Workshop in Machine Learning and Data Analysis

Eyal Grinberg, Shuni Bickel and Noga Bregman, Tel Aviv University

We participated in the challenge titled Prompt Engineering (Math). Our project aimed to answer math questions from 11th grade using LLMs by employing various prompt engineering methods. Prompt engineering is a skill set focused on creating effective prompts that guide AI models, especially large language models (LLMs), to generate specific and useful responses. It involves understanding how these AI models process language and leveraging that knowledge to formulate questions or commands that lead to the desired output, whether it involves writing, coding, image generation, or any other type of content. When using prompt engineering, one might refine the language of the prompt, adjusting its structure, or using certain keywords that the model responds to more effectively. The goal is to maximize the quality and relevance of the AI's responses for specific tasks or queries. Here, we aimed to take a math question and find the best prompt and set of actions so that the final answer would be as accurate as possible.

The scoring metric of the challenge was binary: the answers were either right or wrong, and if they were wrong, the degree of closeness to correct answer did not change the loss. The data provided consisted of training and test questions, both in Russian. While the training questions included a final solution and a clue, the test ones did not have any such information. The current lead when we started was the creator of the challenge, with 86% of success. We decided to consider a score higher than that as a success.

We initiated our approach to the challenge by translating (translation.ipynb) the questions in both the training and test datasets into English for two main reasons. The first, and most obvious, was for our own understanding. Since none of us speaks Russian, we needed to fully comprehend the questions to solve them ourselves and better understand and debug our model if necessary. Without translating the questions, we had very little insight into what was happening inside our model and whether we were on the right track. The second reason was based on our assumption that most LLMs were primarily trained on English data, given that the majority of available data is in English. We used ChatGPT 3.5 to translate all the questions. The entire pipeline can be found in our GitHub repository

After that, we started working on a simple baseline for the project. We utilized langchain.prompts.ChatPromptTemplate for crafting these prompts, simulating a dialog where the system's response evolves based on the user's input. To process the model's output, we employed langchain_core.output_parsers.StrOutputParser, enabling us to parse and utilize the information effectively.

We explored ways to fine-tune interactions with a language model using two types of prompts: "system prompt" and "my_magic_prompt". The "system prompt" served as a flexible placeholder for dynamic content, allowing us to adapt the conversation flow on-the-fly. In contrast, "my_magic_prompt" was where we introduced specific instructions or queries, guiding the model to produce responses aligned with our objectives.

The baseline (base_line.ipynb), with "my_magic_prompt" containing only the problem, received a 50% accuracy rate. It also has utilized the standard "system prompt" to handle dynamic content without specific guidance.

Next, we tried the *Chain of Thought* method. We changed “my_magic_prompt” to instruct the model to take a deep breath, decompose problems into smaller, manageable parts, and tackle them sequentially. This method slightly improved the accuracy to 56%, highlighting a modest gain through structured problem-solving.

In our third phase, we started using ideas from the *Chain Of Code* method. We experimented with a dual-language model setup - the first model generated pseudo-code to outline solutions, then the second model attempted to execute this pseudo-code, providing outputs for non-executable lines and updating delta states throughout the simulation. This approach marked a significant improvement, elevating our accuracy to 61% using GPT-4. This phase underscored the potential of collaborative language model interactions to enhance problem-solving capabilities.

After investigating using two LLM processes for both code generation and execution, we’ve come to the understanding that most math questions are self-contained, meaning all the data to solve the question should be within the context of the question. Therefore, employing an LLM to mimic code execution was redundant. Additionally, LLMs are inherently non-deterministic and prone to errors, which could introduce inaccuracies not present in directly running the code. As a conclusion, we opted to leverage the Python interpreter in our code notebook for code execution.
To ensure consistent code execution, we implemented a standardized code format: This format employs a prefix and suffix to encapsulate the generated code and designates a result variable to store the problem solution. We’ve added a function called “run_python-code” to execute the generated code, which removed the function wrapping and returned the answer saved in the “result” variable.
We’ve used gpt-4 to generate the code, and after using the execution function, we’ve run answer formatting functions to make sure the answers are in the correct format (and if the answer was missing \ an error message we’ve replaced it with a default value of 9999), and our results were 75% accuracy on the train set and 81% on the test set (code_gen_python_organized.ipynb).

To identify areas for model improvement, we conducted an error analysis on the training data. This involved examining questions where the model's predictions diverged from the expected answers and classifying them to “loss buckets” - groups of similar questions that the model had failed answering. We aimed to uncover recurring patterns in the model's errors to refine the model and enhance its performance. Some of the patterns we’ve found were problems of matching statements, for example establishing the correspondence between inequalities and their solutions (problem id 7643), and (surprisingly) problems that can be solved naively by performing many iterations (problem id 9603).

One finding we had was that some questions were answered inconsistently by the model, with the correct answer being one of the model’s answers. Moreover, we’ve found that in most cases, the correct answer corresponded to the mode (most frequent value) of the generated solutions. This finding, coupled with the lower cost of code generation compared to alternative approaches, led us to explore an ensemble-based strategy. By generating and evaluating multiple code solutions for each problem, we aimed to leverage the model's strengths and mitigate inconsistencies. This approach seeks to improve overall accuracy by selecting the mode of the generated answers as the final output. We’ve decided to generate 5 code solutions to each question and select the mode of the 5 answers, since our initial investigations demonstrated that most questions yielded at most 3 distinct solutions, suggesting the feasibility of this strategy. This approach led our results to raise to 83% accuracy on the train set and 86% on the test set (code_gen_python_with_buckets.ipynb).

We additionally considered cases where the mode (most frequent answer) fell into two categories: either it was the default error value (9999) indicating an issue during generation, or it lacked sufficient confidence, evidenced by less than 3 out of 5 generated solutions matching the mode. In these scenarios, we opted for a CoT approach to generate the answer directly. This decision stemmed from our observation that questions with an unreliable mode often involved verbal descriptions (text-based problems). We hypothesized that CoT, designed to handle natural language, would hold an inherent advantage in these situations compared to our code generation approach.
Our exploration of CoT prompting revealed that one-shot CoT delivered the best results. This approach demonstrably improved the model's performance on previously challenging questions with a complex structure, without hindering the effectiveness seen using zero-shot CoT for other problems. Conversely, three-shot CoT led to inconsistencies with answers varying too widely. Thus we’ve decided to answer all the questions mentioned above using 1-shot CoT. This approach led us to a groundbreaking 90% accuracy on the test set (contest-notebook-gen5-strong-majority.ipynb).

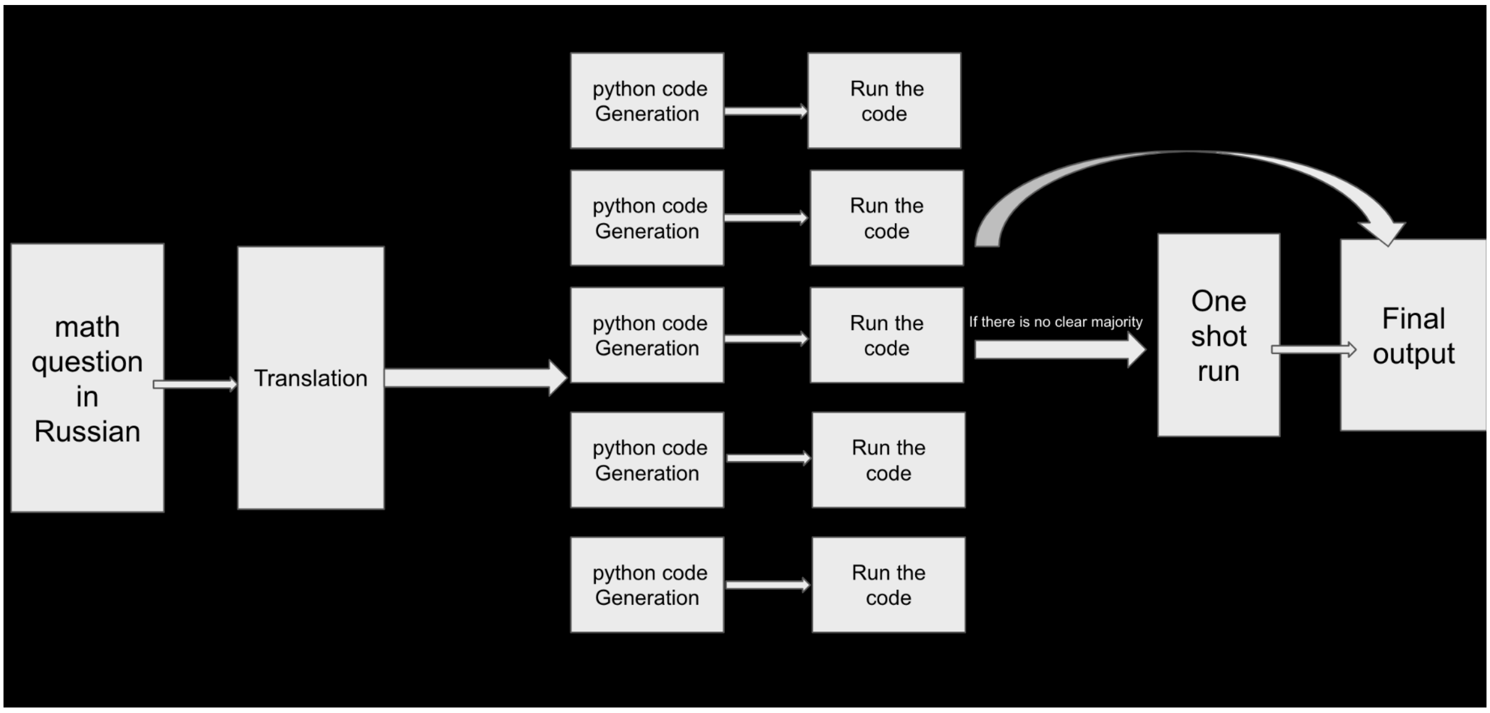
This project has shown the promise of applying code generation to solve math problems with a high success rate. It also demonstrated the potential of leveraging the statistical mode on non-deterministic models to enhance answer accuracy.
Our work underscored the significance of iterative prompt engineering in achieving optimal results. Through this process, we observed how refining the prompt's quality and fine-tuning its elements enhanced the model's performance.

Looking forward, we see significant potential in two key areas for extension. First, a classification pipeline could be developed to categorize problems. This pipeline would then tailor prompts with relevant few-shot examples specific to each category, potentially mitigating some loss-buckets observed in the general model. This approach has the potential to significantly improve performance on complex problems compared to the general solution. Second, the success achieved with math problems, which benefit from their context-free nature, suggests the approach's broader applicability. By equipping the model with the

necessary contextual information, this technique could be adapted to various domains beyond mathematics.
Prompt engineering presents a vast, untapped potential for further exploration. Refining prompt construction techniques promises significant improvements in model performance, as demonstrated in this project, and we wait to see the future discoveries that lie ahead in this field.

git: https://github.com/noga1103/workshop

Appendix
Final Architecture:


math question in Russian
Translation
python code Generation
python code Generation
python code Generation
python code Generation
python code Generation
Run the code
Run the code
Run the code
Run the code
Run the code
If there is no clear majority
One shot run
Final output