Realistic Training Data Generation and Rule Enhanced Decoding in LLM for NameGuess

Yikuan Xia[1], Jiazun Chen[1], Sujian Li[2], Jun Gao[1]*
[1]Key Laboratory of High Confidence Software Technologies, CS, Peking University, China
[2]State Key Laboratory of Multimedia Information Processing, CS, Peking University, China
2101111522@pku.edu.cn,chenjiauzn@stu.pku.edu.cn,{lisujian,gaojun}@pku.edu.cn

Abstract

The wide use of abbreviated column names (derived from English words or Chinese Pinyin) in database tables poses significant challenges for table-centric tasks in natural language processing and database management. Such a column name expansion task, referred to as the NameGuess task, has previously been addressed by fine-tuning Large Language Models (LLMs) on synthetically generated rule-based data. However, the current approaches yield suboptimal performance due to two fundamental limitations: 1) the rule-generated abbreviation data fails to reflect real-world distribution, and 2) the failure of LLMs to follow the rule-sensitive patterns in NameGuess persistently. For the data realism issue, we propose a novel approach that integrates a subsequence abbreviation generator trained on human-annotated data and collects non-subsequence abbreviations to improve the training set. For the rule violation issue, we propose a decoding system constrained on an automaton that represents the rules of abbreviation expansion. We extended the original English NameGuess test set to include non-subsequence and PinYin scenarios. Experimental results show that properly tuned 7/8B moderate-size LLMs with a refined decoding system can surpass the few-shot performance of state-of-the-art LLMs, such as the GPT-4 series. The code and data are presented in the supplementary material.

1 Introduction

As a key structure for organizing information, tabular data is widely used in various domains, from web applications to enterprise databases. Abbreviated column names are commonly used to simplify expressions and comply with database constraints. However, these abbreviations often harm downstream tasks. For example, Text2SQL (Yu et al., 2018), schema-based relation detection (Koutras et al., 2021), and table QA tasks (Yin et al., 2020) suffer performance drops of 10.54, 40.50, and 3.83 percentage points, respectively (Zhang et al., 2023). This issue is also critical for data integration pipelines and data-sharing scenarios, where cryptic schema names hinder understanding, especially in legacy systems with incomplete documentation.

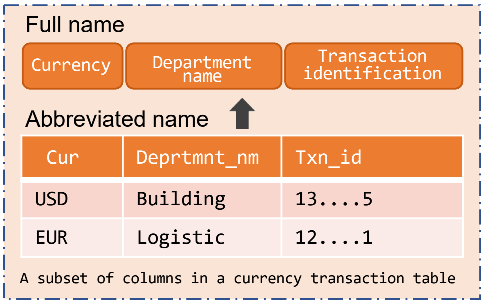

Figure 1: A Real Example of the NameGuess Task.

The NameGuess task, which expands abbreviated column names, is crucial for improving tabular data usability. It requires understanding the table context and capturing multiple abbreviation patterns. For instance, in Fig. 1, a real example from the dataset, both patterns of subsequence (Department→Deprtmnt) and non-subsequence (Transaction→Txn) exist. Humans achieve only 43.4% accuracy on the City Open Data dataset (Zhang et al., 2023), highlighting the challenge of training reliable NameGuess models.

Large language models (LLMs) show promise for NameGuess (Zhang et al., 2023; Cai et al., 2022). They understand table context and generate natural language (Sui et al., 2024). Few-shot in-context learning (Dong et al., 2022) with models like GPT-4 achieves competitive performance but is expensive and suboptimal for some schemes. Tuned moderate-size LLMs (<10B parameters) offer a cost-effective alternative, so we primarily choose this option in this paper. Current methods rely on rule-based training data generation, focus-

*Corresponding authors

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 6990–7007
November 4-9, 2025

ing mainly on subsequence abbreviations (Zhang et al., 2023; Gorman et al., 2021). However, they face two key challenges:

Challenge 1: Unrealistic training data. Real-world abbreviation conventions are complex and not fully captured by rule-based systems. Human annotation is scarce due to privacy and security concerns in tabular data. Existing methods generate abbreviations by removing characters under fixed probabilities (Zhang et al., 2023). These approaches fail to reflect real-world patterns and exclude non-subsequence abbreviations.

Challenge 2: Rule Violation in LLM Outputs. LLMs may generate invalid expansions, failing to follow subsequence rules or handle non-subsequence conventions. Guiding LLMs to adhere to rules remains a challenge.

To address the first issue, through statistics, we find that English table design follows a pattern that primarily uses subsequence abbreviations, with a few other non-subsequence abbreviations. Therefore, we develop a subsequence abbreviation generator trained on human-annotated data, capturing real-world patterns. We also propose a non-subsequence abbreviation generation method. Non-subsequence abbreviations are added to training data, covering diverse schemes like fixed expressions and language-specific methods. Our Realistic Training Data Generation (RTDG) method differs from the conventional rule-based training data generation method and generates training data closer to the real distribution.

To tackle the second challenge, we propose an automaton-based decoding system to constrain LLM output in real-time, which handles subsequence, phonetic subsequence, and fixed non-subsequence patterns. Beam search explores candidate paths, and automaton-guided pruning enforces format rules. To improve the beam search efficiency, we leverage the characteristic of this task and enforce a constraint that prevents the automaton from staying in the same state for extended periods.

In conclusion, we make the following contributions:

- Unlike the previous rule-based training set generation, we create a subsequence abbreviation generator using human-annotated data and incorporate various non-subsequence abbreviations.
- We design an automaton-based beam search decoding system for LLM output constraints and introduce idle blocking to boost search efficiency.
- We perform extensive evaluations, including more challenging cases like non-subsequence and PinYin-based abbreviations. Extensive experiments demonstrate our approach's superiority. Fine-tuning a 7/8B parameter model with our decoding system achieves similar results with the state-of-the-art models (GPT-4 series).

2 Background

This section introduces the NameGuess task, the steps for tuning LLMs to solve it, and the heuristic rules related to it.

2.1 NameGuess Task

The NameGuess task (Zhang et al., 2023) improves table readability and downstream task performance in tabular data. Formally, given a table t with N rows $\{x_1^1, ..., x_K^1\}, ..., \{x_1^N, ..., x_K^N\}$ and K column query names $q_1, ..., q_K$, the goal is to find a generator f_θ that predicts full names $p_1, ..., p_K$. Each p_i is computed as $f_\theta(p_i|q_1, ..., q_K, p_1, ..., p_{i-1}, t)$. Here, p represents full names, and q represents abbreviated names.

2.2 NameGuess through LLM

Table Context. The structured data is serialized into a task prompt during training and inference. The prompt format in (Zhang et al., 2023) is:

> Column names: $\{q_1, ..., q_K\}$ <SEP> row_1: $\{x_1^1, ..., x_K^1\}$ <SEP> ... <SEP> row_i: $\{x_1^i, ..., x_K^i\}$ <SEP> ... <SEP> row_N: $\{x_1^N, ..., x_K^N\}$, As abbreviations of column names from a table, $\{q_1, ..., q_K\}$ stand for $\{p_1, ..., p_K\}$.

Here, <SEP> is a splitting token or a newline token. Notably, we use the same table context flattened form in both LLM few-shot baselines and our prompt for fine-tuning to ensure fair comparison.

Training Data Generation. Real-world annotations of $q_1, ..., q_K$ (abbreviations) and $p_1, ..., p_K$ (full names) are limited, so previous work uses synthetic data for LLM training (Zhang et al., 2023). First, a table corpus containing both abbreviated and full names is collected. Full names are extracted to form training data since applying rules to existing abbreviations may cause inconsistencies.

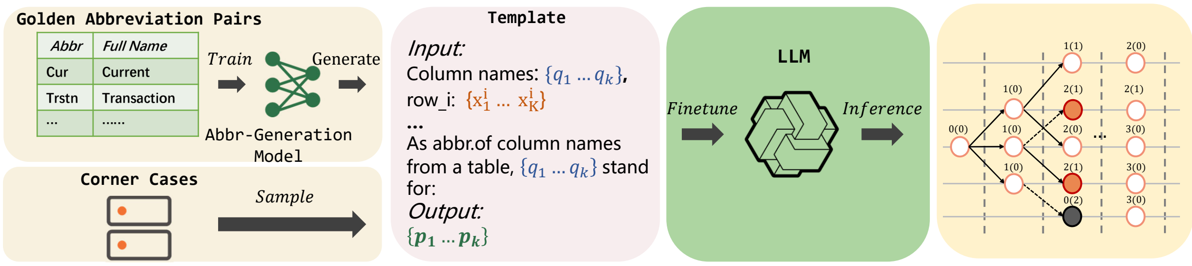

Figure 2: We tune a moderate-size LLM with LoRA for the NameGuess task. Before training, abbreviated forms of tables with full column names need to be generated. Specifically, we use a model-based abbreviation module for subsequence pattern generation and collect a lookup table to supplement corner cases. After training, the LLM recovers full column names through decoding. In decoding, a rule-enhanced automaton-based filter aids accuracy in beam search.

Next, character-removal rules generate abbreviations from full names. Rules include keeping the first characters, removing non-leading vowels and duplicate characters, and randomly removing vowels or consonants with specific probabilities.

Generative Rules vs. Discriminative Rules. Abbreviation rules can be categorized into generative and discriminative rules. Generative rules define the generator f_θ, which provides $P(q_1, ..., q_K | p_1, ..., p_K, t)$, a probability for abbreviated column names. These rules enumerate character-deleting strategies and their probabilities. Discriminative rules check if full names are valid expansions of abbreviations, where $P(p_1, ..., p_K | q_1, ..., q_K, t) > 0$.

For example, for subsequence abbreviations (Zhang et al., 2023; Cai et al., 2022), discriminative rules ensure q_i to be a subsequence of the generated full name. Generative rules include character-deleting strategies and their probabilities. Beyond the subsequence abbreviation, we consider PinYin (a widely used Chinese phonetic abbreviation), lookup table, and mixed rules. Discriminative rules can further be transformed into automata for decoding. In Appendix. A, we list all the rules related to this paper.

3 Method

As illustrated in Fig. 2, our method first optimizes the fine-tuning training set by generating subsequences with a model and collecting a table of non-subsequences (RTDG). After training, during the decoding phase of inference, we introduce an automaton to control the decoding process, ensuring that the generated outputs follow the rules (AutoBeam). The following are the specific details of these two modules.

3.1 Realistic Training Data Generation from Real-life Data

The City Open Data dataset (Zhang et al., 2023) reveals that real-world abbreviation schemes for tabular column names are predominantly subsequence-based, though a notable minority adopt non-subsequence patterns. Specifically, 93.3% of the abbreviated column names (8512 out of 9128 pairs) are subsequences of their corresponding full names after normalization, while 616 pairs (6.7%) involve abbreviations that are not subsequences of the full names.

As the introduction states, using purely synthetic subsequence-based data as training data has two major drawbacks: **1)** Real-life subsequence abbreviation patterns are not fully captured. Heuristic rules used to generate training data deviate from real-world data distributions. This reduces the quality of the trained model; **2)** While subsequence abbreviations are common, other patterns exist, making it challenging to handle a mix of mostly subsequence and some corner-case abbreviations.

To address these issues, we propose using a tuned model to capture the pattern in subsequence abbreviation. Since the non-subsequence abbreviation cases are limited and the tuned moderate-size LLM's generalization capability on these cases is poor, we don't use a unified generation model. Instead, we propose a lookup table collection method for the non-subsequence abbreviations.

Subsequence Abbreviation Generation using Tuned Model. Previous approaches rely on insufficient heuristic rules due to a lack of annotated abbreviation pairs. Real-world training data is needed for better abbreviation generation.

Similar tasks in chat language normalization have been studied, with annotated data released before, e.g., the W-NUT 2015 challenge (Baldwin et al., 2015) and the tweet normalization task (Chrupała, 2014). However, these datasets are unavailable now due to Twitter's data license. Human-annotated abbreviation pairs are available in (Gorman et al., 2021). Professional annotators removed characters from sentences sampled from English pages. This subsequence abbreviation scheme is ideal for training abbreviation generation models.

We assume the distribution of subsequence abbreviations in formal English sentences is similar to that in table column names. Single-word abbreviations are largely context-independent. Thus, we transfer the model trained on text data to generate subsequence abbreviations for tabular data. Specifically, Gorman et al.'s (Gorman et al., 2021) dataset includes sentences with abbreviated words. We collect all full names and corresponding abbreviations in this dataset. Training data is organized with a prompt, which is presented in Appendix. B. We fine-tune Llama3.1-8B to generate the possible abbreviations for an input full word. We gather all individual words in column names to generate the training set. Using the trained model, we generate possible abbreviation candidates for each word. Then, we randomly substitute words with one candidate, avoiding duplicate calculations for words in the training set.

Looup Table for Corner Case. Non-subsequence abbreviations arise from various reasons, such as symbol substitutions (replacing words with symbols, e.g., at→@), phonetically related abbreviations (based on phonetic sounds, e.g., action→axn), and convention-based abbreviations (e.g., Charles → Chuck).

Using the capabilities of LLMs, we construct a lookup table for these abbreviations. We ask a strong LLM to generate non-subsequence abbreviations providing the forming reasons and corresponding examples. The prompt used is shown in Appendix C. We use GPT-4o to generate possible non-subsequence abbreviations for each word in the table column name contexts, forming a lookup table.[1] To generate a non-subsequence abbreviation, we select a memorized term from this lookup table as the output.

Whole Process. We follow similar training set construction steps as (Zhang et al., 2023). The logical name identification and combining processes are the same. First, we use logical name identification process to extract tables with sufficient full-column names from the table corpus. Then, we collect all individual words in the training set and use a trained abbreviation generation model to create possible abbreviated forms. Finally, we apply a mixed strategy: abbreviating words using the subsequence lookup table with probability p_{sub} and the non-subsequence lookup table with probability $1 - p_{sub}$.

[1]Many cases generated by GPT-4o fail to follow the instructions and still appear to be subsequences, so we only keep the non-subsequence part.

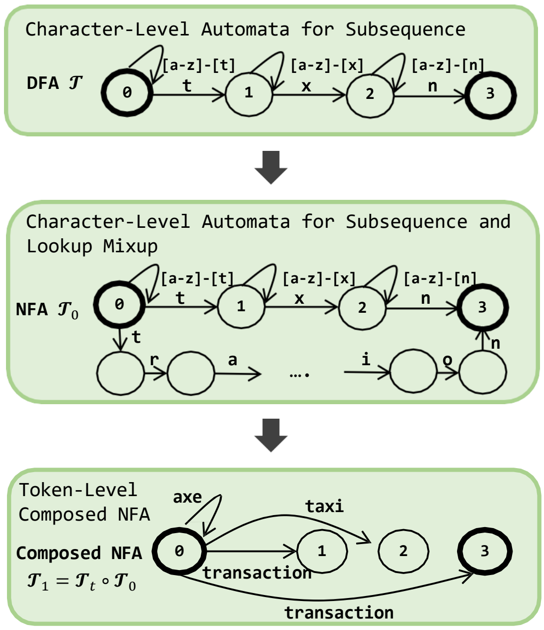

Figure 3: Example of automaton and composed transducer of the abbreviated name q "txn".

3.2 Rule-enhanced LLM Beam Search Decoding via Automata

Despite the strong capabilities of LLM and fine-tuning, LLM still suffers from the problem of hallucination (Rawte et al., 2023). Specifically, we observe that LLMs trained using data following a specific generative rule may still fail to obey the discriminative rule in inference. To solve this issue, we ensure that the LLM output follows the discriminative rules by applying constraints to the LLM's outputs.

Restraints Expressed in Automata. Our task here is to constrain the LLM's output using discriminative rules, e.g., the English abbreviation patterns defined in the previous section (subsequence pattern combined with non-subsequence patterns from the lookup table). These patterns can be expressed by regular expressions or by automata (as automata and regular expressions are equivalent). We propose using automata to represent the restrictions

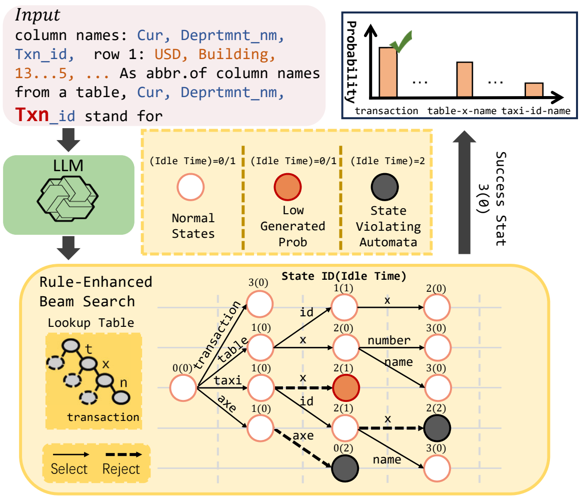

Figure 4: Example of the beam search process. The full name p is "transaction", and the abbreviated name q is "txn" .

because further traversing on these automata to express our restrictions and heuristics is relatively easier than regular expressions.

For example, the basic subsequence abbreviation rule can be expressed as a deterministic finite automaton (DFA) $\mathcal{T}$. In Fig. 3, the fundamental DFA $\mathcal{T}$ consists of the same number of tokens as the abbreviated name q. Only the corresponding character can transit to the next state on each state. For example, a state 0 accepts the first character t, and other characters return to the same state 0.

To cope with the lookup table for non-subsequence abbreviation, we define a non-deterministic finite automaton (NFA) $\mathcal{T}_0$ for the mixed lookup and subsequence abbreviation. For example, in Fig. 3, the NFA $\mathcal{T}_0$ consists of the bypass representing the lookup table. To deal with a more generalized PinYin abbreviation, we define an NFA $\mathcal{T}_{py}$ for it.

The first two automata take characters as input. Since we have to deal with the LLM's tokens as input, we define an NFA $\mathcal{T}_1$ for the mixed lookup and subsequence abbreviation that takes tokens as input. Referring to the computation result, the transitions of $\mathcal{T}_1$ have the subsequence part, where tokens traverse to the farthest covering state (transaction from s_0 to s_1), and the non-subsequence lookup part, where tokens traverse according to the abbreviated form of it in the lookup table (transaction from s_0 to s_3). We list the detailed automaton construction forms in Appendix D.

Beam Search on State machines. Beam search is implemented on state machines by maintaining a fixed number of the most promising states at each search process step. At every transition, the algorithm evaluates all possible following states. It selects the top candidates based on a scoring function (the LLM generation probability), pruning the rest to ensure computational efficiency. Take the example in Fig. 4, starting from state 0, according to our defined NFA in Fig. 3, "table" traverses to the next state, as it covers "t" in "txn" according to the subsequence rule. "transaction" can traverse to the final state, as it covers "txn" according to the non-subsequence lookup bypath. Previous work (Koo et al.) compiles the composed automaton for the constraints they are using (programming language templates, JSON format). However, this approach is impractical for our dynamic, subsequence-changing template scheme. Other widely used controlled generation libraries, e.g., Guidance [2], do not support the complex regular expressions we used in abbreviations either. Therefore, we propose our own beam search engines on these particular abbreviation-related automata (subsequence + non-subsequence lookup rules). We describe the details of the two algorithms in Appendix E. Two major modifications are stated below:

First, we leverage Trie Tree in traversing using a lookup table. During every transition, we must check whether the following sequence can fit in a non-subsequence lookup table. We construct a Trie tree to efficiently retrieve all possible full forms of an abbreviation from a lookup table by traversing paths and recording matches. For example, in Fig. 4, for the abbreviation "txn," we have to check whether "txn," "tx," or "t" has a full name in the lookup table, which can be efficiently implemented using a Trie tree. The subsequence path is easy to compute plainly by the NFA definitions.

Second, we introduce blocking on idle automata states. A naive beam search approach on the defined automata has a major drawback, called the wild-matching phenomenon (Koo et al.; Willard and Louf, 2023). In our expansion task, every token is treated as a valid input in each generation step, allowing tokens to remain idle in the same state indefinitely. This behavior can severely impact the search efficiency of the naive approach, as it leads to unnecessary and excessive exploration of redundant paths. To boost search efficiency, we propose blocking tokens to idle on a state for $threshold_{idle}$

[2]https://github.com/guidance-ai/guidance

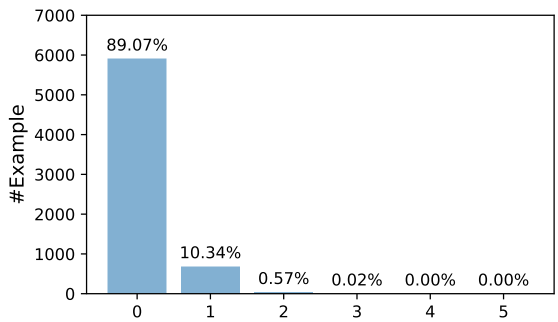

Figure 5: Maximum Number of Consecutive Idling Number in the City Open Dataset.

times. $threshold_{idle}$ is a hyperparameter that controls the number of consecutive idling numbers. According to an observation on human-annotated data presented in Fig. 5, we can conclude that in tabular column name expansion, the phenomenon of consecutive idling on a certain state is quite limited. Stucking in one state once is usually due to generating spaces or conjunctions, and 99.4% of the cases in the test set don't idle for up to 2 times in the city open dataset, which suggests that we can filter out invalid tokens through this heuristic. So we set $threshold_{idle} = 2$. Through experiments, we show that this significantly increases our search quality because wrong paths are dumped early in our approach.

Efficiency. The additional cost of our method is a small part of the original LLM inference cost. The detailed analysis is presented in Appendix. F. We also show this through experiments.

4 Experiment

In this section, we conduct extensive experiments to evaluate our proposed rule-enhanced pipeline. Our objective is to address the following research inquiries through our experiments:

- I1: How does our rule-enhanced method perform compared to the default LLM methods in the NameGuess task? How does each module (the new training set, the rule-enhanced decoding module) affect the performance?
- I2: How does our new pipeline work under different abbreviation schemes, such as the richer non-subsequence and Chinese PinYin abbreviation schemes?

4.1 Experimental Setup

Datasets. We train our model using the GitTables dataset and evaluate it on three datasets: City Open Dataset, Non-subsequence GitTables, and PinYin dataset. We show the details of the training set in Appendix. H.

Table 1: Performance on the City Open Dataset

Model	Method	EM	F1
Llama 3.1_70B	10-shot+Dynamic Exemplar	52.4	69.9
Qwen 2.5_75B	10-shot+Dynamic Exemplar	59.8	76.7
GPT_4o_mini	10-shot+Dynamic Exemplar	56.0	73.6
GPT_4o	10-shot+Dynamic Exemplar	62.4	78.0
GPT_4	10-shot+Dynamic Exemplar	63.5	79.1
Llama 3_8B	Fine-tune(Rule+GE)	56.0	73.9
Llama 3_8B	Fine-tune(Rule+Beam)	57.5	75.9
Llama 3_8B	Fine-tune(Rule+AutoBeam)	62.9	79.2
Llama 3_8B	Fine-tune(RTDG+GE)	60.6	76.5
Llama 3_8B	Fine-tune(RTDG+Beam)	59.7	76.4
Llama 3_8B	Fine-tune(RTDG+AutoBeam)	**66.1**	**81.2**
Qwen 2.5_7B	Fine-tune(Rule+GE)	53.7	71.7
Qwen 2.5_7B	Fine-tune(Rule+Beam)	54.5	72.3
Qwen 2.5_7B	Fine-tune(Rule+AutoBeam)	55.3	74.6
Qwen 2.5_7B	Fine-tune(RTDG+GE)	59.7	76.0
Qwen 2.5_7B	Fine-tune(RTDG+Beam)	60.2	76.5
Qwen 2.5_7B	Fine-tune(RTDG+AutoBeam)	<u>64.5</u>	<u>79.9</u>
Human		43.4	66.5

Evaluation Metrics. We use the metrics in (Zhang et al., 2023): exact match (EM) accuracy and F1 scores based on partial matches. The details of the metrics are described in Appendix. I.

Baselines. We compare with baselines of direct fine-tuning and LLM usage. In training, we compare training on the original dataset (**Rule**: dataset generated by heuristic rules in (Zhang et al., 2023)) with our **R**ealistic **T**raining **D**ata **G**eneration (**RTDG**) method. RTDG involves generating data using a model and substituting non-subsequence cases. During decoding, we compare **AutoBeam** (Rule-enhanced LLM **Beam** Search Decoding Via **Auto**mata) with **GE** (Regular **G**reedy **E**ncoding) and default **Beam** Search (**Beam**).

We test on multiple backbone LLMs. We mainly use Qwen 2.5 7B (Yang et al., 2024) and Llama 3 8B (Dubey et al., 2024) for fine-tuning on Chinese tasks. We also test larger GPT models (Achiam et al., 2023), Llama, and Qwen models as state-of-the-art examples. Large models are tested using in-context learning examples to demonstrate the task. Specifically, the best results we present are mainly achieved using 10-shot dynamic examples retrieved by TF-IDF (10-shot TF-IDF). We show the details of LLM evaluation in Appendix. J.

Implementation Details. We list the implementation details in Appendix. G.

4.2 NameGuess Performance

We list the NameGuess performance on the three datasets (city open dataset, non-subsequence Git-

Table 2: Performance on the Non-subsequence GitTables

Model	Method	EM	F1
Llama 3.1_70B	10-shot+Dynamic Exemplar	42.6	52.2
Qwen 2.5_75B	10-shot+Dynamic Exemplar	49.3	58.9
GPT_4o_mini	10-shot+Dynamic Exemplar	59.2	66.2
GPT_4o	10-shot+Dynamic Exemplar	58.1	63.4
GPT_4	10-shot+Dynamic Exemplar	60.0	<u>66.7</u>
Llama 3_8B	Fine-tune(Rule+GE)	50.8	57.6
Llama 3_8B	Fine-tune(Rule+Beam)	50.9	58.1
Llama 3_8B	Fine-tune(Rule+AutoBeam)	56.8	63.8
Llama 3_8B	Fine-tune(RTDG+GE)	56.4	62.3
Llama 3_8B	Fine-tune(RTDG+Beam)	56.8	63.1
Llama 3_8B	Fine-tune(RTDG+AutoBeam)	<u>60.4</u>	66.5
Qwen 2.5_7B	Fine-tune(Rule+GE)	52.3	59.0
Qwen 2.5_7B	Fine-tune(Rule+Beam)	54.9	62.0
Qwen 2.5_7B	Fine-tune(Rule+AutoBeam)	57.5	64.5
Qwen 2.5_7B	Fine-tune(RTDG+GE)	56.9	63.1
Qwen 2.5_7B	Fine-tune(RTDG+Beam)	59.4	65.8
Qwen 2.5_7B	Fine-tune(RTDG+AutoBeam)	**61.9**	**67.9**

Table 3: Performance on the PinYin Dataset

Model	Method	EM	F1
Llama 3.1_70B	10-shot+Dynamic TFIDF	27.4	37.5
Qwen 2.5_75B	10-shot+Dynamic TFIDF	46.9	58.7
GPT_4o_mini	10-shot+Dynamic TFIDF	48.3	58.2
GPT_4o	10-shot+Dynamic TFIDF	63.9	71.2
GPT_4	10-shot+Dynamic TFIDF	68.2	76.6
Llama 3_8B	Fine-tune(RTDG+GE)	71.1	79.5
Llama 3_8B	Fine-tune(RTDG+Beam)	<u>71.8</u>	80.0
Llama 3_8B	Fine-tune(RTDG+AutoBeam)	<u>71.8</u>	<u>80.3</u>
Qwen 2.5_7B	Fine-tune(RTDG+GE)	69.4	78.4
Qwen 2.5_7B	Fine-tune(RTDG+Beam)	70.5	79.3
Qwen 2.5_7B	Fine-tune(RTDG+AutoBeam)	**73.4**	**81.8**

Tables dataset, and the PinYin dataset) in Tab. 1, Tab. 2, and Tab. 3 respectively.

City Open Dataset. Several conclusions can be drawn from Tab. 1. **1)** As we can see, our best approach lies in Llama 3-8B trained on our realistic training set with model-generated abbreviations and non-subsequence lookup replacements along with a beam search decoding module guided by automaton (RTDG+AutoBeam). Compared to the state-of-the-art LLMs with larger parameters, our best results raise the EM results by 2.6%. **2)** The effect of model parameters. As mentioned in (Zhang et al., 2023), tuned models with 3B parameters (GPT2-neo) can achieve 43% accuracy, which still exists a huge gap with a tuned 7B/ 8B parameter model. Models with similar parameters perform similarly in this task. Larger models exhibit significant marginal effects on performance improvement. **3)** Supervised fine-tuning is crucial for this task. Tuned Llama 3.1 8B can perform better than a similar model with 70B parameters. Tuned models have a stronger capability of following the instructions, avoiding generating answers that can't be parsed, which is a drawback in the few-shot inference pipeline. **4)** Ablation studies. Compared to the basic beam search methods, our best approach of using the automaton-constrained beam search has an average improvement of 4.2% in EM. Also, refining the dataset brings an average of 5.3% improvement in EM on this dataset. This shows that the key components of our method are effective for solving the tabular NameGuess task.

Non-subsequence GitTables Dataset. We list three conclusions from the results of the non-subsequence GitTables dataset. **1)** Our best approach of tuning Qwen2.5-7B using the new dataset and automaton constraint achieves a 1.9% improvement in EM compared to the state-of-the-art GPT4 model. Compared to the baseline fine-tuning model, our best approach achieves an improvement of 9.6% in EM and 8.9% in F1. **2)** Compared to the City Open dataset, which has a relatively small portion of non-subsequence abbreviations, the non-subsequence GitTables dataset with more non-subsequence abbreviations is more difficult, thus having poorer performance. In contrast, our method that deals with this scenario can boost performance on this dataset. **3)** Ablation studies. Similarly, our best approach gains an average of 3.7% and 4.8% performance in EM due to the advanced dataset and decoding module, respectively.

PinYin Dataset. The PinYin dataset is another abbreviation dataset that requires understanding Chinese and its pronunciation. We draw the following conclusions: **1)** Our best approach is tuning Qwen 2.5-7B with the automaton decoding constraint, which outperforms the best state-of-the-art few-shot baseline, GPT-4, by 5.2% in EM. **2)** The few-shot larger LLMs (75B) perform poorly compared with a small Qwen model. This is partially due to the difficulty of transforming PinYin to Chinese, which is unusual in the model's training set. (In some cases, the untuned models still output in English.) To bridge this gap, supervised fine-tuning is required to help the model understand the generative rule in this scenario.

4.3 Efficiency

We present the time proportions for an average sample in Appendix K.

4.4 Case Study

We present a case study of the improvements made to the original answer. The AutoBeam system and realistic training set bring the improvements. The details are listed in Appendix. L

5 Related Work

Abbreviation Expansion. Abbreviation expansion (language normalization) is a key area in natural language processing. It is crucial across domains like SMS (Choudhury et al., 2007; Cai et al., 2022), chatrooms (Aw and Lee, 2012), and social media (Baldwin et al., 2015). In the text entry, Demasco and McCoy (Demasco and McCoy, 1992) explore abbreviation schemes. Gorman et al. (Gorman et al., 2021) investigate neural models for textual contexts. In biomedical articles, Jin et al. (Jin et al., 2019) highlight its importance, while Zhu et al. (Zhu et al., 2014) focus on clinical notes. Recently, Zhang et al. (Zhang et al., 2023) propose the NameGuess task for tabular data, showing that tabular context is key to revealing full names in column headers. Our work builds on NameGuess to generate better results in tabular data.

Machine learning techniques are applied to abbreviation expansion, from hidden Markov models to neural language models. Inspired by contextual spelling correction, the noisy channel paradigm is detailed by Brill and Moore (Brill and Moore, 2000) and used by Gorman et al. (Gorman et al., 2021) for abbreviation modeling. Recent works (Gorman et al., 2021; Cai et al., 2022; Zhang et al., 2023) leverage neural language models. With advancements in LLMs, this field continues to evolve, addressing diverse challenges.

LLM. Since 2017, pre-trained language models (PLMs) have become a research trend due to their strong performance on various tasks (Kenton and Toutanova, 2019). Recently, LLMs with significantly more parameters have shown remarkable capabilities beyond smaller PLMs (Zhao et al., 2023). Several LLMs (Achiam et al., 2023; Yang et al., 2024; Dubey et al., 2024; GLM et al., 2024) have been proposed, reshaping AI research.

LLMs can address abbreviation expansion due to their strong language understanding. Zhang et al. (Zhang et al., 2023) evaluate few-shot in-context learning using state-of-the-art LLMs (above 100B parameters) on the NameGuess task. Our work uses a moderate-size LLM (7/8B parameters), delivering outcomes on par with leading-edge, larger LLMs.

Constrained Language Model Decoding. Constrained decoding is vital in natural language processing, particularly for LLMs. These models generate outputs probabilistically, but real-world applications often require outputs adhering to specific constraints, such as structured formats or domain-specific rules. Since LLMs lack native constraint enforcement, constrained decoding techniques are needed. Hokamp and Liu (Hokamp and Liu, 2017) introduce lexically-constrained sequence decoding. Anderson et al. (Anderson et al., 2017) extend beam search with constraints for valid outputs. Recent works (Scholak et al., 2021; De Cao et al.) use trie-based lexical constraints and incremental parsing for tasks like entity disambiguation and SQL generation. Grammar-constrained decoding (Deutsch et al., 2019) ensures structural validity, and Roy et al. (Roy et al., 2022), and Stengel-Eskin et al. (Stengel-Eskin et al., 2023) show its impact on LLM performance.

A related topic is using automata for constraint implementation. Koo et al. (Koo et al.) and Willard et al. (Willard and Louf, 2023) discuss efficient automaton implementation for programming languages and JSON constraints. Our work avoids fixed templates, addressing changing subsequence patterns. We leverage NameGuess task characteristics and explore how abbreviation scheme constraints are expressed and implemented in automata. Constrained decoding ensures that generated text meets predefined criteria. In this task, we tailor criteria to specific abbreviation schemes, enabling broader applications.

6 Conclusion

In this paper, we present enhancements to LLMs' training and decoding processes to improve their performance on the NameGuess task. Our contributions include the introduction of a model-based subsequence abbreviation generation module and collecting a lookup table for handling non-subsequence abbreviations. Furthermore, we propose leveraging automata to encode discriminative rules for abbreviation expansion. We constrain the beam search process using these rules to improve decoding efficiency. Experimental results demonstrate that our approach enables fine-tuned, moderately sized LLMs with a refined decoding system to achieve performance on par with state-of-the-art models such as GPT-4.

7 Limitations

While our methods improve the NameGuess task, they do not fully exploit finer-grained table context, such as the order of columns or inter-column relationships, which could provide additional information to enhance model performance. Furthermore, our experiments primarily focus on fine-tuned small LLMs, and we have not extensively explored the potential of scaling our techniques to larger LLMs. Future work could investigate how incorporating detailed table features and tuning larger models might improve performance and generalization to more complex tabular data scenarios. For the risks of our work, deploying an immature NameGuess system may lead to incorrect predictions or mismatches, which could cause data misinterpretation or errors in downstream processes.

8 Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 62272008).

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Peter Anderson, Basura Fernando, Mark Johnson, and Stephen Gould. 2017. Guided open vocabulary image captioning with constrained beam search. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 936–945.

Aiti Aw and Lianhau Lee. 2012. Personalized normalization for a multilingual chat system. In *Proceedings of the ACL 2012 System Demonstrations*, pages 31–36.

Timothy Baldwin, Marie-Catherine De Marneffe, Bo Han, Young-Bum Kim, Alan Ritter, and Wei Xu. 2015. Shared tasks of the 2015 workshop on noisy user-generated text: Twitter lexical normalization and named entity recognition. In *Proceedings of the workshop on noisy user-generated text*, pages 126–135.

Eric Brill and Robert C Moore. 2000. An improved error model for noisy channel spelling correction. In *Proceedings of the 38th annual meeting of the association for computational linguistics*, pages 286–293.

Shanqing Cai, Subhashini Venugopalan, Katrin Tomanek, Ajit Narayanan, Meredith Morris, and Michael Brenner. 2022. Context-aware abbreviation expansion using large language models. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1261–1275.

Monojit Choudhury, Rahul Saraf, Vijit Jain, Animesh Mukherjee, Sudeshna Sarkar, and Anupam Basu. 2007. Investigation and modeling of the structure of texting language. *International Journal of Document Analysis and Recognition (IJDAR)*, 10:157–174.

Grzegorz Chrupała. 2014. Normalizing tweets with edit scripts and recurrent neural embeddings. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 680–686.

Nicola De Cao, Gautier Izacard, Sebastian Riedel, and Fabio Petroni. Autoregressive entity retrieval. In *International Conference on Learning Representations*.

Patrick W Demasco and Kathleen F McCoy. 1992. Generating text from compressed input: An intelligent interface for people with severe motor impairments. *Communications of the ACM*, 35(5):68–78.

Daniel Deutsch, Shyam Upadhyay, and Dan Roth. 2019. A general-purpose algorithm for constrained sequential inference. In *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*, pages 482–492.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, et al. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Xi Fang, Weijie Xu, Fiona Anting Tan, Jiani Zhang, Ziqing Hu, Yanjun Qi, Scott Nickleach, Diego Socolinsky, Srinivasan Sengamedu, and Christos Faloutsos. 2024. Large language models (llms) on tabular data: Prediction, generation, and understanding–a survey. *arXiv preprint arXiv:2402.17944*.

Yingqi Gao, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, Wei Li, Yuntao Hong, Zhiling Luo, et al. 2024. Xiyan-sql: A multi-generator ensemble framework for text-to-sql. *arXiv preprint arXiv:2411.08599*.

Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Dan Zhang, Diego Rojas, Guanyu Feng, Hanlin Zhao, et al. 2024. Chatglm: A family of large language models from glm-130b to glm-4 all tools. *arXiv preprint arXiv:2406.12793*.

Kyle Gorman, Christo Kirov, Brian Roark, and Richard Sproat. 2021. Structured abbreviation expansion in context. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 995–1005.

Chris Hokamp and Qun Liu. 2017. Lexically constrained decoding for sequence generation using grid beam search. In *Proceedings of the 55th Annual*

Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Association for Computational Linguistics.

Madelon Hulsebos, Çagatay Demiralp, and Paul Groth. 2023. Gittables: A large-scale corpus of relational tables. *Proceedings of the ACM on Management of Data*, 1(1):1–17.

Qiao Jin, Jinling Liu, and Xinghua Lu. 2019. Deep contextualized biomedical abbreviation expansion. In *Proceedings of the 18th BioNLP Workshop and Shared Task*, pages 88–96.

Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, pages 4171–4186.

Terry Koo, Frederick Liu, and Luheng He. Automata-based constraints for language model decoding. In *First Conference on Language Modeling*.

Christos Koutras, George Siachamis, Andra Ionescu, Kyriakos Psarakis, Jerry Brons, Marios Fragkoulis, Christoph Lofi, Angela Bonifati, and Asterios Katsifodimos. 2021. Valentine: Evaluating matching techniques for dataset discovery. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 468–479. IEEE.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

Vipula Rawte, Amit Sheth, and Amitava Das. 2023. A survey of hallucination in large foundation models. *arXiv preprint arXiv:2309.05922*.

Subhro Roy, Sam Thomson, Tongfei Chen, Richard Shin, Adam Pauls, Jason Eisner, and Benjamin Van Durme. 2022. Benchclamp: A benchmark for evaluating language models on semantic parsing. *arXiv preprint arXiv:2206.10668*.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. Picard: Parsing incrementally for constrained auto-regressive decoding from language models. *arXiv preprint arXiv:2109.05093*.

Elias Stengel-Eskin, Kyle Rawlins, and Benjamin Van Durme. 2023. Zero and few-shot semantic parsing with ambiguous inputs. *arXiv preprint arXiv:2306.00824*.

Yuan Sui, Mengyu Zhou, Mingjie Zhou, Shi Han, and Dongmei Zhang. 2024. Table meets llm: Can large language models understand structured table data? a benchmark and empirical study. In *Proceedings of the 17th ACM International Conference on Web Search and Data Mining*, pages 645–654.

Brandon T Willard and Rémi Louf. 2023. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. 2024. Qwen2 technical report. *CoRR*.

Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. Tabert: Pretraining for joint understanding of textual and tabular data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8413–8426.

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. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921.

Jiani Zhang, Zhengyuan Shen, Balasubramaniam Srinivasan, Shen Wang, Huzefa Rangwala, and George Karypis. 2023. Nameguess: Column name expansion for tabular data. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13276–13290.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223*.

Dongqing Zhu, Stephen Wu, Ben Carterette, and Hongfang Liu. 2014. Using large clinical corpora for query expansion in text-based cohort identification. *Journal of biomedical informatics*, 49:275–281.

A Examples of Generative and Corresponding Discriminative Rules in Related Abbreviation Schemes

We list the examples of generative and discriminative rules in Tab. 4.

B Prompt for Abbreviation Generation

We use the following prompt to train the model for abbreviated word generation.

> Provide several possible abbreviations for the word. Word: p; Abbreviations: $\{q_1, ..., q_k\}$

C Prompt for Corner Case Generation

We construct a prompt for word p to query its non-subsequence abbreviations:

> A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements. Please generate an abbreviation from the full name that is not the full name's subsequence. Here are some examples that may cause this phenomenon.
> #Examples:
> Symbol correlation: {ICL Examples}
> Phonetically related: {ICL Examples}
> Convention: {ICL Examples}
> Generate the possible non-subsequence abbreviation for this word: Word: p
> Abbreviation:

We present some of the examples of non-subsequence abbreviations in Tab. 5.

D Examples of Discriminative Rules in Tab. 1 Expressed in Automaton

Subsequence and Lookup Abbreviation Rules Expressed in Automaton. We mainly discuss the two discriminative rules corresponding to what we use in the training data generation: the subsequence abbreviation and the lookup table abbreviation. Suppose we are generating the full name p for the abbreviated form $q = q_1q_2...q_n$.

For the subsequence discriminative rule $p \in \{x|q\ is\ subsequence\ of\ x\}$, the regular expression of it should be $.*q_1.*q_2.*, ..., .*q_n.*$, where $.*$ matches any sequence of characters and q_i matches the character q_i. We define the DFA $\mathcal{T}$ as a 5-tuple $(S, \Sigma, \delta, s_0, F)$ as:

- $S = \{s_0, s_1, s_2, \ldots, s_n\}$ is the set of states, where each s_i represents a prefix of the string q.
- Σ is the alphabet, consisting of the distinct symbols present in the string q.
- The transition function $\delta : S \times \Sigma \rightarrow S$ is defined as: $\delta(s_i, q_{i+1}) = s_{i+1}, 0 \leq i \leq n-1$. For symbols not part of the sequence q, the DFA transitions to the same state.
- $s_0 \in S$ is the initial state, representing the empty prefix.
- $F = \{s_n\}$ is the set of accepting states, indicating that the entire string q has been successfully read.

This DFA accepts the string p if and only if the subsequence of p exactly matches q. We show an example of the abbreviated form "txn" in Fig. 4.

For the lookup table abbreviation rule $p \in \{x|q \in L(x)\}$, we can search in the reverse lookup index for q, so the output should be fixed. For example, in Fig. 4, for q="txn", p should be the word "transaction" using the lookup table.

Mixed Rule of Subsequence and Lookup Abbreviation. For a mixed discriminative rule $p \in \{x|\exists q^1, ..., q^k = q, x^1, .., x^k = x, q^j \in L(x^j)|$ $q^j\ is\ subsequence\ of\ x^j\}$, which allows part of p abbreviated by lookup table and part of p abbreviated in subsequence form. We can slightly modify $\mathcal{T}$ to $\mathcal{T}_0 = (S_0, \Sigma, \delta_0, s_0, F)$ to cope with this mixed rule as an NFA:

- S_0=S, For each $q_i, ..., q_j$, $\exists x1, .., x_k$, $L(x) = q_i, ..., q_j$, $S_0 + = \{s_i^{x_1}, s_i^{x_2}, \ldots, s_i^{x_{k-1}}\}$ is the set of states, where each s_i^x represents a state in the bypath of this possibly lookup abbreviation.
- The transition function $\delta_0 = \delta$, $\delta_0(s_i^{x_t}, x_{t+1}) = s_i^{x_{t+1}}$, $\delta_0(s_i^{x_{k-1}}, x_k) = s_j$.

Generalization of Other Abbreviation Schemes. We can generalize the mixed lookup table abbreviation to any possible abbreviation scheme in Tab. 4. Take the PinYin abbreviation scheme as an example, the abbreviated form is a subsequence of the full name's PinYin. We can represent the mixed

Table 4: Examples of generative and corresponding discriminative rules in related abbreviation schemes.

Generative Rules	Discriminative Rules	Description
randomly a non-null subsequence of p_i	$p_i \in \{x \mid q_i\ is\ subsequence\ of\ x\}$	subsequence abbreviation in (Gorman et al., 2021)
p=0.2 rule1: keep first k characters; p=0.4 rule2: removing non-leading vowels; p=0.4 rule3: removing duplicate characters,...;	$p_i \in \{x \mid \exists\ rule_a, q_i = rule_a(p_i)\}$	heuristic abbreviation in (Zhang et al., 2023)

p=0.12 rule1: delete final e;
...;
p=0.012 rule13: delete non-duplicate consonants;
p=0.073 rule14: others;
	$p_i \in \{x \mid q_i\ is\ subsequence\ of\ x\}$	statistic of abbreviations in (Gorman et al., 2021)

reserve the first character of p_i	$p_i \in \{x \mid q_i = p_i[0]\}$	optimized abbreviation for KSR in (Cai et al., 2022)
select one of the abbreviations in the lookup table L of p_i	$p_i \in \{x \mid q_i \in L(x)\}$	lookup table for corner cases
split $p_i = p_i^1, ..., p_i^k$, select one of the abbreviations in the lookup table L of p_i^j or randomly a non-null subsequence of p_i^j	$p_i \in \{x \mid \exists q_i^1, \ldots, q_i^k = q_i, x^1, \ldots, x^k = x,\ q_i^j \in L(x^j) \mid q_i^j\ is\ subsequence\ of\ x^j\}$	mix rules
randomly a subsequence of p_i's PinYin	$p_i \in \{x \mid q_i\ is\ subsequence\ of\ PinYin(x)\}$	PinYin abbreviation

Table 5: Examples and Categories of Non-subsequence Abbreviations

Category	Examples
Symbol Substitution	about->@ at->@
Phonetically Related	action->axn afford->a4d
Convention	battleship->bb charles->chuck

rule of PinYin and subsequence form as a new NFA. The new NFA $\mathcal{T}_{py} = (S_{py} = S, \Sigma_{py}, \delta_{py}, s_0, F)$ is also modified from $\mathcal{T}$:

- Σ_{py} is the Chinese character set.
- The transition function $\delta_{py} = \delta$. For each $x \in \Sigma_{py}$, q_i, $\exists max\ j, q_i, ..., q_j\ is\ subsequence\ of\ PinYin(x)$, $\delta_{py}(s_i, x) = s_j$. This represents a bypath of this possible abbreviation of the Chinese token x's PinYin.

Language Model Tokenizer as Composed Transducer. Since we are dealing with token inputs from the LLM's tokenizer instead of the character input in $\mathcal{T}, \mathcal{T}_0$, we have to model the tokenizer as well. Koo et al. propose treating the language model's vocabulary as a transducer, which has the states equal to the base DFA $\mathcal{T}$'s alphabet (Koo et al.). This allows us to composite the language model transducer with the constraint DFA/NFA. We use the same definition of the transducer $\mathcal{T}_t$ of LLM in (Koo et al.), and the composed NFA $\mathcal{T}_1 = \mathcal{T}_t \circ \mathcal{T}_0$. Due to the special form of $\mathcal{T}_0$'s definition, we can directly calculate $\mathcal{T}_1 = (S_1 = S, \Sigma_1, \delta_1, s_0, F_1 = F)$:

- $\Sigma_1 = \mathcal{V}$, where $\mathcal{V}$ is the vocabulary of the language model decoding.
- The transition function $\delta_1(s_i, v) = s_l$, $q_i, q_{i+1}, ..., q_l\ is\ subsequence\ of\ v$ and $q_i, ..., q_{l+1}$ is not subsequence of v. Or $\delta_1(s_i, v) = s_l$, $q_i, q_{i+1}, ..., q_l = L(v)$, v is a token in Σ_1.

Intuitively, one of the tokens v from the vocabulary $\mathcal{V}$ can traverse as far as it can to cover part of q as its subsequence, or it can cover part of q as a lookup value and itself as a lookup key. For example, in Fig. 4, starting from the third state, "taxi" covers "tx" in "txn", so it traverses to the second state. "axe" covers none in "txn", so it stays the first state. "transaction" is a key in the lookup table, and its value is "txn", so it can directly traverse to the fourth state.

For the generalized cases, such as the mixed rules of PinYin and the subsequence abbreviation, the composed NFA with the transducer also has a similar form. We use the same definition of the transducer $\mathcal{T}_t$ of LLM in (Koo et al.), and the composed NFA $\mathcal{T}_{py1} = \mathcal{T}_t \circ \mathcal{T}_{py} = (S_{py1} = S, \Sigma_{py1}, \delta_{py1}, s_0, F_1 = F)$:

- Σ_{py1} is the Chinese vocabulary of the language model decoding.
- The transition function $\delta_{py1}(s_i, v) = s_l$, $q_i, q_{i+1}, ..., q_l$ is subsequence of v and $q_i, ..., q_{l+1}$ is not subsequence of v. $q_i, q_{i+1}, ..., q_l$ is subsequence of $PinYin(v)$ and $q_i, ..., q_{l+1}$ is not subsequence of $PinYin(v)$, where v is a token in Σ_{py1}.

This is similar to traversing to the farthest state, however an additional PinYin transition is required.

E Algorithm of State Traverse Computation and Constrained Beam Search via Automata

We describe the algorithms in detail in this section. The proposed algorithms, State Traverse and Beam Search, are designed to tackle the problem of abbreviation expansion using a combination of finite automata and language models. In Algorithm 1, the State Traverse algorithm initializes by constructing a Trie Tree from a lookup table, which serves as a reference for valid expansions. A Trie Tree is used to efficiently search whether any of $q_{l+1}, ..., q_n$'s prefixes have a corresponding full name. Specifically, $check(T_L, (q_{l+1}, ...q_n))$ means that we traverse from the root to the state of $q_l, ..., q_n$, and we record all the possible lookup abbreviations on the path to form the output of $check(T_L, (q_{l+1}, ...q_n))$. For example, $q_{l+1}, ..., q_n = txn$, the possible by path at this stage consists of all the full names that have the abbreviated form in "txn", "tx" and "t". So, by traversing the path through "txn" to the root, we can tract all the possible full names in the lookup table.

Algorithm 2, the Beam Search algorithm, utilizes the State Traverse algorithm to iteratively explore possible expansions of an abbreviation. Starting from an initial state, it maintains a buffer of candidate expansions, each associated with a probability score and a wait counter to prevent stalling on non-progressive states. After Alg. 1 calculates the set of traversing states using the NFA we defined, for each arrival state $q_{arrival}$ determined by the transition function, the algorithm updates the new state and probability, and appends the new candidate to the buffer if they are below a predefined idling threshold. The generation quality of each candidate is valued by generation probability calculated using the LLM, and the buffer is sorted according to the probability after each iteration. This process continues until the buffer is exhausted, ensuring a breadth-first search of potential expansions while adhering to the constraints imposed by the finite automaton and language model. The combination of these algorithms provides a robust framework for accurately expanding abbreviations in a structured and efficient manner.

F Efficiency Analysis

Different automata built for different abbreviation schemes may have different running complexities, we will take the mixed rule of subsequence abbreviation and lookup abbreviation as an example here. The additional cost of our proposed filter compared to the traditional beam search is the cost of checking the lookup table and the transition functions. **Lookup Table.** In our implementation, the lookup rule in the beam search part is implemented as a prefix tree. In Alg. 1, where we need to check whether a generated token is a prefix of the full name of a potential prefix of $(q_{l+1}, ..., q_n)$. This requires a query in the prefix tree of $q_{l+1}, ..., q_n$. The complexity of querying $q_{l+1}, ..., q_n$ in a prefix tree is $O(l_q)$ in the worst case, where l_q is the length of $q_{l+1}, ..., q_n$. **Transition Function Check.** In our implementation, we conduct the transition function check on the run. For each token v to be checked, we traverse according to the transition rules composed by the rule NFA and the token transducer DFA. The complexity of such a transition is $O(l_v)$, where l_v is the length of the token v. **Overall Complexity.** Suppose that the Beam Width is B, which refers to the number of candidate sequences retained at each step, and the maximum length of the generated sequence in tokens is T. The additional overall complexity of the $B * T * (l_q + l_v)$, which is a small part of the whole language model inferencing cost. Notably, l_q, l_v is a small number regardless of how large the lookup table is, which promises a low additional cost for our method.

G Implementation Details

We use Huggingface's Transformers (Wolf et al., 2019) library to implement the LLMs, we leverage the TRL library and PEFT library to conduct Lora fine-tuning on the LLMs, and we apply the vLLM (Kwon et al., 2023) library to generate sequences from the LLMs more efficiently. The fine-tuning and inference of GPT models are implemented through the OPENAI official API using the default hyper-parameters. Following the conventions in LLM fine-tuning, we train our model using the AdamW optimizer (Loshchilov and Hutter, 2019). The training set mix hyperparameter p_{sub} is set to 0.5. The number of training epochs is set to 3, the learning rate is set to $2e-5$, and the batch size is set to 4. The lora configs are $lora\_alpha = 16, lora\_dropout = 0.1$, and $lora\_rank = 8$. The prompt template we used is in Appendix C. In all experiments regarding beam search, we use a beam width of 10 and a maximum sampling token of 50 to ensure fair comparison. We report the mean result of three times

Algorithm 1: State Traverse

```
Class State_Traverse:
    Function initialize(lookup table L):
        Build a Trie Tree T_L for values in L;
    Function check( Trie Tree T_L, input q = q_1, .., q_n):
        Traverse on T_L from root to q;
        Collect Full name x and Abbreviation q_{l+1}, ..., q_t to S_output on the path;
        return S_output;
    Function run(NFA 𝒯_1 = (Q_1 = Q, Σ_1, δ_1, q_0, F_1 = F), input q = q_1, ..., q_n, language
      model f, lookup table L, input state s_l, current full name p_1, ..., p_m, beam search sampling
      number k):
        𝒱_k ← Top(f(p_{m+1}|p1, ..., p_m, q), k);
        𝒱_valid ← {};
        {x, t|x = L(q_{l+1}, ..., q_t)} ← check(T_L, (q_{l+1}, ..., q_n));
        for each v ∈ 𝒱_k do
            if v, t ∈ {x, t|x = L(q_{l+1}, ..., q_t)} then
                𝒱_valid.add((v, s_t));
            Use v to traverse on 𝒯_1 from s_l to s_u;
            𝒱_valid.add((v, s_u));
        return 𝒱_valid;
```

experiment. The experiments are conducted on an Ubuntu 20.04.6 with an Intel Xeon Silver 4210R CPU and 2 NVIDIA A6000 graphics cards.

H Datasets

We show the statistics of the training set in Tab. 6. We train our models mainly based on the GitTables dataset (Hulsebos et al., 2023). We clean up (filter tables with no column names, tables containing above a half of null values, and tables with few rows and columns) the original GitTables dataset to remove its noisy part. We generate the abbreviation pairs using our proposed method. The combining pattern of the generated abbreviation pairs is the same as that in (Zhang et al., 2023).

We train the abbreviation generation model with the training set extracted from Gorman et al.'s (Gorman et al., 2021) expert annotated wiki sentence dataset on a Llama3-8B model. We follow the construction way in Sec. 3.1. We collect a lookup table, especially for the non-subsequence abbreviations in English. We also follow the construction way in Sec. 3.1 We evaluate our method and the baseline methods on mainly three datasets.

City Open Dataset (Zhang et al., 2023). Zhang et al. collected the City Open dataset from city government tables from New York (NYC), Chicago (CHI), San Francisco (SF), and Los Angeles (LA), covering multiple categories, such as business, education, environment, health, art, and culture. Human annotators are assigned to recover the abbreviated column names and generate new abbreviated forms from full names on these tables. A further quality audit is conducted to enhance the validity of this dataset. The table corpus of this dataset is the whole GitTables dataset.

Non-subsequence GitTables. After the word segmentation of the column names (each column name may be separated into multiple words), we select the tables containing potential full names that can be abbreviated into non-subsequence forms. The words having an acronym in the lookup table are transformed using the lookup table with 0.8 probability, and the rest of the words are transformed using the rules in (Zhang et al., 2023). We split the original GitTables dataset to form the training set and the testing set. The data construction process is the same in both sets. We construct this dataset to show that our training method can further boost performance on different abbreviation schemes and training on the non-subsequence forms can actually generalize to other non-subsequence cases.

PinYin dataset. The PinYin scheme is relatively difficult because it's rare in the LLM's training

Algorithm 2: Constrained Beam Search via Automata

Input: NFA $\mathcal{T}_1 = (Q_1 = Q, \Sigma_1, \delta_1, s_0, F_1 = F)$, input $q = q_1, \ldots, q_n$, language model f, lookup table L, beam search sampling topk k, idling threshold th_{id}, beam width w

Output: Output full name p

```
ST ← State_Traverse();
ST.initialize(lookup table = L);
buf ←[(s_state = s_0, wait = 0, prob = 0, cname = "")];
success ← [];
while buf is not empty do
    (s_state, wait, prob, cname) ← buf.pop();
    V_valid ← ST.run(input = q, current full name = cname, s_l = s_state, NFA = T_1,
      language model = f, beam search sampling number = k);
    if s_state in F_1 then
        success.append((s_state, wait, prob, cname));
        continue;
    for each v, δ_1(s_state, v) ∈ V_valid do
        for s_arrival ∈ δ_1(s_state, v) do
            if s_arrival = s_state then
                new_wait ← wait + 1;
            else
                new_wait ← 0 ;
            if new_wait < th_id then
                buf.append((s_arrival, new_wait, prob + f(v, cname|q), cname+v));
    Sort buf by prob in descending order;
    buf ← buf[:w];
Sort success by prob in descending order;
return success[0].cname;
```

Table 6: Statistics of the used datasets.

Developing Dataset	#Example	#Avg. Col	#Avg. Row
GitTables	163,204	19.5	93
Gorman's Wiki	11,511	/	/
Non-subsequence Lookup	2,473	/	/
Training set_City	79,551	4.6	61
Training set_nonsub	59,492	4.0	47
Training set_PinYin	49,211	3.8	45
Evaluating Dataset	**#Example**	**#Avg. Col**	**#Avg. Row**
City Open_SF	4,781	23.9	643
City Open_CHI	3,975	21.1	605
City Open_LA	462	21.3	578
GitTables_nonsub	19,668	8.5	87
PinYin	14,054	7.1	67

corpus. We transform the GitTables dataset into Chinese and the corresponding PinYin to form this dataset. The English table contents are preserved, and the column names are either kept in English or transformed into their PinYin form in Chinese. We set the probability of keeping and transforming to 0.5 and 0.5, respectively. We split the original GitTables dataset to form the training set and the testing set. The training set is constructed using the same rules. We construct this dataset to show that our proposed pipeline can cope with different abbreviation schemes.

I Evaluation Metric

EM checks if the predicted column name matches the ground truth after normalization, ignoring case, punctuation, and articles. The F1 score measures token overlap between predictions and ground truth, calculated as $2 \cdot \text{precision} \cdot \text{recall}/(\text{precision} + \text{recall})$. Precision is the proportion of correct tokens among predictions, and recall is the proportion of correct tokens in the ground truth. This metric balances accuracy and completeness, capturing partial matches.

Table 7: Case study of improved examples in the three datasets.

Dataset	Ans (Rule+GE)	Ans (RTDG+AutoBeam)	Abbreviation
City Open	["row_id", "BasePay", "employment_type", "job_class", "lump_sum_pay", "other_pay_payerroll_tax", "overtime_pay","pay_grade", "job_class_link", "avg_boss_life"	["ROW ID", "Base Pay", "Employment Type", "JOB CLASS", "LUMP SUM PAY", "Other Pay Payroll Explorer", "Overtime Pay", "Pay Grade", "job class link", "Average Basic Life"]	["rowId", "BsePay", "employmnt_typ", "job_cls", "lump_sm_pay", " othr_pay_payrll_expl", "ovrtm_pay", "pay_grd", "job_cls_lnk", "avg_bsc_life"]
Non-subsequence GitTables	["time", "attenuation", "dispersion", "omegaXvolume"]	["time", "attenuation", "dispersion", "omega_times_volume"]	["time", "atten", "dssn", "omegXVlm"]
PinYin	["名称", "生物体", "已知作用", "位置", "父化合物"]	["名称", "生物体", "已知作用", "位置", "父关键字"]	["MingCheng", "ShengWuTi", "YiZhiuoYong", "WeiZhi", "FuGuanJZ"]

J Demonstration Examples for LLM Baseline

Difficulty in Zero-shot settings. In the flattened form used in our paper, one table's recovery task is organized in one prompt. The output format is restricted to full names separated by |. In our test, most outputs fail to follow our desired format, and the parsing results are chaotic. Therefore, we conclude that zero-shot is not suitable for this task.

Static Exemplar. For the city open dataset, we use the following examples: e.g., "As abbreviations of column names from a table, c_name | pCd | dt stand for Customer Name | Product Code | Date." For the GitTables_PinYin dataset, we use: e.g., "column names: JiLu, JiYin, SWT, row 1: P50402, EMD, Human, row 2: Q9Y6D9, MAD1L1, Human. As abbreviations of column names from a table, 'JiLu| JiYin| SWT' stands for '记录| 基因| 生物体'." (Full column names are Chinese, and abbreviations are subsequences of the full names.)

Dynamic Exemplar. To show the full potential of LLM's few-shot capability on this task, we present the results of a dynamic exemplar. For the TF-IDF settings, we dynamically choose the most related examples from the training set in terms of the TF-IDF metrics and add them as examples. For the random settings, we randomly select one example from the training set as the ICL example. These approaches offer more similar full-abbreviation pairs as examples, which boosts LLM's performance.

Table 8: Performance on the City Open Dataset

Model	Method	EM	F1
Llama 3.1_70B	1-shot Static	43.3	61.8
Qwen 2.5_75B	1-shot Static	50.4	65.2
GPT_4o_mini	1-shot Static	52.9	72.0
GPT_4o	1-shot Static	55.6	72.2
GPT_4	1-shot Static	57.0	73.4
GPT_4o_mini	1-shot Random	45.2	63.5
GPT_4o	1-shot Random	44.2	59.0
GPT_4	1-shot Random	52.2	68.0
GPT_4o_mini	1-shot TF-IDF	48.6	66.2
GPT_4o	1-shot TF-IDF	51.6	67.5
GPT_4	1-shot TF-IDF	54.7	70.9
GPT_4o_mini	5-shot Random	55.5	73.2
GPT_4o	5-shot Random	61.6	77.3
GPT_4	5-shot Random	63.2	79.1
GPT_4o_mini	5-shot TF-IDF	54.3	71.9
GPT_4o	5-shot TF-IDF	60.3	76.2
GPT_4	5-shot TF-IDF	62.5	78.3
GPT_4o_mini	10-shot Random	56.0	73.6
GPT_4o	10-shot Random	62.4	78.0
GPT_4	10-shot Random	63.0	79.1
GPT_4o_mini	10-shot TF-IDF	53.9	72.7
GPT_4o	10-shot TF-IDF	61.5	77.4
GPT_4	10-shot TF-IDF	63.5	79.1
GPT_4o_mini	zero-shot	0	3.5
GPT_4o	zero-shot	0	4.6
GPT_4	zero-shot	20.7	31.9

K Efficiency Experiment

We present the time proportions in the whole end-to-end inference time for an average sample in Fig. 6. We select Qwen 2.5-7B for test in this subsection. The data compares the time spent on two parts, LM Reasoning (original beam search cost) and Rule Judgment (additional cost brought by the automata constraints in beam search), across three different datasets: City Data, Non-subsequence GitTables, and PinYin. For the City Data and Non-subsequence GitTables dataset, the time spent on LM Reasoning is significantly higher than the time spent on Rule Judgment. Specifically, for City Data, LM Reasoning accounts for 99.0% of the total time, while Rule Judgment takes up only 1%. For Non-subsequence GitTables, the proportions are similar. However, for PinYin, LM Reasoning takes only 20.5% of the time, with Rule Judgment making up the remaining 79.5%. This is due to the high cost of converting the Chinese tokens to

Table 9: Performance on the Non-subsequence Dataset.

Model	Method	EM	F1
Llama 3.1_70B	1-shot static	49.9	60.2
Qwen 2.5_75B	1-shot static	46.4	56.4
GPT_4o_mini	1-shot static	35.8	43.9
GPT_4o	1-shot static	35.1	42.5
GPT_4	1-shot static	54.5	65.8
GPT_4o_mini	1-shot Random	38.8	48.8
GPT_4o	1-shot Random	38.6	45.9
GPT_4	1-shot Random	43.8	52.2
GPT_4o_mini	1-shot TF-IDF	47.2	55.6
GPT_4o	1-shot TF-IDF	45.9	52.8
GPT_4	1-shot TF-IDF	49.1	56.3
GPT_4o_mini	5-shot Random	51.0	60.4
GPT_4o	5-shot Random	54.0	61.3
GPT_4	5-shot Random	52.9	61.5
GPT_4o_mini	5-shot TF-IDF	56.1	63.3
GPT_4o	5-shot TF-IDF	58.1	63.8
GPT_4	5-shot TF-IDF	60.1	66.7
GPT_4o_mini	10-shot Random	50.8	60.1
GPT_4o	10-shot Random	58.0	65.4
GPT_4	10-shot Random	55.5	63.5
GPT_4o_mini	10-shot TF-IDF	59.2	66.2
GPT_4o	10-shot TF-IDF	58.1	63.4
GPT_4	10-shot TF-IDF	60.0	66.7
GPT_4o_mini	zero-shot	0	1.7
GPT_4o	zero-shot	0.2	2.3
GPT_4	zero-shot	21.2	27.9

Table 10: Performance on the PINYIN Dataset

Model	Method	EM	F1
Llama 3.1_70B	1-shot static	29.2	36.3
Qwen 2.5_75B	1-shot static	40.6	51.6
GPT_4o_mini	1-shot static	31.1	42.2
GPT_4o	1-shot static	43.6	52.2
GPT_4	1-shot static	52.6	63.5
GPT_4o_mini	1-shot Random	32.1	42.5
GPT_4o	1-shot Random	42.6	51.5
GPT_4	1-shot Random	43.3	54.1
GPT_4o_mini	1-shot TF-IDF	43.3	54.3
GPT_4o	1-shot TF-IDF	48.9	57.3
GPT_4	1-shot TF-IDF	54.9	64.4
GPT_4o_mini	5-shot Random	35.1	46.0
GPT_4o	5-shot Random	55.7	65.5
GPT_4	5-shot Random	56.1	67.3
GPT_4o_mini	5-shot TF-IDF	46.9	56.7
GPT_4o	5-shot TF-IDF	61.1	68.9
GPT_4	5-shot TF-IDF	66.1	74.9
GPT_4o_mini	10-shot Random	35.6	46.3
GPT_4o	10-shot Random	56.1	65.8
GPT_4	10-shot Random	56.8	67.5
GPT_4o_mini	10-shot TF-IDF	48.3	58.2
GPT_4o	10-shot TF-IDF	63.9	71.2
GPT_4	10-shot TF-IDF	68.2	76.6
GPT_4o_mini	zero-shot	0	0.6
GPT_4o	zero-shot	0	0.6
GPT_4	zero-shot	0.2	1.7

PinYin.

These percentages suggest that LM Reasoning is a more time-consuming process compared to Rule Judgment while we are using the mixed rules of subsequence and lookup rules, which is the same as we have analyzed in Sec. 3.2 regardless of the dataset being processed. The time cost of rules is much higher in the PinYin dataset, as the transitions of tokens to their pronunciation are not straightforward, thus costing more time in rule judgment.

L Case Study

In three distinct dataset case studies, we observe improvements made to the original answer (Ans(Rule+GE)) to provide the correct field names in our best answer (Ans(RTDG+AutoBeam)). (The original answer is from the original training set with greedy encoding, and the optimized answer is from the new realistic training set with automaton constraints.) Firstly, in the "City Open" dataset, the original answer contained field names such as "other_pay_payerroll_tax" and "avg_boss_life," which are clearly hallucinations from the LLM. The first abbreviation contains multiple words, thus making it hard to generate the correct answer, while the second abbreviation may be distracted from the job context so that it generates the word "boss". Both errors violate the subsequence constraints ("other_pay_payerroll_tax" $\leftrightarrow$ "othr_pay_paryrll_expl", and "avg_boss_life" $\leftrightarrow$ "avg_bsc_life"). The optimized answer (New Ans) corrected these field names to "Other Pay Payroll Explorer" and "Average Basic Life," making the abbreviated form a subsequence of the generated full names.

Secondly, in the "Non-subsequence GitTables" dataset, the original answer included a field name "omegaXvolume", which could be confusing as it didn't clearly express the relationship between "omega" and "volume." The optimized answer corrected this to "omega_times_volume," clarifying the multiplicative relationship between the two con-

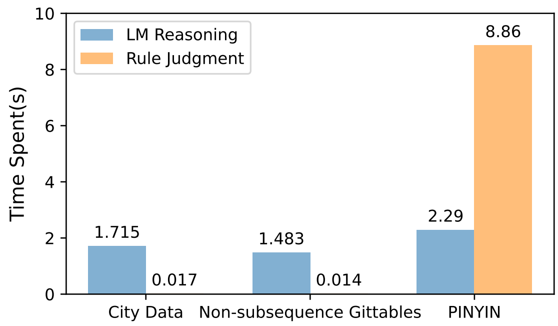

Figure 6: Time comparison of the LM running time and additional constraint running time.

cepts. This is corrected due to the "times" ↔ "X" relationship in the lookup table, and through training on such datasets with non-subsequence pairs, the model values “times” over "X" to make the prediction correct.

Lastly, in the "PinYin" dataset, the original answer had a field name "父化合物" (Father Compound, Pronunciation: FuHuaHeWu), which is distracted by the biochemistry context of this table and violates the subsequence rule of Chinese PinYin. ("FuHuaHeWu" ↔ "FuGuanJZ") The optimized answer changed this to "父关键字" (Father Keyword, Pronunciation: FuGuanJianZi), which satisfies the constraints and appears to match with the ground truth.

These case studies demonstrate that by adopting our methods, we can significantly enhance the readability and usability of data, thereby facilitating the data analysis and processing process.

M End-2-End Tabular Task Performance.

To illustrate the realistic effect of our method's performance on end-2-end tabular tasks (Fang et al., 2024), though not the key topic of this work, we conduct experiments on two tasks. The first task is TextSql. We use the BIRD benchmark, and we use the XiYAN-7B model (the state-of-the-art 7B model on this dataset) as the test model for SQL generation (Gao et al., 2024). We test on the simplest settings, only generation and no other tricks. We do not include the description information for each column in the dataset because the descriptions contain direct information on the full name of the column. We generate abbreviations using the same process in this paper and produce the results. The second task is the schema matching task. We test on the GitTables dataset, and we aim to match two tables, the original table and the table with abbreviated column names. The results are shown in Tab. 11.

Both experiments show that abbreviated column names can harm the performance of tabular tasks, and recovering the column names using our proposed pipeline can largely relieve this issue.

Condition	Text2SQL	Schema Matching
Original Data	0.56	1.0
Simplified Column Names	0.49	0.84
Recovered Column Names	0.55	0.89

Table 11: Results of Text2SQL and Schema Matching under Different Data Conditions