed breakdown by arithmetic category is provided in Table IV. The model attained perfect accuracy (100%) in the Pure Addition, Pure Subtraction, and Pure Multiplication categories. In the Pure Division category, only one question was misclassified, yielding an accuracy of 99.78%. In contrast, the Miscellaneous category—which includes more complex queries involving operations such as square roots and exponents—achieved an accuracy of 82.89%

Category	Total Questions	Correct Answers	Accuracy (%)
Pure Addition	427	427/427	100.00
Pure Subtraction	440	440/440	100.00
Pure Division	462	461/462	99.78
Pure Multiplication	509	509/509	100.00
Miscellaneous	76	63/76	82.89
Total	**1,914**	**1,901/1,914**	**99.27**

TABLE IV
Category-specific performance of the Maths Prompt Translator.

The slightly lower accuracy in the *Miscellaneous* category can be attributed to the increased syntactic complexity of those

queries—such as the need to correctly implement square root operations and manage mixed arithmetic expressions.

Figure 2 illustrates the accuracy across the five categories using a bar chart.

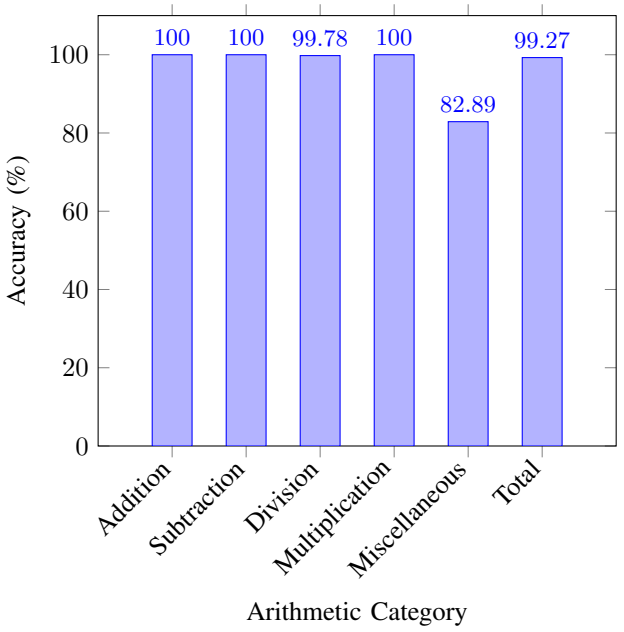

Fig. 2. Accuracy per arithmetic category.

C. Discussion of Limitations

While the model performs exceptionally well in standard arithmetic categories, the reduced accuracy in the Miscellaneous category highlights challenges in generalising to operations with higher syntactic complexity and more complex formatting varations. These limitations suggest that further fine-tuning or additional dataset diversity might be required to fully address the nuances of such queries.

In summary, the testing results demonstrate the robustness of the Maths Prompt Translator, with overall high accuracy and only minor discrepancies in the few errors observed. This analysis provides valuable insights into the model's performance and highlights potential areas for future improvement.

D. Hardware Specifications

All of the training and testing were conducted on a system with the following hardware configuration:

- **CPU:** Intel i7-13700k
- **GPU:** NVIDIA RTX 4080
- **RAM:** 32 GB

These specifications provided the necessary computational resources for efficient training and testing of the model, ensuring that the training and testing are both reproducible and scalable.

E. Summary

These results indicate that while the Maths Prompt Translator excels in standard arithmetic tasks, challenges still remain for queries with higher syntactic complexity like the ones in the miscellaneous category. Overall, the high accuracy and low error rate underscore the model's effectiveness and provide a strong foundation for future refinements.

VI. Integration with APPS and MBPP Datasets

Beyond the custom arithmetic dataset, we explored integrating two widely recognized code-generation benchmarks, APPS and MBPP, into the broader pipeline.

A. Complexity of the APPS Dataset

The APPS dataset [7] covers a broad spectrum of coding challenges, from simple tasks to algorithmically demanding problems requiring data structures like trees or graphs. Multiple valid solutions often exist for the same prompt (e.g., different sorting algorithms), making a single "ground-truth" solution less definitive. Evaluating correctness thus extends beyond syntactic validation and arithmetic accuracy, demanding checks against functionality, edge cases, and computational efficiency.

B. Complexity of the MBPP Dataset

The MBPP dataset [8] typically consists of smaller-scale Python problems that still demand basic logical reasoning, data manipulation, and sometimes domain-specific knowledge. Although these problems are more constrained than those in APPS, they too can be solved in various ways (e.g., using recursion vs. iterative approaches). Consequently, evaluation requires a flexible metric that can recognize correct but stylistically distinct solutions.

C. Challenges in Unified Testing

Applying the exact same testing methodology used for arithmetic prompts is impractical for APPS and MBPP. Their problems frequently:

- Involve multiple valid solution strategies.
- Require external libraries or particular algorithmic patterns.
- Demand performance considerations or advanced error-handling.

Automated evaluation therefore needs to account for functional correctness, potential edge cases, and multiple acceptable solutions. Simple string comparisons or single reference solutions may incorrectly label valid code as erroneous, thereby necessitating more robust and context-aware test harnesses.

D. Implications for Future Work

Integrating APPS and MBPP offers an avenue to increase the model's generalization capabilities. By exposing the model to a broader range of tasks:

- The system can learn more advanced programming concepts.

- Evaluation methodologies can be refined to handle diverse problem-solving strategies.
- Insights gained can inform improvements in dataset design, fine-tuning techniques, and real-world applicability.

Despite the complexity of evaluating advanced datasets, these explorations underscore the potential for a unified code-generation model that can handle both basic and complex queries with varying solution paths.

E. Trained model results

Interesting results occurred after using these datasets while fine-tuning the model. Problems like "sum of integers in range n" and "fibonacci" series are possible, although with limited possibilities and unexpected combinations. Using further datasets with more problems and solutions could further refine the model and expand its logical capabilities.

VII. Key lessons

- **Team Collaboration and Process:**
  - The project taught the team to work together in a professional and efficient manner, despite initial challenges.
  - It provided significant insight into how large language models and AI operate.
  - The experience of building a dataset from the ground up—and subsequently leveraging it for model training—demonstrated the end-to-end process of solving domain-specific tasks.
- **Effectiveness of Fine-Tuning:**
  - Fine-tuning the CodeT5 Base model on a custom dataset of arithmetic queries resulted in exceptionally high performance, with an overall testing accuracy of 99.27%.
  - This confirms that pre-trained large language models can be effectively repurposed for specialised code generation tasks through proper adaptation.
- **Importance of Dataset Diversity:**
  - Creating a custom dataset—expanded from a seed of manually curated examples using AI-assisted methods—was crucial for capturing a wide range of natural language expressions and arithmetic problem structures.
  - While the dataset was effective, the findings also suggest that further refinements in diversity may enhance performance for more complex query types.
- **Error Analysis and Superficial Discrepancies:**
  - Detailed error analysis revealed that most flagged errors were minor, such as operand misplacement, missing or extra parentheses, and syntax discrepancies in expressions using library functions.
  - These superficial discrepancies, although penalised under strict evaluation metrics, do not compromise the underlying arithmetic correctness, suggesting potential for more lenient evaluation or post-processing strategies.
- **Category-Specific Challenges:**
  - The model achieved near-perfect performance in Pure Addition, Subtraction, Multiplication, and Division.
  - In contrast, the Miscellaneous category (involving more complex syntactic constructs) attained a lower accuracy of 82.89%, highlighting an area for future improvement in handling complex queries.
- **Sensitivity to Hyperparameter Changes:**
  - The project demonstrated that minor adjustments in training variables—such as the number of epochs—can have a significant impact on performance metrics like convergence speed and final accuracy.
- **Generalisation and Robustness:**
  - The use of a stratified random sampling strategy ensured that the testing set mirrored the training set's distribution, thereby validating the model's generalisation capabilities.
  - Overall high performance indicates that the model is robust, though further refinements are needed to address the nuances of more complex queries.
- **Implications for Future Research:**
  - The project provides valuable insights into the practical application of AI for automated code generation.
  - Future work could focus on enhancing the model's handling of syntactic complexity, implementing more sophisticated post-processing techniques, and further diversifying the dataset to encompass an even broader range of problem types.

Overall, the project not only demonstrates the feasibility of employing fine-tuned LLMs for generating Python code from natural language queries but also underscores the importance of meticulous dataset design and thorough error analysis. The development and evaluation of the Maths Prompt Translator have yielded several important insights and lessons for the integration of AI-driven code generation into software development. Most importantly, the project has shown how enriching and powerful AI can be as a tool in software development, inspiring further exploration and innovation in the field. The lessons learned here lay a solid foundation for future advancements in AI-powered software development, bridging the gap between theoretical research and practical application.

VIII. UI Implementation Overview

The code provides a streamlined *Maths to Python Translator* interface with three main components:

1) **Custom Styling & Templates:**
   - The html files establich common layout with scripts for Prism.js (syntax highlighting) and include a simple form to collect math problems and display translated Python code.
2) **JavaScript (Main.js):**

- Submits the math problem to the `/translate` endpoint via `fetch` and manages loading states.
- Dynamically updates the UI with highlighted Python code.
- Offers example problem links to auto-fill the problem input.

3) **Server-Side Integration:**
   - The Flask back-end (`app.py`) receives the problem, uses an AI model to generate code, and returns the result as JSON.
   - Results can be stored in a database using a `MathProblem` model.
   - `ai_model.py` handles the CodeT5-based text-to-code translation.

IX. Conclusion

This study presented the development and evaluation of the Maths Prompt Translator, an AI-powered system based on the CodeT5 model that has been fine-tuned to convert natural language arithmetic queries into Python code. Our experiments demonstrate that, with proper fine-tuning on a custom-curated dataset, pre-trained large language models can achieve remarkably high performance—attaining an overall testing accuracy of 99.27% across a diverse set of arithmetic problems.

The evaluation revealed near-perfect performance in standard arithmetic categories, with 100% accuracy observed in Pure Addition, Pure Subtraction, and Pure Multiplication, and only minimal errors in Pure Division. Conversely, the Miscellaneous category, which involves more syntactic complexity such as operations with square roots and exponents, showed a lower accuracy of 82.89%. Detailed error analysis indicated that most of the discrepancies were superficial (e.g., operand misplacement, formatting variations, or minor syntax errors) rather than substantive computational inaccuracies.

These results underscore the effectiveness of our approach in leveraging AI-driven techniques for automated code generation, while also highlighting areas for future improvement—particularly in enhancing the model's handling of more complex queries. Furthermore, the project demonstrates the potential for bridging the gap between theoretical AI concepts and their practical application in software engineering, providing valuable insights for both academic research and industry practice.

Future work may focus on expanding dataset diversity, refining post-processing strategies, and further fine-tuning the model to mitigate the minor errors observed in syntactically challenging queries. Overall, the Maths Prompt Translator represents a significant step forward in the integration of AI in automated programming, offering a promising foundation for subsequent research and development in this rapidly evolving field.

References

[1] Salesforce AI Research. (2021). *CodeT5 Base model for code understanding and generation*. Hugging Face. Retrieved from https://huggingface.co/Salesforce/codet5-base

[2] Wang, Y., Wang, W., Joty, S., & Hoi, S. C. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859. Retrieved January 30th, 2025 from https://arxiv.org/pdf/2109.00859

[3] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., ... & Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of machine learning research, 21(140), 1-67. Retrieved January 30th, 2025 from https://jmlr.org/papers/volume21/20-074/20-074.pdf

[4] OpenAI. (2024). ChatGPT (3.5) [Large Language Model]. https://chatgpt.com/

[5] Bautista, J. C. (2014). Mathematics and Python Programming. Lulu. com. Retrieved January 30th, 2025, from https://books.google.de/books?id=K0B8BwAAQBAJ&printsec=frontcover&hl=da&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false

[6] Chan, H. (2023, August 2). How to open .ipynb file (Jupyter Notebook). Digital Humanities Initiative. Retrieved January 30th, 2025, from https://digitalhumanities.hkust.edu.hk/tutorials/how-to-open-ipynb-file-jupyter-notebook/

[7] D. Hendrycks, et al., ‘‘*Measuring Coding Challenge Competence With APPS*”, GitHub repository, 2021, Available at: https://github.com/hendrycks/apps.

[8] Google Research, “*MBPP: Mostly Basic Python Problems*”, GitHub repository, 2021, Available at: https://github.com/google-research/google-research/tree/master/mbpp.