An empirical study of validating synthetic data for formula generation

Usneek Singh
Microsoft
Bangalore, India

José Cambronero*
Google
Atlanta, USA

Sumit Gulwani
Microsoft
Redmond, USA

Aditya Kanade
Microsoft
Bangalore, India

Anirudh Khatry*
University of Texas at Austin
Austin, USA

Vu Le
Microsoft
Redmond, USA

Mukul Singh
Microsoft
Redmond, USA

Gust Verbruggen
Microsoft
Keerbergen, Belgium

Abstract

Large language models (LLMs) can be leveraged to help write formulas in spreadsheets, but formula data resources are scarce, impacting both the base performance of pre-trained models and limiting the ability to fine-tune them. Given a corpus of formulas, we can use another model to generate synthetic natural language utterances for fine-tuning. However, it is important to validate whether the natural language (NL) generated by the LLM is accurate for it to be beneficial for fine-tuning. In this paper, we provide empirical results on the impact of validating these synthetic training examples with surrogate objectives that evaluate the accuracy of the synthetic annotations. We demonstrate that validation improves performance over raw data across four models (2 open and 2 closed weight). Interestingly, we show that although validation tends to prune more challenging examples, it increases the complexity of problems that models can solve after being fine-tuned on validated data.

1 Introduction

Derived-column formulas in spreadsheets generate a new column by transforming existing columns in a table, and they have been shown to be challenging to write (Gulwani et al., 2012). To aid users in writing such formulas, we can ask for a description in natural language (Zhao et al., 2024). Unfortunately, since such formulas are sparse, pre-trained language models (especially smaller) struggle in generating them without fine-tuning (for example, one of our models, Phi-2, achieved a pass@10 score of only 0.03, indicating a very low success rate in generating the correct formulas within 10 attempts.).

To construct a dataset for fine-tuning, public spreadsheet workbooks can be used but they contain only tables and formulas, whereas a fine-tuning dataset also requires paired natural language (NL) descriptions corresponding to each (Table, Formula). Traditionally datasets for NL-to-code tasks have been manually annotated (Zhou et al., 2024; Austin et al., 2021). This is a time-consuming and expensive process. Leveraging LLMs, known for their text generation capabilities, is a viable alternative (Tan et al., 2024) assuming that the **synthetic NL generated by LLMs is accurate**, as recent studies have shown that quality is more important than quantity (Zhou et al., 2024; Li et al., 2023; Lozhkov et al., 2024).

In this paper, we leverage LLMs to predict the accuracy of synthetic NL using 3 surrogate objectives, and show empirical results of fine-tuning models on subsets of synthetic data that are accepted by these objectives. Fine-tuning models on validated subsets shows better performance in predicting formulas compared to using raw data. For example, GPT-4 fine-tuned on data validated by generating code in an alternate common programming language saw up to a 28% improvement in evaluation scores along with a 23% reduction in training time. Additionally, we observe that the models fine-tuned on validated data perform better on more complex problems. We also find that models fine-tuned on validated data still manage to learn to use functions removed during validation.

Our key contributions are as follows.

- We define three surrogate objectives (output prediction, alternative code generation, and classification) to predict accuracy of synthetic natural language in the NL-to-Formula task.
- We empirically analyze the effect of validating

*Work done at Microsoft

Findings of the Association for Computational Linguistics:
NAACL 2025, pages 7062–7069

synthetic data using these objectives on fine-tuning performance of different models.

2 Related work

Formula generation FlashFill (Gulwani, 2011; Gulwani et al., 2012) generates derived-column formulas by example, as users struggle with this task. SpreadsheetCoder (Chen et al., 2021b) suggests formulas from surrounding context in spreadsheets. FLAME (Joshi et al., 2024) is a small language model that understands formulas for tasks like repair and retrieval, but does not handle natural language. The NL-to-Formula (NL2F) task is introduced with a dataset obtained by converting the TEXT2SQL dataset to spreadsheet formulas (Zhao et al., 2024). Unlike (Zhao et al., 2024), our work centers on empirically evaluating different NL validation strategies.

LLMs for synthetic data Tan et al. (2024) discusses the applications of LLMs in data annotation for classification tasks. Goel et al. (2023) demonstrates the use of LLMs in the medical domain, where they assist in labeling data with expert verification. Wang et al. (2024), Kim et al. (2024), and Tang et al. (2024) explore human-LLM collaborative approaches for annotation and verification. There has been no comparison of NL validation techniques on synthetic NL for NL2F.

Data quality for LLM fine-tuning Chen and Mueller (2024) proposed an approach for automated filtering and verification of datasets to ensure high quality for LLM fine-tuning, leveraging the BSDetector (Chen and Mueller, 2023) to obtain confidence scores from LLM outputs. These techniques require existing ground truth labels (utterances) which are not available in our case. Zhou et al. (2024) and Li et al. (2023) manually curate data to demonstrate that instruction tuning with a small (< 1000) set of high-quality examples yields competitive results. While their work focuses on selecting examples based on alignment (already assuming correctness), our work evaluates technique-based selection on accuracy of NL instructions.

3 Validating synthetic data

Let $T = [C_i]_1^n$ be a table with n columns uniquely identified by a corresponding h_i label. A derived-column formula F is a formula where each leaf node in the AST (Abstract Syntax Tree) of F is either a constant value or a column identifier h_i.

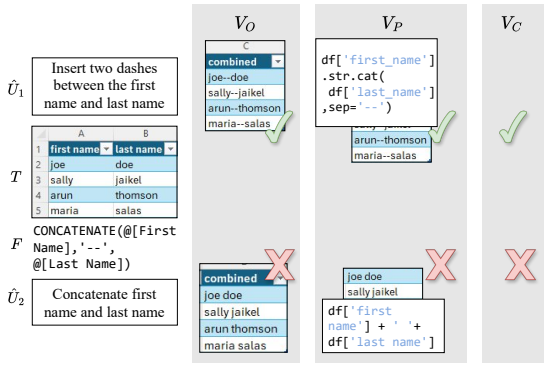

Figure 1: Overview of different validators implemented on top of GPT-4 represented by (a) $\boldsymbol{V_O}$: This validator directly computes $F(T)$ from $(\hat{U}, T)$; (b) $\boldsymbol{V_P}$: Validator predicts python program P from $(\hat{U}, T)$ to compare $P(T)$ with $F(T)$; (c) $\boldsymbol{V_C}$: Validator directly classifies $\hat{U}$ based on input $(\hat{U}, T, F)$.

Let U be an utterance in natural language that describes how to derive a column from T. A derived-column task is specified by (U, T, F). Given U and T the goal is to find a formula F' such that $F'(T) \equiv F(T)$, where equivalence indicates both formulas produce the same outputs given the same inputs.

To fine-tune a model, we therefore need examples of the form (U, T, F). T and F can be mined from large spreadsheet corpora (Singh et al., 2023; Joshi et al., 2024) and we can use an LLM to generate an utterance $\hat{U} = LLM(T, F)$.

A *validator* $V(\hat{U}, T, F) \rightarrow \mathbb{B}$ is a function that predicts whether $\hat{U}$ accurately describes the formula F operating on table T. These validators can be defined in any way—even using human annotators. To reduce manual effort, we define three validators using an LLM. An overview of these three validators is shown in Figure 1.

Output prediction (V_O) This validator asks the LLM to directly predict the output values $F(T)$ from $(\hat{U}, T)$ and uses an element-wise row comparison to evaluate correctness. For numbers, we allow an absolute difference of 0.05. For strings, we use a longest common sub-sequence ratio of 0.8 as passing criterion. This approach leverages natural language to emulate the computation directly. It is inspired from the alternate task of output prediction discussed in Khatry et al. (2023)

Alternate code generation (V_P) This validator asks the LLM to predict a program P in another language (we use Python) from $(\hat{U}, T)$ and compares $P(T)$ (execution of P on T) with $F(T)$ using

Excel Formula	Natural Langauge description
`[@TOTAL]-[@17-Jun]`	How do I compare the total number of ATMs in June 2017 with the total number of ATMs in December 2016 for each bank?
`IF(ROW()=ROW([]), IFERROR(OFFSET([@Date],-1, 0)+IncrRequest,Start))`	How do I generate a new date for each row based on the previous date and a fixed increment, starting from a given date?
`COUNTIFS([@Rushing Att],"="&[@Rushing Att], [Rk],"<="[@Rk])`	How do I rank the players by their rushing attempts in ascending order?
IF(OR([@1]="HEAD",[@1]="LIST"),"BLANK CELL","")	How do I hide the cells that have the format 'HEAD' or 'LIST' in the first column?
VALUE([@Column53])	How do I convert the text in column 53 to a date format?

Figure 2: Examples of cases filtered by validators implemented on top of GPT-4. The synthetic natural language descriptions in these examples are under-specified, contain incorrect intent, or convey an unclear idea.

element-wise comparison with the same relaxations for strings and numbers previously described. This leverages the abilities of LLMs to generate popular programming languages (Ni et al., 2023).

Classification (V_C) This validator directly asks the model to classify whether $\hat{U}$ accurately describes F over T. It is based on the self-reflection certainty objective from BSDetector (Chen and Mueller, 2023).

More details about the validators are provided in Appendix C. We also provide a few examples in Figure 2 to illustrate cases filtered by the validators from the raw dataset.

4 Experimental setup

We describe training data and models, and the testing benchmark.

Training data We mine (T, F) pairs that satisfy our derived-column definition from publicly available Excel workbooks (Singh et al., 2023). We create a training set and validation set of size 7833 and 422 respectively. Each (T, F) pair is annotated with an utterance $\hat{U}$ using GPT-4 at a low temperature.

Models We use two open (`phi-2` (2B) and `mistral-7b-instruct` (7B)) and two closed-weight (`gpt-35-turbo` and `gpt-4`) models. `phi-2` ($8 \times$ V100) and `mistral` ($1 \times$ A100) were fine-tuned for 10 and 15 epochs respectively. We selected the best checkpoint using validation loss. `gpt-35` ($16 \times$ A100) and `gpt-4` ($24 \times$ A100) were fine-tuned using the Azure API. `mistral`, `gpt-35`, `gpt-4` were fine-tuned using LoRA (Hu et al., 2021).

Testing data The SOFSET dataset (Barke et al., 2024) consists of 201 spreadsheet formula tasks from StackOverflow. Of these, we filter the 139 tasks that satisfy our derived-column definition.

Metric We use the pass@k metric (Chen et al., 2021a) based on execution match of formula, were k represents the number of predictions considered out of the total number of predictions provided. In our evaluation system, we generate $n = 10$ predictions at temperature 0.6 and compute pass@5 metric.

5 Results and Discussion

We perform experiments to empirically explore the following research questions.

RQ1 How do different validators compare?

RQ2 What is the impact of validating data on fine-tuning performance?

RQ3 What are the differences in cases solved by models trained on validated NL and raw dataset?

RQ4 Can models finetuned on validated data learn the functions removed during validation?

5.1 RQ1: Comparing validators

We apply our three validation approaches to our initial set of 7833 points. This produces the data subsets described in Table 1. We shows properties of the formulas accepted by each validator. Since V_O is bottle-necked on numerical operations, it succeeds for fewer unique functions and operators. Similarly, V_P struggles with more functions than V_C as there might not be an easy Python equivalent.

Figure 3 shows overlap in examples accepted by different validators. Each validator uniquely

Table 1: Summary of training data subsets with different validation approaches. "# functions" refers to unique functions, "# calls" to average function calls, "depth" to function nesting level, and "# ops" to average arithmetic operator count in formulas.

V	Size	# functions	# calls	depth	# ops
$\varnothing$	7833	122	1.03	0.87	1.28
V_O	2266	71	0.71	0.65	1.01
V_P	4095	95	0.86	0.77	1.22
V_C	5246	109	0.87	0.79	1.24

accepts at least some examples. 1403 (18%) examples does not pass any validator.

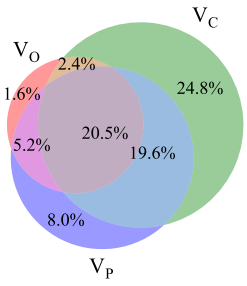

Figure 3: Summary of overlap of different data subsets produced by different validation strategies.

5.2 RQ2: Effect on fine-tuning performance

We compare the impact of validated data versus raw (unvalidated) data, as well as the impact of validated data versus rejected cases by each validator, on the downstream performance of the NL2F task.

Versus raw Table 2 shows base model (few-shot) and fine-tuning performance on different subsets of data. For the smaller models, `phi-2` and `mistral`, the performance increase with fine-tuning is more significant. With all models, a smaller, validated dataset yields better performance than raw data. ***V_P* yields the best performance on average with nearly half the size of raw data.** `gpt-4` improves only when fine-tuned on validated data. Surprisingly, `gpt-35` without fine-tuning outperforms the fine-tuned version, likely due to differences in data distribution between training and testing benchmarks. Besides performance, fine-tuning with validated data also reduces training time significantly, as shown in Table 3. We see the performance on dataset created by the intersection of all validators (marked by $\cap$) is limited by the worst performing validator in each case.

Versus invalidated Table 4 compares the performance of fine-tuning on the accepted (& subsampled) and rejected ($\neg$) examples for each validator. We sub-sample the accepted sets to 2266—the number of examples in the smallest set (V_O). We observe that, despite the smaller size of the validated data subset (subsampled), it outperforms its larger invalidated (rejected) counterpart in most (11/12) comparisons. The only case where this not happens is for V_O on `gpt-4`, likely due to the many functions (51) that were eliminated from the training data.

Table 2: Performance comparison of the different models on SOFSET Benchmark using pass@5 metric. Three out of the four models give best performance when fine-tuned on data validated by V_P.

FT on	# ex	phi-2	mistral	gpt-35	gpt-4
Base	0	0.02	0.04	**0.30**	0.36
Raw	7833	0.08	0.16	0.29	0.32
V_O	2266	0.06	**0.17**	0.28	0.35
V_P	4095	**0.10**	0.14	**0.30**	**0.41**
V_C	5246	0.09	0.14	**0.30**	0.40
$\cap$	1607	0.06	0.13	0.27	0.34

Table 3: Training time and relative improvement for different models on data subsets. Models fine-tuned on V_P and V_C subsets require less time than on raw data while delivering better downstream performance.

Data	phi-2	mistral	gpt-35	gpt-4
Raw	15h44m	8h51m	4h45m	14h00m
V_O	-73%	-71%	-60%	-47%
V_P	-48%	-45%	-37%	-23%
V_C	-36%	-32%	-19%	-19%

Table 4: Pairwise comparison of performance of subsampled ($\subset$) data from validated (V) against rejected ($\neg V$) examples. Results of pairs ($\subset V, \neg V$) are marked in green if ($\subset V > \neg V$), blue if ($\subset V = \neg V$).

FT on	# ex	phi-2	mistral	gpt-35	gpt-4
V_O	2266	0.06	0.17	0.28	0.35
$\neg V_O$	5567	0.05	0.16	0.27	0.35
$\subset V_C$	2266	0.07	0.14	0.27	0.36
$\neg V_C$	2587	0.04	0.12	0.26	0.34
$\subset V_P$	2266	0.08	0.12	0.31	0.37
$\neg V_P$	3738	0.05	0.11	0.24	0.32

5.3 RQ3: Analysing solved cases

Figure 4 shows properties of the solved cases (where at least one prediction was correct) after fine-tuning different models on raw data and validated subsets. We see that fine-tuning on datasets with fewer unique functions still enables all models (except for `mistral`) to solve cases with more unique functions. The average function call count

increases for validated subsets compared to the raw data, indicating more complex formulas are solved by models fine-tuned on validated data. For `gpt-4` and `gpt-35`, average operator count also increases with fine-tuning on validated data.

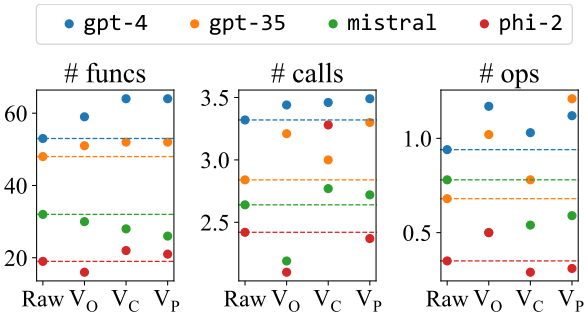

Figure 4: Comparison of correctly solved cases on models fine-tuned with different validation subsets based on (a) Number of unique functions (b) Average number of function calls (c) Average operator count of formulas

5.4 RQ4: Recovery of functions removed during validation

By analyzing the output generations of the fine-tuned models, we identify new functions that were not present in their base model (without fine-tuning) predictions. Our results show that there are functions that were not in the fine-tuning dataset *and* were not in the base model's predictions, but after finetuning on validated datasets, we see these functions used in trained model predictions (see Table 5). This suggests that fine-tuning on a high-quality dataset allows the model to remember knowledge that it had learned during pre-training, without teaching it to hallucinate on potential mistakes in the synthetic data.

Table 5: Number of functions learned by different fine-tuned models that were removed during validation. Some examples include 'SUMIF', 'TIME', 'QUOTIENT','ROWS','AGGREGATE'

	`phi-2`	`mistral`	`gpt-35`	`gpt-4`
V_0	1	2	2	3
V_P	3	2	3	3
V_C	1	1	2	2

5.5 Recommendations

From our study, we see that a single validator (Alternate Code generation) works best on 3 out of 4 models. We also see that Output Prediction validator shows lower performance in general, likely because many functions (51) were removed from the training data during validation. However, practitioners should experiment with different validation methods, starting with the subset here, as validation in general improves performance.

6 Conclusion

We empirically evaluate the effect of automated validation of synthetic data using LLMs on the fine-tuning performance of derived-column NL-to-formula. We validate synthetic NL annotations with three surrogate tasks (classification, code generation in Python, and output prediction) and fine-tune different models on the examples accepted by each of these methods. In general, fine-tuning on smaller, validated datasets improves performance. Despite validation resulting in datasets with simpler formulas, that does not cause the fine-tuned models to only solve simpler problems. Models fine-tuned on validated data are able to recover some functions that were removed during validation.

7 Limitations

Although we have focused on validating the correctness of natural language instructions, we have not addressed techniques for correcting them. Exploring methods for correcting instructions could be beneficial, as it would prevent the loss of data points. While having a smaller set of high-quality data can be advantageous for efficient training, achieving the best results may require maintaining a larger dataset by correcting invalid instructions.

In our study, the distribution of training data for fine-tuning is different than the testing data, which might not fully reflect the potential of fine-tuning. Additionally, our research has concentrated on formulas that expect a single, well-structured (formatted) input table. We aim to extend our work to include formulas that involve multiple tables and unstructured input. Furthermore, we have explored the potential of our technique in one language (English). We believe it will be valuable to investigate multilingual systems for validation setups.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Shraddha Barke, Christian Poelitz, Carina Suzana Negreanu, Benjamin Zorn, José Cambronero, Andrew D Gordon, Vu Le, Elnaz Nouri, Nadia Polikarpova, Advait Sarkar, et al. 2024. Solving data-centric tasks using large language models. *arXiv preprint arXiv:2402.11734*.

Jiuhai Chen and Jonas Mueller. 2023. Quantifying uncertainty in answers from any language model and enhancing their trustworthiness.

Jiuhai Chen and Jonas Mueller. 2024. Automated data curation for robust language model fine-tuning. *arXiv preprint arXiv:2403.12776*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021a. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Petros Maniatis, Rishabh Singh, Charles Sutton, Hanjun Dai, Max Lin, and Denny Zhou. 2021b. Spreadsheetcoder: Formula prediction from semi-structured context. In *International Conference on Machine Learning*, pages 1661–1672. PMLR.

Akshay Goel, Almog Gueta, Omry Gilon, Chang Liu, Sofia Erell, Lan Huong Nguyen, Xiaohong Hao, Bolous Jaber, Shashir Reddy, Rupesh Kartha, et al. 2023. Llms accelerate annotation for medical information extraction. In *Machine Learning for Health (ML4H)*, pages 82–100. PMLR.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330.

Sumit Gulwani, William R Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Harshit Joshi, Abishai Ebenezer, José Cambronero Sanchez, Sumit Gulwani, Aditya Kanade, Vu Le, Ivan Radiček, and Gust Verbruggen. 2024. Flame: A small language model for spreadsheet formulas. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 12995–13003.

Anirudh Khatry, Joyce Cahoon, Jordan Henkel, Shaleen Deep, Venkatesh Emani, Avrilia Floratou, Sumit Gulwani, Vu Le, Mohammad Raza, Sherry Shi, Mukul Singh, and Ashish Tiwari. 2023. From words to code: Harnessing data for program synthesis from natural language. *Preprint*, arXiv:2305.01598.

Hannah Kim, Kushan Mitra, Rafael Li Chen, Sajjadur Rahman, and Dan Zhang. 2024. Meganno+: A human-llm collaborative annotation system. *arXiv preprint arXiv:2402.18050*.

Ming Li, Yong Zhang, Zhitao Li, Jiuhai Chen, Lichang Chen, Ning Cheng, Jianzong Wang, Tianyi Zhou, and Jing Xiao. 2023. From quantity to quality: Boosting llm performance with self-guided data selection for instruction tuning. *arXiv preprint arXiv:2308.12032*.

Anton Lozhkov, Loubna Ben Allal, Leandro von Werra, and Thomas Wolf. 2024. Fineweb-edu.

Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I. Wang, and Xi Victoria Lin. 2023. Lever: learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org.

Mukul Singh, José Cambronero Sánchez, Sumit Gulwani, Vu Le, Carina Negreanu, Mohammad Raza, and Gust Verbruggen. 2023. Cornet: Learning table formatting rules by example. *Proceedings of the VLDB Endowment*, 16(10):2632–2644.

Zhen Tan, Alimohammad Beigi, Song Wang, Ruocheng Guo, Amrita Bhattacharjee, Bohan Jiang, Mansooreh Karami, Jundong Li, Lu Cheng, and Huan Liu. 2024. Large language models for data annotation: A survey. *arXiv preprint arXiv:2402.13446*.

Yi Tang, Chia-Ming Chang, and Xi Yang. 2024. Pdfchatannotator: A human-llm collaborative multimodal data annotation tool for pdf-format catalogs. In *Proceedings of the 29th International Conference on Intelligent User Interfaces*, pages 419–430.

Xinru Wang, Hannah Kim, Sajjadur Rahman, Kushan Mitra, and Zhengjie Miao. 2024. Human-llm collaborative annotation through effective verification of llm labels. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 1–21.

Wei Zhao, Zhitao Hou, Siyuan Wu, Yan Gao, Haoyu Dong, Yao Wan, Hongyu Zhang, Yulei Sui, and Haidong Zhang. 2024. Nl2formula: Generating spreadsheet formulas from natural language queries. *arXiv preprint arXiv:2402.14853*.

Chunting Zhou, Pengfei Liu, Puxin Xu, Srinivasan Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, et al. 2024. Lima: Less is more for alignment. *Advances in Neural Information Processing Systems*, 36.

A Training Data Characteristics

In this section, we summarise important formula properties for the training data extracted from excel workbooks (see Table 6). From the original corpus, we remove any formulas that have deprecated functions to produce a set of 10,389 (table, formula) pairs. We then remove any pairs where the formula results in a missing/empty value for all output rows or uses multiple tables. After the process of filtering, our final dataset consists of 7,833 (table, formula) pairs. This dataset has formulas which use 122 distinct built-in functions. The most popular functions match those typically employed by Excel spreadsheet users: `IF`, `SUM`, `IFERROR`, `CONCATENATE`, `AND`. The other properties are summarised in Table 6). The function call count refers to the frequency of Excel function calls within a formula. The depth of formulas denotes the extent of nested function calls within them. Operator count is the number of arithmetic operators (+, -, *, /) in a formula.

Table 6: Characteristics of formulas used in Training Data obtained from Excel spreadsheets

	Fxn. call count	Formula depth	Op. count
0	3554	3554	2887
1	2625	2682	2811
2	965	1030	1169
3	285	325	435
4	115	125	187
≥ 5	289	113	344

B Model hyper-parameters used while Fine-tuning

Phi-2 For the Phi-2 model, fine-tuning was performed for 10 epochs with a batch size of 8. The learning rate was set to 1e-6, and the Adam optimizer was used along with a cross-entropy loss function.

Mistral The Mistral model was fine-tuned for 15 epochs using the LoRA technique (Hu et al., 2021). The specific parameters for LoRA included a LoRA rank ($Lora\_r$) of 64, a LoRA alpha ($Lora\_alpha$) of 16, and a LoRA dropout ($Lora\_dropout$) of 0.1. The target modules for LoRA adaptation were "q_proj", "k_proj", "v_proj", "o_proj", "gate_proj", "up_proj", "down_proj", and "lm_head". No bias configuration was used, and the task type was Causal Language Modeling (CAUSAL_LM). The learning rate for this model was set to 2e-4, and the batch size was 8. Optimization was carried out using the PagedAdamW 32-bit optimizer.

GPT-35 and GPT-4 They have been fine-tuned using LoRa on default settings used for these models in Azure API documentation[1].

C Technical Details of validators

In this section, we provide the details about the prompts used with each validator in the above study. We use greedy decoding for all prompts to ensure more precise computation.

Output Prediction We use an LLM (GPT-4) as a validator V. We prompt the LLM with input table and NL to compute the output for the target column directly. Then we validate the target output by comparing them with the actual outputs, with validation deemed successful only if the expected and actual outputs match for all rows in the table. The matching criteria differs based on the datatype: for a numeric value we allow an absolute difference of up to 0.05 and a string is considered a match when the longest matching contiguous subsequence coefficient (defined as length of longest matching sub-sequence divided by length of the longer string) is greater than 0.8. The prompt used for this technique is provided in Figure 5.

```
Below is an instruction that describes a
task. Write a response that
appropriately completes the request. The
assistant answer questions from a table
by predicting the expected output for
each row Input.  Refer to these examples
for guidance.

Instruction: #example NL

Input_Table: #example Table
```

Figure 5: Prompt used for Output Prediction validation

Alternate code generation We use an LLM (GPT-4) as V and task it with Python generation using NL and table as the input. The matching criterion is same as that of Direct computation. The prompt is provided in Figure 6.

Classification We prompt an LLM (GPT-4) as validator V to generate a binary outcome, judging whether the given natural language query ac-

[1] https://learn.microsoft.com/en-us/azure/ai-services/openai/

```
Your objective is to provide a Python
code that can effectively execute the
provided natural language query on the
table. Refer to these examples for
guidance.

Instruction: #example NL

Input_table: #example table
```

Figure 6: Prompt used for Alternate code generation validation

curately describes the formula when applied to the corresponding table. The prompt is provided in Figure 7.

```
Your task is to evaluate the
appropriateness of a given natural
language query in describing the formula
applied to a table.  Provide a binary
response of 'Yes' or 'No' indicating
whether the query effectively captures
the essence of the formula's execution
on the table. Refer to these examples
for guidance.

Instruction: #example NL

Formula: #example Formula

Input_table: #example Table
```

Figure 7: Prompt used for Classification validation