Alternate Task Technique: Improving Natural Language to Code in Low-Resource Languages

Anonymous ACL submission

Abstract

When using large language models (LLMs) to generate code from natural language (NL2Code), the target programming language influences precision. We empirically observe that LLMs are more likely to produce correct code when the target language is a popular language and struggle for low-resource target programming languages. Prompt engineering approaches can address the problem to some extent, but can not fully close the gap between popular and low-resource target languages. We introduce "alternate task technique" that uses an LLM to perform a surrogate task whose results are combined with the LLM's results on the original task to improve performance on the original NL2Code task. Using SQL, Python Pandas, and Power Query language M as three targets, we show that our approach brings the performance of LLMs on the low-resource M language significantly closer to its performance on the more popular Python Pandas and SQL languages.

1 Introduction

The emergence of Large Language Models (LLMs), such as Codex (Chen et al., 2021) and GPT (Brown et al., 2020; OpenAI, 2023), has fundamentally transformed the field of program synthesis from natural language (NL), leading to the rapid development of NL interfaces (i.e., PowerAutomate; PowerBI Q&A) and code assistants (GitHub Copilot) that are widely used by practitioners and developers. In-context learning (Dong et al., 2023) plays a crucial role in this transformation, enabling LLMs to generate code for a diverse range of programming languages with minimal input. The input to LLM typically consists of a prompt describing the task to be performed in natural language and, potentially, a few examples, which are used for few-shot prompting (Brown et al., 2020; Logan IV et al., 2021). Our focus is on developing LLM-based solution for NL to code tasks using in-context learning as it is cost-effective: it does not require training and deployment of custom models for specific programming domains (e.g., SQL, Python) and it also avoids data-hungry and compute-heavy fine tuning.

The simplicity and strength of in-context learning makes it an excellent choice for synthesizing data manipulation programs, i.e. programs that are designed to extract, query, and transform data using NL descriptions. There are plenty of data manipulation languages. Some, such as SQL and Python Pandas, are popular, while others, such as PowerQuery M (Power Query), are not as well represented in the publicly-available documents and code repositories on which the LLMs are trained. Unsurprisingly, we observed that LLMs perform poorly when generating code in M, compared to when the target language is Python Pandas or SQL. How can we improve the performance of LLMs on generating code in a low-resource language? One potential approach is by adding more information about the target language in the prompt, perhaps in the form of few-shot examples; however, it was not sufficient to close the performance gap.

In this paper, we introduce the *alternate task technique* (ATT), where three agents come together to solve a task. The first agent solves the original task, the second agent solves a different, but closely related, task, and the third agent consolidates the two outputs. The agents may be implemented using LLMs or other tools. ATT is a general principle that can be instantiated in many different ways. Several recent works in the literature can be viewed as instances of ATT. Specifically, in recent work (Chen et al., 2023a), the authors solve the NL to code task by also solving the alternate "NL to test case generation" task, and then consolidating the generated code and the generated test cases. In another recent work (Du et al., 2023), the alternate task is picked to be identical to the first task, and the two answers are consolidated using multi-turn debate between the first two agents. Thus, there is no explicit third

agent in (Du et al., 2023) and its job is performed by a multi-turn consensus between the two agents. In yet another work on multiagent debate (Liang et al., 2023), an explicit third agent is deployed to consolidate the results from the debating agents.

We instantiate the ATT approach on the "NL to data-manipulation code" task, especially for generating code in low-resource languages. This task differs from the more general "NL to code" task in that there is a fixed *data context*: in addition to the NL description of the intended task, we also have the dataset on which the task needs to be performed. We use the alternate *output prediction* task: generate the *output* dataset from the NL task description and input dataset. We then compare the predicted outputs with the output generated by execution of the predicted programs, and use the result to pick the program that is most likely to be the correct program. Thus, our third agent is a tool that executes generated programs and checks if the execution results are consistent with predicted outputs.

While prior work has also utilized the data context, it is largely limited to using the data context for creating better prompts for the original task (Poesia et al., 2022; Pourreza and Rafiei, 2023; Trummer, 2022). In this paper, we use (a sample of) the input dataset to also predict the output from the NL.

Our Contribution. In this paper, we present the alternate task technique (ATT) and evaluate it on data-manipulation code generation for multiple targets. The alternate task leverages the data context in previously unexplored ways. We show that ATT gives small improvements on the task of generating code in popular targets, such as SQL and Pandas, where we are able to get close to or beat the state-of-the-art without any fine-tuning or custom model learning. Moreover, it produces significant gains when the target is a low-resource language, namely PowerQuery M, and helps close the accuracy gap between the popular and niche targets.

2 Motivating Scenario

In this section, we motivate our solution using an example NL statement that we need to translate to a Power Query M (Power Query) expression. M is a data manipulation language used in business intelligence applications, such as Microsoft's Excel and PowerBI (PowerBI), and is typically used to filter, transform and combine data from one or more sources. Since it is a custom language used by specialized tools, LLMs perform poorly on tasks that involve generating M expressions.

Consider a scenario where a user is working in Power Query with a table `Source` containing a column named `'Path'` with some text data. The user issues an NL query: *"Trim the end of all contents in column "Path" by one character"* and wants an M expression for it (this is a real query which was scraped from a Help Forum[1]).

We first prepare the best possible prompt to send the LLM. This includes the NL query, the schema of the table `Source`, some sample rows from `Source`, and some examples of NL queries alongside the corresponding M expressions (few-shot prompting). From this prompt, the LLM returns the M expression: `Table.AddColumn(#"Source","Trimmed",each Text.End([Path]),type text)`. This expression is syntactically well formed, but it is semantically incorrect as `Text.End` requires two arguments and not one (the second argument denotes the number of characters to be selected). It is quite common for LLMs to have a poor Top-1 accuracy and better Top-25 accuracy; see Table 3. So, our first agent asks the LLM to generate N=25 candidate M expressions. How do we pick the correct expression from this list?

Let us use ATT and ask a second agent to solve a related task and then use those results to find the correct M expression. The second agent is tasked to predict directly the output table (not the M code) from the NL statement and the input table. The second agent is also implemented using an LLM, and it generates N=25 predictions. In the case of our running example, the correct output table is present in the list of predictions. Finally, the third agent consolidates the two outputs. It reranks the 25 program candidates based on whether the outputs they produce (when executed) match the predicted outputs, and return the top candidate after reranking. The third agent is not LLM-based, but an external tool. It is able to rerank and get the correct M expression at the top: `Table.TransformColumns(#"Source", {{"Path", each Text.Start(_, Text.Length(_)-1), type text}})`.

Note that the second agent can solve the alternate task in other ways. For example, the second agent can ask the LLM to generate Pandas code

[1] https://stackoverflow.com/questions/72548765/trimming-end-of-column-in-power-query

and then execute that code to generate the output. In fact, the correct line of code: `df["Path"] = df["Path"].str[:-1]`, where the dataframe `df` is a reference to `Source`, is more likely to be generated by LLMs. The agents are free to pick any approach to solve the task assigned to them.

3 Alternate Task Technique

The *alternate task technique* (ATT) involves designing a task that is similar, but not necessarily identical, to the original task and using the information from this alternate task to accomplish the original task.

Suppose we have a task to predict the random variable Y given the random variable X; that is, we want to sample from $Pr(Y \mid X = x)$. Suppose we can find an alternative random variable Z (related to Y) such that we can sample from $Pr(Z \mid X = x)$ as well as estimate $Pr(Y = y, Z = z \mid X = x)$ extremely well. In such a case, we can apply ATT to predict Y given $X = x$ using 3 agents as follows:
(a) Agent 1 generates a baseline sample $\{y_1, \ldots, y_n\}$, with associated probabilities $\{p_1, \ldots, p_n\}$, by sampling from $Pr(Y \mid X = x)$.
(b) Agent 2 generates an alternate sample $\{z_1, \ldots, z_m\}$, with associated probabilities $\{q_1, \ldots, q_m\}$, by sampling from $Pr(Z \mid X = x)$.
(c) Agent 3 computes an updated estimate of probability $Pr(Y = y_i \mid X = x)$ as $\alpha p_i + (1 - \alpha) p'_i$, where p'_i is a new estimate computed as $\sum_{j=1}^{m} Pr(Y = y_i, Z = z_j \mid X = x) * q_j$. We are free to choose the value of the parameter α, which we picked to be $1/2$ in our experiments. The updated probabilities for each candidate y_i computed in Step (c) are then used to to pick the best candidate.

Each agent can be implemented using an LLM or a tool. In our experiments, we use LLMs to implement Agent 1 and Agent 2, and use a tool as Agent 3. The reason why ATT works is because Agent 2 potentially uses a "different sequence of neuron firings" when it is tasked to predict Z (than what Agent 1 uses to predict Y), and thus it provides us with new knowledge. ATT is a general principle that can be instantiated in many different ways depending on how we pick Z and how we use $Pr(Z \mid X = x)$ to update $Pr(Y \mid X = x)$. We focus here on one particular instantiation for NL to code that uses output prediction as the alternate task, and an execution-based tool as Agent 3.

As noted earlier, the recent work (Chen et al., 2023a) is an instance of ATT and other works (Du et al., 2023; Liang et al., 2023) are multiturn versions of ATT where only one sample is generated in each round and the three agents engage in multiple rounds to update their predictions (probabilities). The ATT principle itself is not explicitly stated in any earlier work.

4 NL To Code

We begin by first describing the problem setup, followed by our approach that instantiates ATT in a particular way.

Problem Setup. The problem we consider can be stated as follows: *Given a natural language description nl of some task along with the dataset D over which that task should be performed, our objective is to generate the expression or program s in a given target language that will accomplish the task.* As we mentioned before, the so-called data context, which consists of the dataset D, is clearly an important source of information for generating the desired program s. The most common way of exploiting the data context is to include a summary of D in the prompt that is passed to the LLM. This step is now standard (Brown et al., 2020; Chen et al., 2022, 2021; Nijkamp et al., 2022) and we do *not* consider the problem of prompt generation in this paper. Instead, in all our experiments, we used the best possible prompt we could design for the task by leveraging the prompts proposed in the existing literature; see Appendix 9.2 and 9.3.

We are interested in improving Top-1 accuracy, as in multiple other works (Gu et al., 2023; Fu et al., 2023), especially for low-resource target languages. Our motivation comes from scenarios where some user is working in a data processing application that employs its own domain-specific language, and the user wants to generate code for a natural language query nl. The user wants one program, and not $K > 1$ programs because the user has to review the generated program(s) to determine if it is correct and reviewing multiple programs is distracting and increases cognitive load.

LLM parameters. We assume access to an LLM that accepts a prompt, a temperature value, and a number N, and returns N completions for the prompt sampled from a distribution that has entropy proportional to the set temperature. For each response, the LLM also returns a probability value associated with each token of the response. These

Algorithm 1 Overall approach for generating program given NL description nl and dataset D.

Require: An NL description nl of some task and an input dataset D
Ensure: Return the program that performs task nl on D

1: **function** NL2CODE(nl, D)
2: prompt $\leftarrow$ PREPAREPROMPT(nl, D)
3: L, probs $\leftarrow$ LLM(prompt, tmp=0.6, N=24) $\cup$ LLM(prompt, tmp=0, N=1)
4: O, Oprobs $\leftarrow$ OUTPUTPREDICTION(nl, D)
5: scores $\leftarrow$ GETSCORES(L, probs, O, Oprobs, D)
6: **return** Top candidate from L by score

probabilities, returned in logarithmic form (logprobs (Shi et al., 2022)), measure how likely each token is to occur in the context of the previous tokens and the given prompt; see also (OpenAI API).

Overall Approach. Our overall approach is outlined in pseudocode in Algorithm 1:

Prompt generation The description nl and a summary of the dataset D, consisting of the schema and a small sample of D, is put together into a prompt that also includes some few-shot examples of the task (Line 2); see Appendix 9.2.

Temperature Mixing The LLM is asked to generate 25 candidates L (and their probabilities) using high temperature to increase diversity, but we also add the top candidate generated with temperature 0 into the mix (Line 3). Temperature mixing is discussed further in Section 4.3.

Output prediction We use the LLM to generate multiple predictions O (along with probabilities) for the output of "executing" the nl task on the input D (Line 4). Output prediction is discussed further in Section 4.1.

Score calculation Each candidate is assigned a default score based on probability returned by the model (Shi et al., 2022), which is then further refined based on the predicted outputs O (Line 5). This is discussed further in Section 4.1 and 4.2.

4.1 Output-Prediction based Score Tuning

The key new steps in our NL to code pipeline coming from the use of ATT are shown in Algorithm 2. Specifically,
(1) We create a prompt for output prediction on Line 2; see Appendix 9.3.
(2) We use the LLM to predict 25 possible outputs O, along with their probabilities Oprobs, given the prompt (with temperature 0.6) on Line 3.

Algorithm 2 Output-prediction based Score Tuning

Require: An NL description nl, an input sample dataset d, candidates L, and their logprobs, execution results $O_{\texttt{exe}}$ for L, a function sim that measures similarity between two outputs s.t. it returns 1 when outputs are equal and value between 0 and 1 otherwise
Ensure: Scores scores for each candidate in L

1: **function** OUTPUTPREDICTION(nl, d)
2: prmpt $\leftarrow$ PROMPTFOROUTPUTPREDICTION(nl, D)
3: O, Oprobs $\leftarrow$ LLM(prmpt, tmp=0.6, N=25)
4: **return** O, Oprobs
5: **function** GETSCORES(L, probs, O, Oprobs, d)
6: **for** $s \in L$ **do** ▷ for each candidate s
7: $o_{\texttt{exe}} \leftarrow$ Execute s on d ▷ output generated by s
8: score$[s] \leftarrow$ probs$[s] *$ DQM$(o_{\texttt{exe}})$ ▷ initialize
9: **for** $o \in O$ **do** ▷ for each predicted output
10: $p \leftarrow$ Oprobs$[o] *$ sim$(o, o_{\texttt{exe}})$
11: score$[s] \leftarrow$ score$[s] + p$
12: **return** score

(3) On Lines 8-11 we assign a new score to each candidate s in the list L of candidates equal to:

$$\texttt{probs}(s) + \sum_{o \in O} \texttt{Oprobs}(o) * Pr(s \mid o, nl, D),$$

where $Pr(s \mid o, nl, D)$ is an estimate of the probability that s is the desired program given o is the desired output on D, and probs(s) is the original probability estimate of s given nl, D.
(4) We estimate $Pr(s \mid o, nl, D)$ by using any similarity metric on the output space (on Line 10) to compare o with the output $o_{\texttt{exe}}$ generated by executing s on D. The simplest metric is one that returns 1 if $o == o_{\texttt{exe}}$ and 0 otherwise; however, one could use other metrics.

To relate back to ATT, note that Steps (1)-(2) correspond to Agent 2 and Steps (3)-(4) describe Agent 3. Furthermore, Oprobs$(o) * Pr(s \mid o, nl, D)$ is an estimate of $Pr(s, o \mid nl, D)$, which is just $Pr(Z = z, Y = y \mid X = x)$. Note that the final score we are assigning to a candidate is a sum of the probability estimates we get using two different ways: directly from $Pr(Y \mid X = x)$ and indirectly from $\sum_z Pr(Y, Z = z \mid X = x)$, consistent with the choice of $\alpha = 1/2$ in ATT.

Output prediction using LLMs can be accomplished in other ways too. One could use the inputs nl and D to generate code in a different target (than what the user wants), such as Python, and then execute that to predict outputs.

4.2 Well-formedness based Score Tuning

We add one further signal to the score assigned to a program s, namely the well-formedness of the output $o_{\texttt{exe}}$ that would be generated by s. For

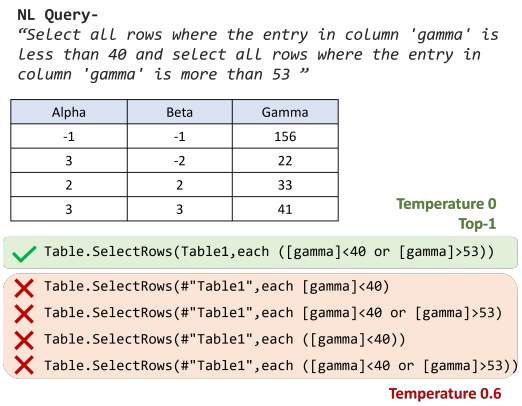

Figure 1: Example of *temperature mixing*: The correct program is ranked second using logprobs at *temperature 0.6*. Adding *temperature 0* candidate bumps the correct program to the first rank.

example, if the output table contains a new column of null values, then we can mark this output as being less likely. Specifically, we scale the logprob-based probability value assigned to a candidate s by a factor $\texttt{DQM}(o_{\texttt{exe}})$ that computes a data quality metric for $o_{\texttt{exe}}$ (Line 8). In our experiments we used a simple data quality metric that penalizes s if $o_{\texttt{exe}}$ has null columns or if $o_{\texttt{exe}}$ is an empty table.

4.3 Temperature Mixing

Our approach relies on the correct program being present in the list of candidate programs generated by the LLM. However, in cases where the LLM does not have sufficient prior knowledge of the data manipulation language, it is possible that the correct program is not included in the candidate set produced by the LLM. Therefore, we augment the candidate set with additional programs. This augmentation is done by leveraging the LLM itself and the fact that it performs temperature sampling (Brown et al., 2020).

In the context of code generation, we noticed that the quality of programs synthesized from the LLM varies significantly with changing temperatures. In fact, there is a tradeoff. At higher temperatures, we get diverse N samples, but the top-1 accuracy drops because the N samples can exclude the one that has the highest average logprobs (e.g., the program that would be surfaced when temperature is set to 0). On the other hand, at lower temperatures, we get the highest average logprobs candidate, but we lose diversity and the N samples tend to contain the same candidate multiple times, which makes reranking unproductive.

To mitigate these issues and avoid missing correct programs, we introduce temperature mixing into our approach. In particular, we generate programs at both a low and a high temperature (i.e., 0 and 0.6, respectively), concatenate the results, as shown on Line 3 of Algorithm 1, and then apply the our reranking approach based on ATT. Temperature mixing is particularly effective when the model is more uncertain about the output, which can happen either because the query is ambiguous or very complex, or if the target language is unfamiliar to the model. In these cases, sampling at low temperature is important because the probability distribution computed by the model already has high entropy (more uncertainty) and lowering the temperature helps bring down the uncertainty.

5 Experimental Evaluation

We perform our evaluation on three different target languages: SQL, Power Query M, and Pandas.

The Benchmarks. For Pandas, we used the "Jigsaw" dataset (Jain et al., 2022). Since the M expression language is limited and there are not any available public benchmarks, we leveraged the Jigsaw dataset to create a benchmark for M. In particular, we filtered the Jigsaw dataset and extracted only the transformations that M supports to create the "JigsawM" benchmark set for M. Additionally, we also created another dataset by scraping PowerQuery help forums and collecting M expressions' NL descriptions and M expressions from there, which we call "Forum" in the tables. It also includes some benchmarks we obtained from the PowerQuery team. For SQL, we used the "Spider" dev (Yu et al., 2018) and "KaggleDBQA" (Lee et al., 2021) datasets.

Table 1 presents some statistics about the benchmarks. Each benchmark consists of a number (Column #n) of pairs of NL statements and the associated code. The average number of characters in the NL description and code are given in the two columns named "Avg" in Table 1.

Metrics. We use *execution match accuracy* as the metric for evaluation. A candidate *execution matches* the ground truth if both programs return identical outputs when run on the input dataset. We report the percent of benchmarks where we get an execution match, also called *semantic match* (SM). We also report *exact match accuracy* (EM) – where we test if the candidate syntactically matches the ground truth. Since a task can be performed

Benchmark		Statistics			Baseline		ATT	
Target	Name	# n	Avg(nl)	Avg(code)	SM	EM	SM	EM
M	Forum	59	59.35	80.81	54.2	24.1	67.8	27.8
M	JigsawM	442	65.66	75.83	19.7	6.8	64.7	20.2
Pandas	JigsawM	442	65.66	50.61	67.2	21.7	69.2	23.6
Pandas	Jigsaw	793	70.49	56.47	71.2	20.8	74.1	23.9
SQL	Spider	1034	68.04	108.32	73.2	26.9	76.0	28.8
SQL	KaggleDBQA	272	55.79	96.00	62.7	38.9	63.2	38.9

Table 1: Consolidated results. For each target language (Target) and benchmark name (Name), the column #n is the number of (nl query, code) pairs in that benchmark set, Avg(nl) is the average length of the nl, Avg(code) is the average length of the code, SM is the semantic match (execution match) accuracy and EM is the exact match accuracy, reported both for the baseline (without alternate task) and for our approach using alternate task technique.

in many different ways, we do not use it to draw conclusions and report it just for completeness.

Relatively poor accuracy on M. From the baseline results in Table 1, we see that LLMs get execution match (SM) accuracy of 60%-75% for SQL and Pandas, but only 19%-55% for M. On the same JigsawM benchmark, accuracy for baseline was 19.7% for M, while it was 67.2% for Pandas. Clearly, accuracy is consistently poorer for M. Note that the baseline approach uses the same prompt as our approach. The prompt includes few-shot examples, column names, input table name, and sample rows; see Appendix 9.2. In other words, the baseline uses the *best possible prompt* we could design for the task. We also used the best choices for temperature and the number of candidates to generate and rank by average logprobs. We did not consider as baseline any approaches that utilize custom ML models or require fine-tuning large language models (see Section 7) as we don't want to make any assumption about availability of training data. It is also worth noting that our baseline already surpasses the SOTA using prompt engineering (Pourreza and Rafiei, 2023) on SQL as depicted in the Spider leaderboard (Spider).

Alternate task technique closes the gap. The results for alternate task technique (ATT) in Table 1 show that we improve the execution match (SM) accuracy to 64%-68% for M – an improvement of 13.6% on Forum benchmarks and 45% on JigsawM. In comparison, the improvement for SQL and Pandas was limited to 0-3%. For all targets and all benchmarks, our ATT approach yields consistent execution match accuracy in the range 63%-76%. Thus, our approach disproportionately benefits M code generation. This is not surprising since the alternate task technique was designed for target languages that are not well-represented in the LLM's training data. This partially provides evidence for the intuition that output prediction potentially exploits alternate new pathways of the LLM, which helps us extract new additional information from the LLM to use to solve the original task – especially in the case when the direct use of LLM for the task yields poor results.

Table 2: Gains in execution match accuracy from temperature mixing.

Target	Benchmark	SM gain
M	Forum	+1.3
M	JigsawM	+3.4

Improvement from Temperature Mixing. We next evaluate the gains from temperature mixing. This is also a technique that helps for languages such as M that are not well-represented in LLM's training data. Temperature mixing only adds one candidate from the temperature-0 run. Table 2 reports gains for M in the range 1.3% to 3.4% for execution match accuracy coming from temperature mixing. This shows that the alternate task technique is the main contributor of the gains for M, but temperature mixing aids it by adding new candidates to the pool.

6 Discussion and Future Work

A key assumption underlying our output-prediction based scoring technique is that candidates generated by the LLM can be executed inside a try-catch block. This assumption is easy to satisfy for languages that have few or no side-effects. This is the case for the PowerQuery M target language. For such languages, we can use execution-based

score tuning in production. However, when the language is richer and general purpose, such as Python, models like Codex can generate programs that have negative side-effects (e.g., deleting files, etc). Fortunately, as our results show here, baseline techniques that just use prompt engineering and logprobs-based ranking already provide good accuracy for languages like Python Pandas.

We note that some of gains from using ATT come from demotion of candidates that do not successfully execute – either because they are syntactically ill-formed or throw runtime exception. This demotion of candidates that do not generate outputs happens automatically in our technique.

Comparison with Fine-tuning and Custom Models. Our evaluation does not consider baselines that require custom model training or fine-tuning because both those steps are data hungry and not cost effective. We observed that text-davinci-002 fine tuned with order of several thousand (NL, code) pairs performed poorly and gave 0% execution match on the benchmarks reported in our evaluation – indicating insufficient data.

Chain-of-thought and alternate task technique. Chain-of-thought (CoT) prompting (Wei et al., 2022) refers to the technique of prompting the LLM that encourages the model to verbalize the intermediate reasoning steps used for solving the task. Mathematically, CoT estimates $P(Y = y|X = x)$ by $P(Y = y|Z = z, X = x) * P(Z = z|X = x)$; that is, by going through Z. In CoT, estimates for both $P(Y|X, Z)$ and $P(Z|X)$ are performed by the LLM. In the terminology of ATT, CoT employs Agent 2 to generate Z, but does not use Agent 1 to directly generate Y, and instead uses Agent 3 to generate Y from Z and X. We can say that CoT uses $\alpha = 0$, eliminates Agent 1, and merges Agent 2 and Agent 3 into one agent that does both steps.

Cost Overhead for ATT. Our approach involves making 2 calls (3 if using temperature mixing), but the two calls ask for N=25 completions. This incurs only a small additional computational cost because we are requesting more tokens in the output, but we do not incur any additional cost for input tokens because they are sent just once. For example, the cost (estimate based on the count of input and output tokens used) for the NL2SQL evaluation is USD 1.35 for Kaggle and USD 8.60 for Spider for N=25 generations. If we instead perform only N=1 generation, the cost would be USD 1.06 for Kaggle and USD 6.47 for Spider. So, going from N=1 to N=25 incurs only a small overhead. Table 3 shows that generating 25 candidates improves the chances of getting the correct candidate in the pool, and ATT helps Top-1 accuracy get close to Top-25.

Table 3: NL2M top-K accuracy at temp=0.6 for k=1, 5, 25: The first number in each cell is Exact Match accuracy and second number is execution match accuracy.

	k = 1	k = 5	k = 25
Forum	24.1, 54.2	45.9, 72.6	51.0, 74.3
JigsawM	06.8, 19.7	27.9, 57.3	41.7, 73.4

7 Related Work

Few-shot Prompting. Our contributions are not related to few-shot prompting, but we use few-shots in our prompts. Few-shot prompting refers to inclusion of some concrete examples of the task in the prompt. It has been shown to help the LLM generate good program recommendations (Brown et al., 2020; Chen et al., 2022, 2021; Nijkamp et al., 2022; Liu et al., 2021), including recommendations in less popular languages (Hendy et al., 2023). A wide collection of work exists on few-shot prompting ranging from crafting prompt templates (Shin et al., 2020; Zhong et al., 2021; Gao et al., 2020; Shi et al., 2022), considering the permutations of examples (Zhao et al., 2021; Lu et al., 2021), to increasing the number of few-shot examples (Wei et al., 2022). Given LLM's sensitivity to prompts, many works exist in prompt aggregation(Arora et al., 2022), or training models that perform aggregations itself (Jiang et al., 2020; Schick and Schütze, 2020), as well as chain-of-thought prompting (Liu et al., 2023a), and, more recently, repair (Chen et al., 2023b; Shinn et al., 2023), but we leave these as potential directions for future work.

Data Context. Since we are operating in the domain in which *data is available*, we tested various ways to summarize the associated input data in the prompt as it is well-known that small changes in the prompt can have significant effects on the generated programs (Min et al., 2022). Examples include using encoding the input data within `CREATE SQL` statements, introducing new tokens like `<T>` for demarking table names as well as, simple dictionaries that list each table and its associated column attributes and types (Scholak et al., 2021; Shaw et al., 2021). The most performant representation for tables was as a Python list-of-list, which we

used; see Appendix 9.2. Similar to existing work (Gemmell and Dalton, 2023), we include a sample of 3-8 rows per table in the prompt.

Natural Language to Code. The Spider leaderboard (Spider) contains a list of works that leverage machine learning for text-to-SQL generation and are evaluated on the Spider dataset. The approaches fit into three categories: custom ML models (Li et al., 2023; Fu et al., 2023; Cao et al., 2021; Xu et al., 2021), prompt engineering with pre-trained language models such as Codex and GPT-4 (Pourreza and Rafiei, 2023; Poesia et al., 2022), and fine-tuned large language models (Scholak et al., 2021; Shaw et al., 2021). Our work falls into the second category as we operate under the assumption that we do not have enough data to train a custom model or to fine-tune a large language model. The top performance results in this category are obtained by the work in (Pourreza and Rafiei, 2023). This work achieves 74.2% and 69.9% top-1 execution accuracy on the Spider dev test (the dataset we are also using for our evaluations) using the GPT-4 and Codex models respectively. Our approach provides 76% top-1 execution accuracy using the Codex model demonstrating that we are able to surpass the SOTA methods using ATT.

In the context of Pandas, the most relevant work to ours is the one published in (Jain et al., 2022). The main difference is that their method requires input/output test cases from the user. These tests are used to validate and refine the programs generated by the LLM, or to modify the LLM-produced code so that it can satisfy the test cases. In contrast, our method solely relies on the natural language utterance and does not require any additional tests.

Reranking. Generating code from natural language is challenging (Yu et al., 2018; Chen et al., 2021; Austin et al., 2021; Li et al., 2022). Since the desired code is more likely to be generated when multiple programs are sampled, there is extensive work around designing reranking techniques, including execution-based reranking techniques, to select the best candidate among multiple samples (Shi et al., 2022; Zhang et al., 2022; Ni et al., 2023; Li et al., 2022). A lot of work has focused on improving Top-1 accuracy (Shi et al., 2022; Ni et al., 2023; Zhang et al., 2022). Unlike our work, some works consider a different signal for reranking: namely, translating the code back the NL and checking consistency, which is related to maximizing mutual information objective to pick the top candidate (Liu et al., 2023b; Li et al., 2016; Zhang et al., 2022), which we can integrate in our score-based reranking framework. We introduce the new alternate task technique, and its instantiation to output-prediction based score tuning, which translates the NL to output and checks for consistency to rerank.

Tool Plugins in LLMs. Recently, there is work on coupling tools with LLMs (Chen et al., 2022; Schick et al., 2023; Yao et al., 2023) and output-prediction based score tuning can be seen as a way to improve performance of an LLM using an external tool, namely an interpreter. ATT is a specific way of coupling tools that uses LLMs ability to solve the original and a slightly different task, and then a tools ability to consolidate all the information extracted from LLMs.

8 Conclusion

In this paper, we presented a novel technique, called the alternate task technique (ATT), for synthesizing data manipulation programs from natural language (NL) and an input dataset. Our approach leverages the input dataset by asking the LLM to "execute" the NL on the input dataset to generate candidate output datasets. The predicted outputs are used to rerank the programs predicted directly by the LLM. We evaluate our framework over SQL, Pandas, and PowerQuery M, using a variety of new and existing benchmarks. We observe that ATT provides small gains for popular target languages SQL and Pandas as much, but adds significant accuracy gains for M, and makes accuracy on M comparable to that for Pandas and SQL. ATT can serve as a general methodology for improving LLM accuracy on tasks that involve knowledge of some niche and low-resource domain.

Limitations

First, the alternate task technique is observed to add significant value only when generating code in low-resource languages. The gains for popular target programming languages was limited. Second, we have used LLMs to perform the surrogate task of output prediction. This task can become considerable hard if the NL descriptions become more complicated and the input dataset starts to have large number of rows and columns. Our benchmarks did not contain such hard instances. For such hard cases, to restore feasibility of our overall approach, it might become necessary to generate

outputs by other means – such as, using the LLM to generate Pandas code, and executing it on the input, rather than asking the LLM to directly produce the output. We believe that we will get same results for that modification, but that hypothesis has to be rigorously evaluated. Third, our approach makes one additional call to the LLM to perform the surrogate task. LLM calls require compute and incur cost. Fourth, our proposed approach is based on executing code generated by an LLM. In general, this is untrusted code and not safe for execution outside of a sandbox environment.

References

Simran Arora, Avanika Narayan, Mayee F Chen, Laurel J Orr, Neel Guha, Kush Bhatia, Ines Chami, Frederic Sala, and Christopher Ré. 2022. Ask me anything: A simple strategy for prompting language models. *arXiv preprint arXiv:2210.02441*.

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*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Ruisheng Cao, Lu Chen, Zhi Chen, Yanbin Zhao, Su Zhu, and Kai Yu. 2021. LGESQL: Line graph enhanced text-to-SQL model with mixed local and non-local relations. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2541–2555, Online. Association for Computational Linguistics.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023a. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023b. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, Lei Li, and Zhifang Sui. 2023. A survey on in-context learning.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. 2023. Improving factuality and reasoning in language models through multiagent debate.

Han Fu, Chang Liu, Bin Wu, Feifei Li, Jian Tan, and Jianling Sun. 2023. CatSQL: Towards real world natural language to sql applications. *Proc. VLDB Endow.*, 16(6):1534–1547.

Tianyu Gao, Adam Fisch, and Danqi Chen. 2020. Making pre-trained language models better few-shot learners. *arXiv preprint arXiv:2012.15723*.

Carlos Gemmell and Jeffrey Dalton. 2023. Generate, transform, answer: Question specific tool synthesis for tabular data. *arXiv preprint arXiv:2303.10138*.

GitHub Copilot. 2021. GitHub Copilot. `https://github.com/features/copilot`.

Zihui Gu, Ju Fan, Nan Tang, Lei Ju Cao, Bowen Jia, Sam Madden, and Xiaoyong Du. 2023. Few-shot text-to-sql translation using structure and content prompt learning. SIGMOD.

Amr Hendy, Mohamed Abdelrehim, Amr Sharaf, Vikas Raunak, Mohamed Gabr, Hitokazu Matsushita, Young Jin Kim, Mohamed Afify, and Hany Hassan Awadalla. 2023. How good are gpt models at machine translation? a comprehensive evaluation. *arXiv preprint arXiv:2302.09210*.

Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1219–1231.

Zhengbao Jiang, Frank F Xu, Jun Araki, and Graham Neubig. 2020. How can we know what language models know? *Transactions of the Association for Computational Linguistics*, 8:423–438.

Chia-Hsuan Lee, Oleksandr Polozov, and Matthew Richardson. 2021. Kaggledbqa: Realistic evaluation of text-to-sql parsers. *arXiv preprint arXiv:2106.11455*.

Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023. Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql.

Jiwei Li, Michel Galley, Chris Brockett, Jianfeng Gao, and Bill Dolan. 2016. A diversity-promoting objective function for neural conversation models. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 110–119. Association for Computational Linguistics.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Zhaopeng Tu, and Shuming Shi. 2023. Encouraging divergent thinking in large language models through multi-agent debate.

Aiwei Liu, Xuming Hu, Lijie Wen, and Philip S Yu. 2023a. A comprehensive evaluation of chatgpt's zero-shot text-to-sql capability. *arXiv preprint arXiv:2303.13547*.

Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2021. What makes good in-context examples for gpt-3? *arXiv preprint arXiv:2101.06804*.

Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin G. Zorn, Jack Williams, Neil Toronto, and Andrew D. Gordon. 2023b. "what it wants me to say": Bridging the abstraction gap between end-user programmers and code-generating large language models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 598:1–598:31. ACM.

Robert L. Logan IV, Ivana Balazevic, Eric Wallace, Fabio Petroni, Sameer Singh, and Sebastian Riedel. 2021. Cutting down on prompts and parameters: Simple few-shot learning with language models.

Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. 2021. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. *arXiv preprint arXiv:2104.08786*.

Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837*.

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. *arXiv preprint arXiv:2302.08468*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

OpenAI. 2023. Gpt-4 technical report.

OpenAI API. OpenAI API. `https://platform.openai.com/docs/api-reference/completions/create`.

Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of text-to-sql with self-correction.

Power Query. Power Query M. `https://learn.microsoft.com/en-us/powerquery-m/`. Accessed: June 21, 2023.

PowerAutomate. 2023. AI in PowerAutomate. `https://powerautomate.microsoft.com/en-us/`.

PowerBI. 2023. PowerBI. `https://powerbi.microsoft.com`.

PowerBI Q&A. 2023. Q&A in PowerBI. `https://learn.microsoft.com/en-us/power-bi/natural-language/q-and-a-intro`.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools.

Timo Schick and Hinrich Schütze. 2020. It's not just size that matters: Small language models are also few-shot learners. *arXiv preprint arXiv:2009.07118*.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: parsing incrementally for constrained auto-regressive decoding from language models. *CoRR*, abs/2109.05093.

Peter Shaw, Ming-Wei Chang, Panupong Pasupat, and Kristina Toutanova. 2021. Compositional generalization and natural language variation: Can a semantic parsing approach handle both? In *Proc. 59th Annual Meeting of the Assoc. for Comput. Linguistics and the 11th Intl. Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 922–938, Online. Association for Computational Linguistics.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I Wang. 2022. Natural language to code translation with execution. *arXiv preprint arXiv:2204.11454*.

Taylor Shin, Yasaman Razeghi, Robert L Logan IV, Eric Wallace, and Sameer Singh. 2020. Autoprompt: Eliciting knowledge from language models with automatically generated prompts. *arXiv preprint arXiv:2010.15980*.

Noah Shinn, Beck Labash, and Ashwin Gopinath. 2023. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366*.

Spider. The Spider leaderboard. https://yale-lily.github.io/spider.

Immanuel Trummer. 2022. Codexdb: Synthesizing code for query processing from natural language instructions using gpt-3 codex. *Proceedings of the VLDB Endowment*, 15(11):2921–2928.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*.

Peng Xu, Dhruv Kumar, Wei Yang, Wenjie Zi, Keyi Tang, Chenyang Huang, Jackie Chi Kit Cheung, Simon J.D. Prince, and Yanshuai Cao. 2021. Optimizing deeper transformers on small datasets. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2089–2102, Online. Association for Computational Linguistics.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

Tianyi Zhang, Tao Yu, Tatsunori B Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida I Wang. 2022. Coder reviewer reranking for code generation. *arXiv preprint arXiv:2211.16490*.

Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate before use: Improving few-shot performance of language models. In *International Conference on Machine Learning*, pages 12697–12706. PMLR.

Zexuan Zhong, Dan Friedman, and Danqi Chen. 2021. Factual probing is [mask]: Learning vs. learning to recall. *arXiv preprint arXiv:2104.05240*.

9 Appendix

9.1 Comparision with CoT and Machine-Translation

One can wonder about performing NL2M task by first doing NL2Pandas and then translating Pandas to M. We note that writing semantic preserving translators from one language to another is always a tricky and time-consuming task. Moreover, such translators need to be maintained and kept up to date. Our approach avoids the need for writing translators.

9.2 Prompt For Code Generation

```
The assistant answers questions from a
table by converting them to PowerQuery M
queries.

Columns:Country/Region,Lat,Long
Sample Data:[
        ["USA","50","100"],
        ["India","23","160"],
        ["Australia","-40","180"]
        ]
Table Name:Regions
Question:Put first row as headers
M:Table.PromoteHeaders(
            Regions,
            [PromoteAllScalars=true]
            )

Columns:Country/Region,Lat,Long
Sample Data:[
        ["USA","50","100"],
        ["India","23","160"],
        ["Australia","-40","180"]]
Table Name:World Table
Question:Group column "Country/Regions" by Count
M:Table.Group(#"World Table",
                {"Country/Region"}, {
                    {
                    "Count", each
                      Table.RowCount(_),
                    Int64.Type}
```

```
                    }
            )

Columns:First,Last,Id
Sample Data:[["Adam","Baker","123"],
            ["Brian","Doe","234"],
            ["Barbara","Davis","567"]]
Table Name:Personal Details
Question:Only include first names that start
        with B in column First
M:Table.SelectRows(#"Personal Details",
           each Text.StartsWith([First], "B"))

Columns:First,Last,Id
Sample Data:[["Adam","Baker","123"],
            ["John","Doe","234"],
            ["Clark","Davis","567"]]
Table Name:Source
Question:Rotate the table
M:Table.Transpose(Source)

Columns:name,surname,id,pos
Sample Data:[["aAdamb","Baker","123","Engineer"],
        ["aJohnb","Doe","234","Researcher"],
        ["aClarkb","Davis","567","Manager"]]
Table Name:Details Table
Question:Extract the contents between "a" and "b"
        in column "name"
M:Table.AddColumn(#"Details Table",
            "Text between delimiters",
            each Text.BetweenDelimiters(
                [name], "a", "b", 0, 0),
            type text)

Columns:{Columns}
Sample Data:{Snippet of the data}
Table Name:{Table Name}
Question:{NL Query}
M:
```

9.3 Prompt For Alternate Task

The assistant answers questions from a table by showing how the data is transformed in Power Query when given the description of the transformation task.

```
Columns:First,Last,Id
Sample Data:[["Adam","Baker","123"],
            ["Brian","Doe","234"],
            ["Barbara","Davis","567"]]
Table Name:Personal Details
Question:Only include first names that
        start with B in column First
Output Table:[["Brian","Doe","234"],
            ["Barbara","Davis","567"]]
Column Outputs:First,Last,Id

Columns:Country/Region,Lat,Long
Sample Data:[["USA","50","100"],
            ["India","23","160"],
            ["Australia","-40","180"]]
Table Name:World Table
Question:Add 10 to all the values
        in column "Lat"
Output Table:[["USA","60","100"],
            ["India","33","160"],
            ["Australia","-30","180"]]
Column Outputs:Country/Region,Lat,Long

Columns:name,surname,id,pos
Sample Data:[
     ["aAdamb","Baker","123","Engineer"],
     ["aJohnb","Doe","234","Researcher"],
     ["aClarkb","Davis","567","Manager"]
     ]
Table Name:Details Table
Question:Extract the contents between "a" and
    "b" in column "name"
Output Table:[
      ["Adam","Baker","123","Engineer"],
      ["John","Doe","234","Researcher"],
      ["Clark","Davis","567","Manager"]
      ]
Column Outputs:name,surname,id,pos

Columns:name,surname,id,pos
Sample Data:[
     ["aAdamb","Baker","123","Engineer"],
     ["aJohnb","Doe","234","Researcher"],
     ["aClarkb","Davis","567","Manager"]
      ]
Table Name:Details Table
Question:Remove the first letter from the
        "surname" column
Output Table:[["aAdamb","aker","123","Engineer"],
        ["aJohnb","oe","234","Researcher"],
        ["aClarkb","avis","567","Manager"]]
Column Outputs:name,surname,id,pos

Columns:Country/Region,Lat,Long
Sample Data:[["USA","50","100"],
            ["India","23","160"],
```

```
                ["Australia","-40","180"]]
Table Name:World Table
Question:Add new column "Lat+10" by
        adding 10 to all
        the values in column "Lat"
Output Table:[["USA","50","100","60"],
                ["India","23","160","33"],
        ["Australia","-40","180","-30"]]
Column Outputs:Country/Region,Lat,Long,
                Lat+10

Columns:{columns}
Sample Data:{Sample Data}
Table Name:{Table Name}
Question:{ NL Query}
Output Table:
```