Empowering Language Models with Toolformer: An Implementation

Madhushree Nijagal
University of Wisconsin-Madison
mnijagal@wisc.edu

Sumedha Joshirao
University of Wisconsin-Madison
joshirao@wisc.edu

Tanisha Hegde
University of Wisconsin-Madison
tghegde@wisc.edu

Abstract

Language models (LMs) exhibit a remarkable capacity for problem-solving with limited instructions on a large scale. However, they encounter difficulties with basic arithmetic operations and factual retrieval. In this project, our aim is to implement the Toolformer model using a reference code that is available and extend its capabilities to incorporate other APIs, such as Wolfram API and Calendar APIs. Toolformer is a trained model designed to make decisions regarding API selection, timing, argument choices, and the effective integration of results into future token predictions. The original paper primarily focuses on four essential tools for API calls: a Calculator, Wikipedia, a QA system, and a translation system. Our objective is to enhance the model's functionality by expanding its scope to include additional API tools, followed by trying other models and observing its effect on the model. We analyse the effect of different datasets and its effect on data generation, particularly while filtering samples followed by finetuning the model.

1 Introduction

The integration of artificial intelligence and machine learning has been one of the most transformative advancements in the field of computer science and technology. Among the many breakthroughs in this domain, the emergence of powerful language models has been a game-changer. These models, with their immense capacity to understand and generate human language, have found applications in a wide range of tasks, from natural language processing to content generation and translation. While their proficiency in text-related applications is well-documented, researchers and engineers have been increasingly exploring their capabilities beyond the realm of mere text understanding.

Language models (LMs) are versatile tools for numerous language processing tasks. However, these models come with certain limitations that hinder their capacity to deliver precise information. These limitations arise from their inability to access real-time updates, a tendency to generate fictitious information, limited mathematical capabilities, and a lack of comprehension regarding the temporal advancement. The Toolformer paper [7] aims to address these challenges by empowering the model with the capability to utilize external resources such as search engines, calculators, and text translation services.

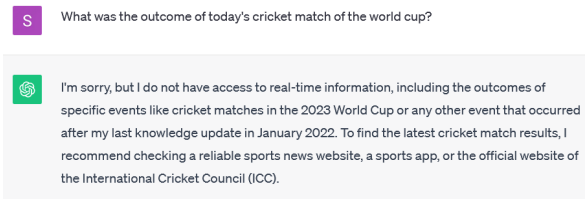

Figure 1: An example output for a real-time question to ChatGPT

Figure 1 illustrates an example where ChatGPT, a language model, encounters difficulty in providing a real-time answer regarding the outcome of the World Cup 2023 cricket match. This response underscores the limitations of language models, which generate answers based on the data they were trained on, making them less adept at handling questions about events that occur after their training.

Consequently, to ensure the accuracy of model responses, it becomes imperative to grant the model access to real-time data through various tools such as search engines and tool APIs. Along with this, it is also important that the model under-

stands which tools to use and when while preserving the model's existing functionality.

The Toolformer model was developed keeping all these requirements in mind. This model learns the usage of these external tools in a self-supervised way without feeling the need of huge amounts of human annotated data and ensures that the generality of the model is not lost. The current implementation of the Toolformer model is trained for these five tools: a question answering system, a Wikipedia search engine, a calculator, a calendar and a machine translation system.

Within the scope of our project, we have the objective of enhancing the model's functionality by delving into additional tools beyond those investigated by the authors. Specifically, we are contemplating the incorporation of tools such as the Wolfram API. The subsequent sections delve into the existing research in this domain, perform an experimental analysis on the current codebase, elaborate on the dataset generation for the chosen API, and outline the project contributions by each team member.

2 Related work

2.1 HTLM: Hyper-Text Pre-Training and Prompting of Language Models

In the paper [3], the authors present HTLM, a hyper-text language model trained on a massive 23 TB web crawl dataset, specifically using simplified HTML data from the common crawl. They employ a BART-style de-noising auto-encoder, allowing them to excel in zero-shot prompting by converting tasks into HTML format.They make use of HTML prompting for generation and classification tasks. They utilize HTML templates, either manually chosen or generated by the model through auto-prompting, to define the HTML structure of the task. The template is subsequently populated with the task input and mask tokens acting as placeholders for the output. The instantiated template serves as the prompt for the model. Given that BART models reconstruct the entire input, straightforward heuristics are employed to correspond to the text surrounding the masks and extract the ultimate output.Their experiments demonstrate that HTLM surpasses previous outcomes in zero-shot prompting for summarization by generating prompts that effectively capture the underlying meaning of each summarization dataset. The experiments show that fine-tuning the model on structured data improved the performance of the model drastically as compared to other models that were pretrained using natural text only.

2.2 PAL: Program-aided Language Models

Recently, Large language models (LLMs) have shown the ability to perform mathematical calculations when provided with limited number of samples due to the use of "chain-of-thought" technique. This technique allows the model to understand the problem statement by dividing the problem statement into a number of steps and then evaluating each step. Even though LLM's do perform pretty well in this step-wise approach, they are prone to making some logical mistakes in calculations. In an attempt to solve this problem, the authors of the paper [5] introduced Program-aided Language (PAL) models. These models read the natural language problem statements, then generates programs as the intermediate step used for reasoning and then runs these programs to a runtime engine like the Python interpreter. In this setup, the LLM's primary task is to decompose the problem into executable steps, while the computational aspect is handled by the runtime engine, in this case, the Python interpreter. This paper specifically uses the Python interpreter for the computation task. They make use of CODEX (code-davinci-002) as their base LLM. For their experimental analysis they make use of following benchmark datasets: GSM8K, ASDIV, MAWPS, SVAMP and BIG-Bench Hard. Experiments conducted by the authors also showed that the technique also worked when the base Language model was changed to a text language model like text-davinci-002 and text-davinci-003. The only limitation of this approach is that it guarantees the correctness of the results under the assumption that the programmatic steps themselves are accurate.

2.3 Toolformer: Language Models Can Teach Themselves to Use Tools

The paper [7] forms the base of our project. Language models though adept with various natural language tasks, usually do not perform well for calculations and do not have access to up-to-date information related to recent events. This paper attempts to solve this problem by providing the models the ability to access external tools like search engines, calculator and the calendar. Prior to the introduction of this paper, the existing sys-

tems required huge amounts of human annotated data. The authors in this paper introduce a new model - Toolformer that learns in a self-supervised way without requiring large amounts of human annotations and is able to determine which , when and how the external tool is to be used without losing generality of the LM. They aim at providing the model this ability to use the tools using the help of API calls. The only limitation is that it should be possible to represent the input and output of these API calls as text sequences. The training dataset is first augmented with API calls using the following steps- sampling API calls, filtering API calls and model finetuning, after which comes the inference, which is a regular decoding step until the response is identified. The paper mainly focuses on Question Answering, Calculator, Wikipedia search, Machine translation system and the Calendar as the external tools that would help address the limitations of the current Language models. They make use of a subset of the CCNet dataset for finetuning and GPT-J as their foundational model. The model is evaluated based on the SQuAD, Google-RE and T-REx subsets of the LAMA benchmark. Their experiments offered compelling evidence that Toolformer significantly enhances the zero-shot performance of a 6.7 billion parameter GPT-J model, enabling it to surpass the performance of a much larger GPT-3 model across various downstream tasks. Even though the model shows promising results, it still has a few limitations. One such limitation is that the the model is not able to use the external tools in a chain i.e use the output of the first tool as the input for the second tool. The second limitation is that the current implementation does not allow the tool to be used in an interactive way. Also, the models trained with Toolformer are sensitive to the exact wording of the input when deciding whether or not to call the tool APIs and are also sample efficient.

2.4 BLOOM: A 176B-Parameter Open-Access Multilingual Language Model

In the paper [6] the authors present the BLOOM model, which is a Transformer language model with a decoder-only architecture. BLOOM was trained on the ROOTS corpus, a dataset that encompasses hundreds of sources in a total of 59 languages, including 46 natural languages and 13 programming languages. BLOOM was developed by BigScience, a collaborative effort involving hundreds of researchers. It underwent training on the French government-funded Jean Zay supercomputer over a span of 3.5 months. The paper documents the entire process of creating BLOOM, starting from the construction of its training dataset ROOTS, all the way to its architecture and tokenizer design. Furthermore, the evaluation results of BLOOM and other large language models are discussed, revealing competitive performance that sees improvement through multitask finetuning. The authors anticipate that the release of this potent multilingual language model will unlock fresh possibilities and research avenues for large language models. Additionally, they hope that sharing their experiences will assist the machine learning research community in organizing new large-scale collaborative projects akin to BigScience. This collaborative framework not only enables outcomes that are unattainable for individual research groups but also encourages diverse individuals from various backgrounds to contribute their ideas and engage in the development of substantial advancements in the field.

2.5 TALM: Tool Augmented Language Models

In their work, the authors introduce Tool Augmented Language Models (TALM), a method that enhances language models by incorporating non-differentiable tools and utilizing an iterative "self-play" technique to improve performance, even with limited initial tool demonstrations. They make use of the pretrained T5 model for finetuning, inference and evaluation, and it is evaluated using knowledge-oriented Natural Questions (NQ) which is a diverse QA task and MathQA. Consistently, TALM demonstrates superior performance compared to a non-augmented language model on both a knowledge-oriented task (NQ) and a reasoning task (MathQA).The limitation of this approach is that it is only useful where a language model is finetuned for downstream tasks.

3 Implementation

3.1 Architecture

Our implementation is based on an existing work on the Toolformer [7]. In our case, to enable language models to scale its performance we use pretrained causal language models on the HuggingFace platform [2]. The original Toolformer pa-

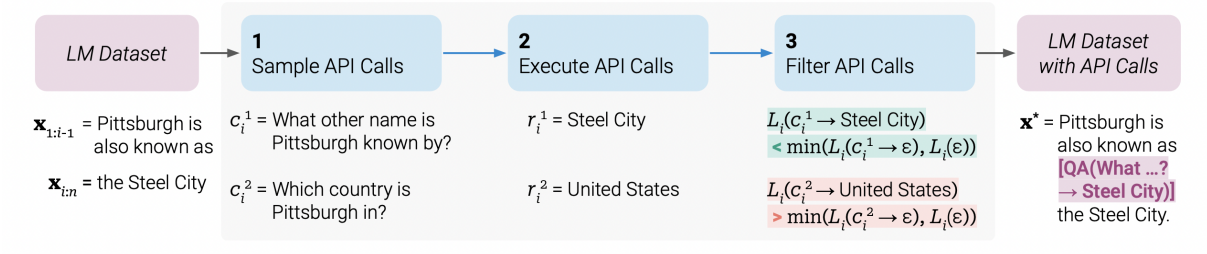

Figure 2: Overview of the steps of the toolformer [7]

per[7] employed GPT-J for training and finetuning. However, due to restricted GPU resources, we have integrated the Palmyra Small [8] model from HuggingFace instead. Like GPT-3, Palmyra Small belongs to a model family with a decoder-only architecture. It was primarily pre-trained on English text using self-supervised causal language modeling objectives. The primary function of Palmyra Small is to receive a text sequence and forecast the succeeding token. Palmyra comprises a total of *128 million* parameters and was trained on Writer's (a generative AI platform that makes LLM's accessible) custom dataset. We have referred a partial github implementation of the toolformer[4]. .

3.2 Toolformer: Approach and Analysis

In order to implement the Toolformer [7], we need to transform the existing dataset C to C* by augmenting API calls. The steps to transform the dataset is shown in Figure 2. These API calls support different tools. The inputs and outputs to these API calls are represented as sequences of text. The existing implementation uses special tokens to indicate the start and end of each API call. The start token is indicated by the '[', the end token by ']' and '->' is the output token. To convert the dataset into C* we follow 3 steps as shown in Figure 2.

- **Sampling API calls:** For each API call we store the prompt P as shown in the Prompts section. This encourages the LM to annotate the dataset with API calls. During sampling, we calculate the probability of the position of API calls for all positions $i \in \{1.., n\}$. From this, we sample up to k positions. We also store a sampling threshold value τ_s and keep the top k values that are greater than the sampling threshold i.e $I = \{i | p_i > \tau_s\}$.

[0]https://huggingface.co/Writer/palmyra-small

- **Obtain API response:** From each of the k positions obtained in the sampling, we obtain upto m API calls for each position $i \in I$ given the sequence $[P(x), x_1, ..x_{i-1}, [APIcall], x_i, x_n]$ where dataset $C = \{x_1, .., x_n\}$ and n is the length of the dataset. We then execute all the API calls that are invoked using internal utilities. Currently we have implemented the Calculator, Wolfram API and the Calendar API that are python scripts. The response of these API queries are a single text response.

- **Filtering API response:** In order to filter API calls, we consider a sequence of weights $(w_i | i \in N)$. The weighted cross-entropy loss is calculated by: $L_i(z) = \sum_{j=1}^{n} w_{j-1} * logp_M(x_j | z_j, x_{i:j-1})$ for M when prefixed by z. We consider two instances of the loss:
  - Positive loss: This is the weighted loss taken over all tokens $x_i, ..x_n$ if the API call and the result are included in the prefix of M. It is represented by L_i^+ : $L(e(c_i, r_i))$.
  - Negative loss: This is calculated as the minimum of the weighted loss when (i) doing no API call and (ii) doing an API call but getting no response. This is represented by L_i^- : $min(L_i(\epsilon), L(e(c_i, \epsilon)))$

  We consider a filtering threshold τ_f and filter out all API calls that have $L_i^- - L_i^+ < \tau_f$.

Model Finetuning and Inference: After sampling and filtering API calls, we augment all the APIs and interleave them with current input sequences that will give us the new dataset C*. In our implementation of the paper, the model is finetuned on Palmyra Small on the augmented prefix inputs. Finetuning on C* would further

help the LM determine which tool to be used when and how. In order to inference the generated output, we decode the response until we find the output token "->". The appropriate API call is made and decoding is continued.

3.3 Datasets

We used 3 different datasets to test the sensitivity of the model: c4, math_dataset, and SVAMP.

3.3.1 c4

To investigate the Calendar API's ability, we evaluated the model on the c4 dataset https://huggingface.co/datasets/c4. c4 is a colossal, cleaned version of Common Crawl's web crawl corpus[10]. It was based on Common Crawl dataset. We used the 'realnewslike' dataset variant due to its smaller size. The model picks facts that have dates in them to append the API.

3.3.2 math_dataset

In order to test the ability of the Calculator API and the Wolfram API, we evaluated the model on a math dataset math_dataset[1][11]. The math_dataset code generates mathematical question and answer pairs, from a range of question types at roughly school-level difficulty. This is designed to test the mathematical learning and algebraic reasoning skills of learning models. We used the 'arithmetic__add_or_sub_multiple' variant of the dataset. The model picks the evaluation expression from the math_dataset to calculate the value.

3.3.3 SVAMP

We used an additional SVAMP dataset[9] to test the model augmented with the Calculator API . The SVAMP dataset has structured word problems. It is a collection of simple math word problems focused on arithmetic. We created the data samples by integrating 'Body', 'Question' and 'Answer' tags. The model finds a match with an example if it includes the words "equal to", "total of", "average of" and "How many". It will then go on to associate an expression from the word example and further calculate a value.

[0]https://commoncrawl.org

[0]https://huggingface.co/datasets/math_dataset

[0]https://huggingface.co/datasets/ChilleD/SVAMP

3.4 Tools implemented

We have currently implemented the following 3 APIs:

- **Calculator:** We perform simple arithmetic functions and support basic 4 operations of '+', '-', '*' and '/'.
- **Calendar:** The second tool implemented is the Calendar that supports by appending only the current date to the prompt. It takes in no input.
- **WolframAPI:** We implement the WolframAPI to perform simple mathematical calculations given the worded problems as API input to perform mathematical operations.

3.5 Prompts

Below are the prompts that we used in order to sample API calls. Examples for the tools we used are shown below:

3.5.1 Calendar Prompt

```
calendar_prompt = """
Your task is to add calls to a
Calendar API to a piece of text.
The API calls should help you get
information required to complete
the text.  You can call the API
by writing "[Calendar()]" Here
are some examples of API calls:
Input:  Today is the first Friday
of the year.  Output:  Today is
the first [Calendar()] Friday of
the year.
Input:  The president of the
United States is Joe Biden.
Output:  The president of the
United States is [Calendar()] Joe
Biden.
Input:  The current day of the
week is Wednesday.
Output:  The current day of the
week is [Calendar()] Wednesday.
Input:  The store is never open
on the weekend, so today it is
closed.
Output:  The store is never open
on the weekend, so today [Calen-
dar()] it is closed.
Input:  <REPLACEGPT>
Output:  """
```

3.5.2 Calculator Prompt

calculator_prompt = """ Your task is to add calls to a Calculator API to a piece of text. The calls should help you get information required to complete the text. You can call the API by writing "[Calculator(expression)]" where "expression" is the expression to be computed. Here are some examples of API calls:
Input: The number in the next term is 18 + 12 x 3 = 54.
Output: The number in the next term is 18 + 12 x 3 = [Calculator(18 + 12 * 3)] 54.
Input: The population is 658,893 people. This is 11.4% of the national average of 5,763,868 people.
Output: The population is 658,893 people. This is 11.4% of the national average of [Calculator(658,893 / 11.4%)] 5,763,868 people.
Input: From this, we have 4 * 30 minutes = 120 minutes.
Output: From this, we have 4 * 30 minutes = [Calculator(4 * 30)] 120 minutes. Input: <REPLACEGPT> Output: """

3.5.3 Wolfram Prompt

wolfram_prompt = """ Your task is to add calls to a Scientific API to a piece of text that related to chemistry, math, physics. The questions should help you get information required to complete the text. You can call the API by writing "[Wolfram(question)]" where "question" is the question you want to ask. Here are some examples of API calls:
Input: The complex conjugate of 2 + 3i is 2 - 3i.
Output: The complex conjugate of 2 + 3i is [Wolfram("What is the complex conjugate of 2 - 3i")] 2 - 3i.
Input: Solve $x^2 + 4x + 6 = 0$. The answer is $x = -2 - isqrt(2)$
Output: Solve $x^2 + 4x + 6 = 0$. The answer is [Wolfram("Solve $x^2 + 4x + 6 = 0$")] $x = -2 - isqrt(2)$
Input: Given a sequence of numbers: 21.3, 38.4, 12.7, 41.6. The mean is 28.5
Output: Given a sequence of numbers: 21.3, 38.4, 12.7, 41.6. The mean is [Wolfram("What is the mean of 21.3, 38.4, 12.7, 41.6")] 28.5
Input: <REPLACEGPT>
Output: """

4 Experiments

This section summarizes the different experiments run and elaborates on the experimental parameters considered to obtain results for data generation and finetuning.

We investigate whether the inclusion of additional APIs enables the model to utilize tools without requiring further supervision and to determine the utility of calling an external API. To assess this, we carefully select distinct datasets tailored to the use of each API and evaluate their performance. Specifically, we examine the variance in model accuracy when using the simple calculator API compared to the Wolfram API across different math datasets. As a baseline, we also finetune the model with the original dataset to discern any potential improvements resulting from the augmented APIs. Our experimental framework is divided into two main sections: Data Generation (Section 4.2) and Model Finetuning (Section 4.3). In Table 1 we have shown some of the examples run with the existing code base. The filtering threshold value $\tau_f = 0.05$ and sampling threshold value $\tau_s = 0.1$.

4.1 Experimental Setup

Throughout all our experiments we have used Palmyra-Small(128M) as our model M. Our dataset C changes as per APIs to c4 for Calendar API, math_dataset and SVAMP for Calendar and Wolfram API. The change in dataset has been done due to the number of examples useful for each API. The c4 dataset does not contain any mathematical expression leading to most of the data being skipped for low scores. This has no

Examples	$L_i^- - L_i^+$	Useful
John has 20 apples and his friend gave him 7 more. John now has [Calculator(20 / 7)] 27 apples.	0.4047	No
From this, we have 3 * 30 minutes = [Calculator(3 * 30)] 90 minutes.	0.0682	Yes
Complex conjugate of 4 + 7i is [Wolframe("What is the complex conjugate of 4 + 7i")] 4 - 7i	0.0906	Yes

Table 1: Example API calls for different tools indicating the $L_i^- - L_i^+$ used for filtering API calls

significant change on the model.

4.2 Data generation

This sections details about the APIs used and the logic used while generating the data along with our observations.

4.2.1 Calendar API

The Calendar API used the inbuilt python date function to generate the date nearest to the context of the sentence. The context of the sentence is determined by searching for a number in the string and calling the date function with the closest value possible. For a sentence like `Today is the first Friday of the year.` The API gives back the Friday that is nearest to the current date. `Output: Today is Friday, December 1, 2023.` The filtering threshold $\tau_f = 0.25$ for the Calendar API.

4.2.2 Calculator API

The Calculator API is based on a simple Python script that and only supports the operators '+', '-', '*' and '/'. Given an input expression, it only considers the digits and operators and trims everything else. For sampling API calls we only process sentences that either (i) have at least two digits withing a maximum span of 16 tokens. and the result is augmented to the question (ii) contain one of the sequences "=", "equals", "equal to", "total of", "average of", "how many", or (iii) contain at least three numbers; for texts that only match the last criterion, we only keep a random subset of 1%. Table2 summarizes the number of examples useful from the dataset for each API. The filtering threshold $\tau_f = 0.0$ for the SVAMP dataset and $\tau_f = 0.3$ for the math_dataset for Calculator API.

4.2.3 Wolfram API

Similar to the Calculator API, the three criteria are used to work sample API calls. The differentiating factor is with the API being tolerent towards text. A prompt that generates for example `"Wolfram("What is the mean of $ 460 during 5 weeks of harvest")"` does not error. The filtering threshold $\tau_f = 0.0$ for the SVAMP dataset and $\tau_f = 0.3$ for the math_dataset for Wolfram API.

4.3 Model Finetuning

We generate $25,000$ examples for each API by distributing the generation over the number of GPUs (8 GPUS = 3125 examples each. The model is finetuned on 8 RTX2080TI GPUs with FP16. We use a block size of $1,024$ and train 10 epochs. The learning rate is $1e-5$ and weight decay is $2e-02$, We use adam as the optimizer with $\beta_0 = 0.9$ and $\beta_1 = 0.999$. We use the run_clm.py script from hugging face to train and evaluate the model on the datasets. The model Writer/Palmyra-small was finetuned on datasets augmented with the Calendar, Calculator and Wolfram APIs. The evaluation accuracy are as shown in Table 2.

	c4	math	SVAMP
Calendar API	0.81	NA	NA
Calculator API	NA	0.83	0.41
Wolfram API	NA	0.808	0.59

Table 2: Evaluation accuracy of APIs and different datasets

The evaluation accuracy of the Writer/palmyra-small when trained without the augmented dataset for math_dataset was 0.63 and on SVAMP was 0.41 respectively. The performance of the model significantly improves when trained on datasets augmented with APIs, i.e c4, math_dataset, SVAMP. The total number of examples picked by the data generation from math_dataset for arithmetic operations are 220919 across 8 devices. On the other hand, for SVAMP only a total of 5600 examples were picked. The nature of these two datasets are completely different. Where math_dataset has mathematical expressions in the examples, the hit rate is much higher leading to

	math_dataset			SVAMP		
	Total	**Found**	**% Used**	**Total**	**Found**	**% Used**
Calculator	220919	25000	11.3 %	700	8	1.1%
Wolfram	220919	25000	11.3%	700	326	46%

Table 3: Usage of Samples from the dataset

better performance when compared to SVAMP, which is in the form of structured word problems. The toolformer is sensitive to the exact wording of the input when deciding whether to call an API or not. This showed us, that the toolformer performance is very dataset specific. The usage of the samples is depicted in Table 3.

5 Conclusion

We implemented the Toolformer, a language model that learns how to use various tools like the calculators, calendar via simple API calls. We were limited on the resources with the GPU memory and were not able to test it on GPT-J models to reproduce the numbers as in the original paper. By fine tuning on a dataset augmented with API calls, it reduces the perplexity of future tokens. The implemented toolformer performs on par with the numbers mentioned in the research paper on the Writer/palymra model (a 128M parameter model). The current implementation shows that an one API may not necessarily be useful for a particular dataset. Generating the augmented data affects the finetuning of the model and the focus should be mainly on generating the right data. Since the calculator API only works with expressions, it needs to be tolerant to parenthesis and must work according to parenthesis rules. Currently the Calculator API can evaluate only simple arithmetic operations. In the future we plan to change the filtering threshold value and analyse the sensitivity of the model to the threshold.

6 Group Contributions

Madhushree came with the initial idea of building upon the Toolformer paper. Sumedha and Tanisha had discussions with the professor to formalize this idea and how to re-implement the paper. Sumedha conducted research on background and prior work in the literature to give us a good foundation on what had been done already and what we would be doing as a part of the project. We divided the implementation of the three APIs among the the team members. Sumedha worked on implementing the Calendar API. Madhushree worked on implementing Calculator API. Tanisha implemented the Wolfram API. Madhushree and Tanisha both trained and evaluated the model on different datasets and calculated the accuracy scores. They also calculated the usability of the APIs. All 3 of us contributed to the making of the presentation and slides as well as the writing and editing of the proposal and final report.

References

[1] David Saxton et al. “Analysing mathematical reasoning abilities of neural models”. In: *arXiv preprint arXiv:1904.01557* (2019).

[2] Thomas Wolf et al. “Transformers: State-of-the-Art Natural Language Processing”. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. DOI: 10.18653/v1/2020.emnlp-demos.6. URL: https://aclanthology.org/2020.emnlp-demos.6.

[3] Armen Aghajanyan et al. *HTLM: Hyper-Text Pre-Training and Prompting of Language Models*. 2021. arXiv: 2107.06955 [cs.CL].

[4] Conceptofmind. *Toolformer Repository*. Accessed Dec 6th 2023. 2022. URL: https://github.com/conceptofmind/toolformer.

[5] Luyu Gao et al. “PAL: Program-aided Language Models”. In: *ArXiv* abs/2211.10435 (2022).

[6] Teven Le Scao et al. “Bloom: A 176b-parameter open-access multilingual language model”. In: *arXiv preprint arXiv:2211.05100* (2022).

[7] Timo Schick et al. “Toolformer: Language Models Can Teach Themselves to Use Tools”. In: 2023.

[8] Writer Engineering Team. *Palmyra-base Parameter Autoregressive Language Model*. https://dev.writer.com. Jan. 2023.

[9] ChilleD. *SVAMP Dataset*. URL: https://huggingface.co/datasets/ChilleD/SVAMP.

[10] Hugging Face. *C4 Dataset*. URL: https://huggingface.co/datasets/c4.

[11] Hugging Face. *Math Dataset*. URL: https://huggingface.co/datasets/math_dataset.