Asking language models how to represent data for fine-tuning

Anonymous ACL submission

Abstract

Language models are often used for tasks involving structured data like tables and graphs, but there is no general approach for choosing the best format to represent such data across different tasks for fine-tuning. In this study, we show how the pre-trained model can suggest its own formats for representing structured data in a general task We also compare the performance of different formats after fine-tuning the models to see how they relate to the pre-trained performance. Our results show that different formats perform best across different models after fine-tuning for the same task. Interestingly, the format that performs best before fine-tuning always remains one of the top choices afterwards. This approach can help avoid the need for trial-and-error during fine-tuning, saving time, computational resources, and reducing the environmental impact of training large models.

1 Introduction

When using language models on tasks that involve structured data, such as tables or graphs, an important decision is how to represent this data in a textual format. Previous work has highlighted the importance of choosing a good representation when prompting large language models (LLMs) for tasks that involve tables (Singha et al., 2023; Ye et al., 2023b) and graphs (Guo et al., 2023). In this paper, we explore three research questions about formatting structured data for fine-tuning small language models (SLMs).

To answer these questions, we explore three types of structured data (tabular data-frames, database schemas, and graphs) using three small language models (Mistral, Phi-3, and CodeLlama). For tabular data, we focus on two text-to-code tasks: Excel formulas (Singh et al., 2024) and Python functions (Yin et al., 2022). For database schemas, we analyze the text-to-SQL dataset (Yu et al., 2018), and for graph tasks, we use graph question-answering data from Wang et al. (2024a).

First, we evaluate the extent to which the choice of format still matters for fine-tuning. Out of two hypotheses—either the model learns to use any format during fine-tuning or the model learns more efficiently with a specific format—we show that the latter one holds consistently across different tasks and models. The importance of choosing a good representation thus remains for fine-tuning.

Second, we evaluate whether there is a correlation in the performance before and after fine-tuning using a specific format. If this holds—and we show that it does in 16 out of 18 settings—the decision about which format to use can be made using inference only, sparing the developer an expensive trial-and-error approach.

Third, we evaluate whether the small language model can implicitly suggest candidate formats to evaluate. If it can—which it does—a developer is less reliant on their own experience with structured formats, and thus less likely to *forget* validating a specific format.

To summarize, our key contributions are as follows:

- We show that small language models prefer specific textual representations of structured data over others, even when fine-tuning.
- We demonstrate that performance after fine-tuning closely aligns with the base model performance across different formats, reducing the need for extensive fine-tuning experiments. This finding saves significant time and computational resources.
- We show that small language models can provide their own suggestions for data formats, offering a practical way to identify suitable formats for structured data tasks.

2 Experimental setup

We study this approach across three distinct data structures: tables, database schemas, and graphs, using three open-source language models for fine-tuning: Phi-3, Mistral and CodeLlama.

2.1 Datasets

We consider three commonly used structures on four datasets (two datasets for tables) following prior studies on structured data understanding for LLMs (Jiang et al., 2023; Zhuang et al., 2024).

Tables Tables are a common structure in text-to-code tasks. We focus on two such tasks: generating Python code and Excel formulas from natural language. For **Python**, we use the Arcade dataset (Yin et al., 2022) involves generating code that uses the `pandas` library to manipulate dataframes, selecting all 208 that we can execute for testing and synthetically generating 4836 training and 1210 validation samples using `gpt-4-turbo` following Singha et al. (2024). For **Excel** formulas, we use the 5668 validated training samples (5246 train and 422 validation) and 200 test cases from Singh et al. (2024).

Database schemas Text-to-**SQL** is another text-to-code task, but input is a database schema instead of a whole table. We use the Spider dataset (Yu et al., 2018) of 1032 tests and split the training set into 5509 training and 1482 validation examples.

Graphs Whereas tables have an intuitive textual representation for language models, like CSV or column-oriented JSON, graphs are less straightforward. Following an exploration of the ability of LLMs to solve graph problems in natural language (Wang et al., 2024a) we consider the tasks of **cycle detection** (graph $\rightarrow$ bool), **flow estimation** (graph $\rightarrow$ float) and finding the **shortest path** (graph $\rightarrow$ path). There are 100 tests for each task. A single model is trained across all tasks on a training and validation set of consist of 6500 and 500 samples respectively, uniformly divided over all tasks.

2.2 Fine-tuning setup

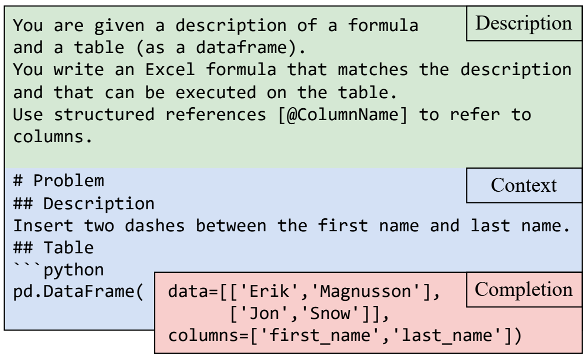
We fine-tuned all models: `Phi-3-mini-4k-instruct` (3.8B parameters), `Mistral-7B-Instruct-v0.2` (7B parameters), and `CodeLlama-7b-hf` (7B parameters) using low-rank adaptation (LoRA) (Hu et al., 2021) for 10 epochs. The best checkpoint was determined by selecting the one with the lowest validation loss. All experiments were conducted on a single A100 GPU. For all models, we use a batch size of 8, optimizer as `adamw_torch` and weight decay of 0.001. For LoRA configuration, we set the rank to 64, alpha parameter to 16 and dropout to 0.1. The learning rate for Mistral and CodeLlama was set to 2e-4 and for Phi-2 it was set to 1e-4. These settings are based on commonly used configurations in similar fine-tuning setups (Hu et al., 2021). These values have been validated in our preliminary experiments.

2.3 Evaluation metrics

For all the code generation tasks, we use the pass@k (Chen et al., 2021) metric based on execution match of code, which estimates the probability that at least one out of k generations passes all provided tests. We compute pass@5 over 10 predictions at temperature 0.6. A temperature of 0.6 was chosen to balance diversity and quality in generations during fine-tuning, like prior works in LLMs (Chen et al., 2021).

Similarly, for flow estimation and shortest path, we use exact match with the pass@5 metric, based on 10 predictions at a temperature of 0.6. For cycle detection, which requires generating a binary response (*true* or *false*) we use exact match for a single prediction at zero temperature because using pass@k for higher k gives inflated results.

3 Getting formatting suggestions



~~~
You are given a description of a formula
and a table (as a dataframe).
You write an Excel formula that matches the description
and that can be executed on the table.
Use structured references [@ColumnName] to refer to
columns.

# Problem
## Description
Insert two dashes between the first name and last name.
## Table
```python
pd.DataFrame( data=[['Erik','Magnusson'],
                    ['Jon','Snow']],
              columns=['first_name','last_name'])
~~~

Figure 1: An example of incomplete prompt and its completion used for generating formatting suggestions.

We can leverage the pre-trained model to suggest format for data structures by providing a partial prompt to the model, letting it auto-complete the data structure and then parsing the format. A typical fine-tuning prompt includes a task description (like writing formulas from natural language) and some context on the problem instance (like the natural language utterance and a table). We structure this prompt to ensure that the data structure is the last part of the context, cut the prompt short right before the data structure, and let the model auto-complete the structure. An example for NL2F is shown in Figure 1.

We generate 10 predictions for each instance at a temperature of 0.8 for each problem and analyze the results with regular expressions. In the following two sections, we respectively analyze the discovered formats and how their occurrence statistics correlate to fine-tuning performance.

3.1 Suggested formats

An overview of occurrence statistics is shown in Table 1 and detailed in the following paragraphs.

Dataframe tables We prefix the format with a `pd.DataFrame` constructor to encourage the model towards more variety, as the default mode is to suggest markdown. We find the following formats

ⓐ **Record**: a list of row dictionaries, where each row maps a column name to a single value.

ⓑ **Column**: a dictionary with each column name mapped to a list of of its values.

ⓒ **Row**: a list of column names followed by a list of values for each row.

ⓓ **Row-invert**: similar to Row, but with column names listed after the row values.

and illustrate them in 2. Tables 1a and 1b show that the column format is suggested significantly more often than others for all models. Interestingly, the second format differs across models and tasks. CodeLlama has the most diversity, suggesting each format more than 1.5% of completions.

Database schemas The most common formats generated by the models are

ⓐ **SQL code**: the representation resembles SQL code for creating tables, with column names and data types enclosed within the statement.

ⓑ **Open column**: a natural listing of table name, a colon (:) and a list of column names. Unlike the closed bracket format, column names can be placed on new lines.

ⓒ **Closed bracket**: tables are represented with column names enclosed in parentheses, similar to function parameters.

Table 1: Occurrence statistics of formats suggested by different models across all tasks.

Format	Mistral	Phi-3	CodeLlama
Column	85.60	85.40	69.70
Record	0.35	5.30	4.40
Row	6.55	0.15	2.50
Row-invert	6.15	5.40	21.65
Others	1.35	3.75	1.75

(a) Formula

Format	Mistral	Phi-3	CodeLlama
Column	80.53	91.92	73.27
Record	0.72	7.69	4.90
Row	3.51	0.05	2.93
Row-invert	13.61	0.24	15.48
Others	1.63	0.10	3.41

(b) Python

Format	Mistral	Phi-3	CodeLlama
Closed bracket	50.49	66.4	33.32
SQL code	26.39	16.10	29.57
Column list	22.35	13.3	21.43
Markdown	0.14	1.50	9.76
Others	0.61	3.48	5.92

(c) SQL

Task	Format	Mistral	Phi-3	CodeLlama
CD	**Adj. dict**	40.61	10.00	19.80
	Adj. matrix	0.20	13.00	10.10
	Edge list	58.99	31.52	21.31
	NL Graph	0.05	11.11	4.85
	Others	0.15	34.37	43.94
FE	**Adj. dict**	27.90	12.70	17.40
	Adj. matrix	3.00	4.00	15.80
	Edge list	67.10	80.40	28.10
	NL Graph	0.03	2.70	8.30
	Others	2.00	0.02	30.40
SP	**Adj. dict**	74.8	45.90	29.20
	Adj. matrix	0.60	0.60	8.60
	Edge list	24.4	38.20	26.00
	NL Graph	0.01	14.70	7.80
	Others	0.19	0.60	27.40

(d) Graphs: cycle detection (CD), flow estimation (FE) and shortest path (SP)

ⓓ **Markdown**: each schema is represented as a table header in Markdown.

which are illustrated in Figure 3. All models favour the closed bracket format, but there is more variation than for tables. CodeLlama is again the most diverse, with three formats almost getting suggested an equal number of cases. It is also the only model that suggests markdown a significant number of times.

Graphs Following are the most commonly generated formats across all tasks

ⓐ
```
pd.DataFrame.from_records([
    { "Year": 2017, "Quarter": "Dec",
      "Head count": 2104, "Percent": 0.36 },
    { "Year": 2020, "Quarter": "Sep",
      "Head count": 3151,"Percent": 0.42 }
])
```

ⓑ
```
pd.DataFrame({
    "Year": [2017, 2020],
    "Quarter": ["Dec", "Sep"],
    "Head count": [2104, 3151],
     "Percent": [0.36, 0.42]
})
```

ⓒ
```
pd.DataFrame(
    columns=["Year", "Quarter", "Head count", "Percent"],
    data=[
        ["2017", "Dec", "2104", "0.36"],
        ["2020", "Sep", "3151", "0.42"],
    ],
)
```

ⓓ
```
pd.DataFrame(
    data=[
        ["2017", "Dec", "2104", "0.36"],
        ["2020", "Sep", "3151", "0.42"],
    ],
    columns=["Year", "Quarter", "Head count", "Percent"],
)
```

Figure 2: Table dataframe structures obtained from the base model completions for Formula and Python tasks.

ⓐ
```
CREATE TABLE department (
    Department_ID number,
    Name text,
    Creation text,
    Ranking number,
);

CREATE TABLE management (
    department_ID number,
    head_ID number,
    temporary_acting text,
);
```

ⓑ
```
Table department, columns: Department_ID, Name, Creation, Ranking
Table management, columns: department_ID, head_ID, temporary_acting
```

ⓒ
```
department(Department_ID, Name, Creation, Ranking)
management(department_ID, head_ID, temporary_acting)
```

ⓓ
```
## department
| Department_ID | Name | Creation | Ranking |
| ------------- | ---- | -------- | ------- |

## management
| department_ID | head_ID | temporary_acting |
| ------------- | ------- | ---------------- |
```

Figure 3: Database schema representations obtained from the base model completions for the SQL task.

ⓐ **Edge list**: The connection between any two nodes is represented as a triple (i, j, w) with i and j nodes and w the weight of the edge between them.

ⓑ **Adjacency dictionary**: Each node and its associated connected nodes are represented as a list along with their weights.

ⓒ **Adjacency matrix**: The graph is represented as an adjacency matrix format where each entry (i, j) in the matrix represent the weight between nodes i and j.

ⓓ **NL**: Each edge is presented as a sentence *node i is connected to node j with a weight of w*.

which are detailed in Figure 4. we find adjacency dictionary, adjacency matrix and edge list to be the most commonly used representations. There is a significant number of cases where the completions are just textual description about the problem without any graph representation. This prompted us to add the NL format, which was used in recent studies (Wang et al., 2024a; Ye et al., 2023a).

ⓐ
```
[(1, 2, 4), (1, 3, 1),
 (2, 1, 4), (2, 4, 2)]
```

ⓑ
```
{'1': [(2, 4), (3, 1)],
 '2': [(1, 4), (4, 2)]}
```

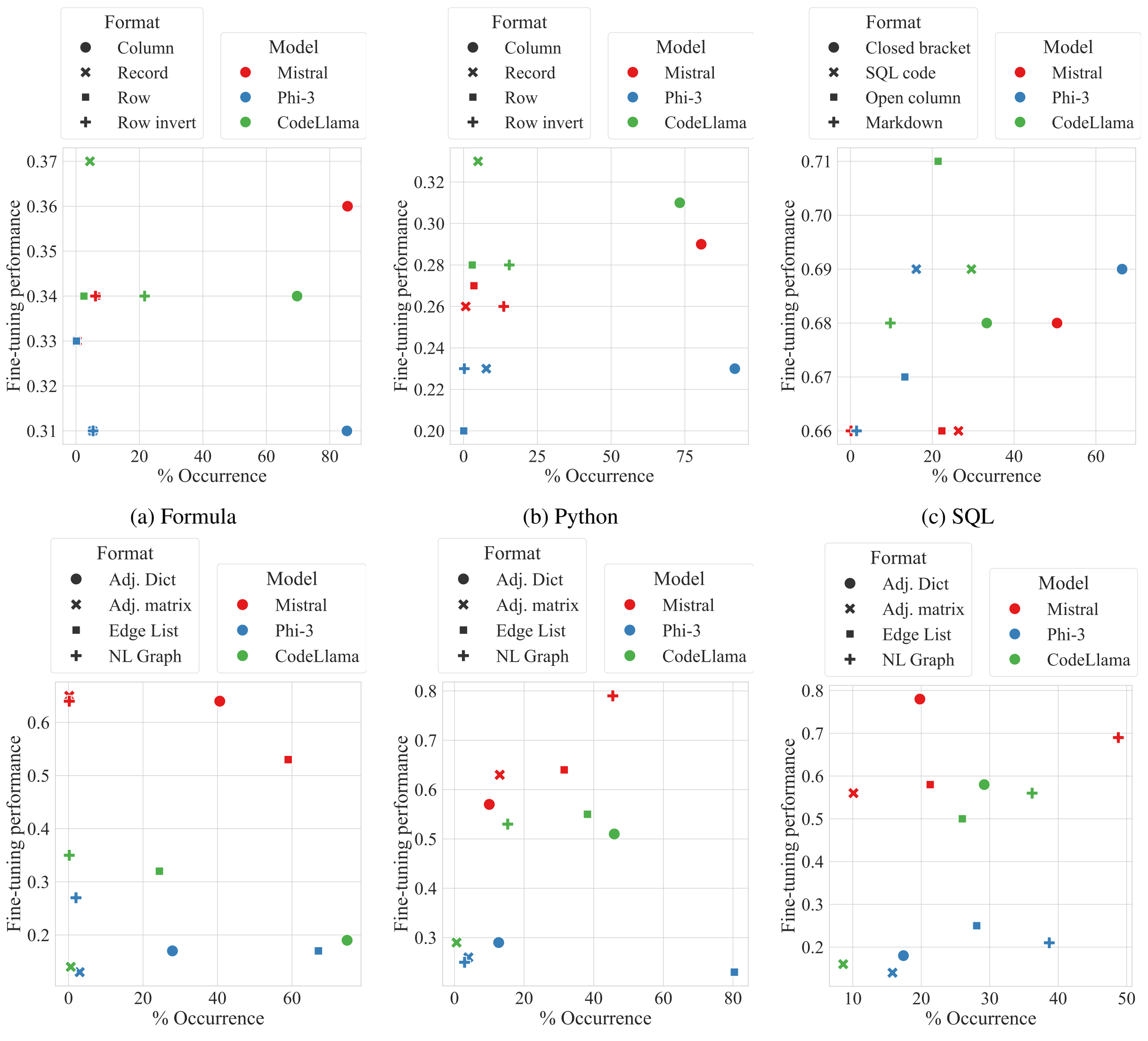
ⓒ Implicit nodes
```
[[0 4 1 0]
 [4 0 0 2]]
```

ⓓ
```
Node 1 is connected to Node 2 with an edge of weight 6.
Node 3 is connected to Node 4 with an edge of weight 5.
```

Figure 4: Graph representations obtained from base model completions.

3.2 Occurrence versus performance

We study the correlation between occurrence statistics and fine-tuning performance in Figure 5, for a total of 18 cases (3 models × 6 problems). There are some correlations (4/18) for Mistral on all text-to-code and flow estimation, and for Phi-3 on Python. For the majority of settings, however, it does not hold, with 5/18 cases (Mistral on cycle detection, Phi-3 on formula and flow estimation, CodeLLama on formula and SQL) obtaining the worst performance for the most common format. This motivates the analysis between performance before and after fine-tuning to still determine an ap-



(a) Formula (b) Python (c) SQL

(d) Cycle Detection (e) Flow estimation (f) Shortest Path

Figure 5: Relation between occurrences of formats and fine-tuning performance on text-to-code tasks for different models. There is some correlation, especially for the Mistral model, but it does not always hold up. This motivates the analysis between performance before and after fine-tuning on these formats in Section 4.

propriate format without different fine-tuning runs.

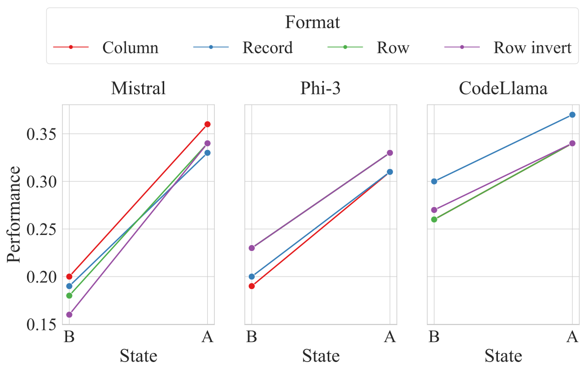
4 Performance before and after fine-tuning

Next, we study the correlation between the performance before and after fine-tuning on different formats to identify any underlying patterns.

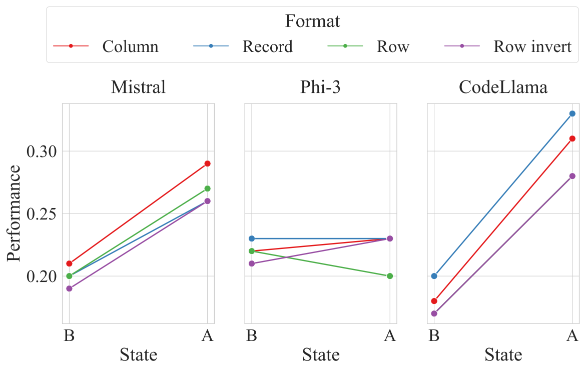
We use a few-shot prompt with three examples in the prompt to evaluate base model performance. Using 3 examples align with previous studies (Brown et al., 2020) where adding a few static examples allows models to infer task-specific patterns. We report results averaged over **three fine-tuning runs** (with different random seeds) to ensure the robustness of results.

4.1 Results

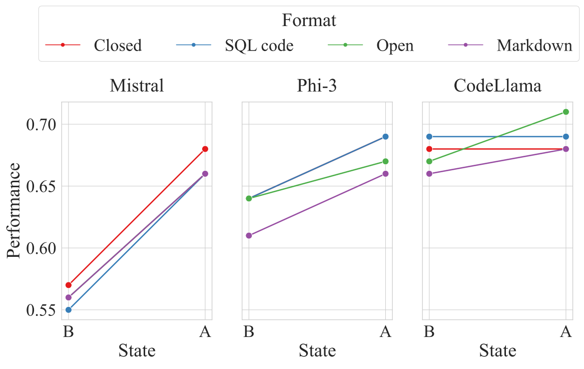
Formula (Figure 6a) We observe that the best performance on the Mistral base model is achieved with the Column format, which also delivers the highest performance after fine-tuning. For Phi-3, both Row and Row-invert format give equal and highest performance before and after fine-tuning. Similarly, for CodeLlama, the Record format yielding the best base model performance continue to give highest performance post fine-tuning. This indicates that different models prefer different representations for the same task. But, the format that performs best during base model inference consistently leads to the best fine-tuning results. Some other interesting insights are: For Mistral, the Row-



(a) Formula

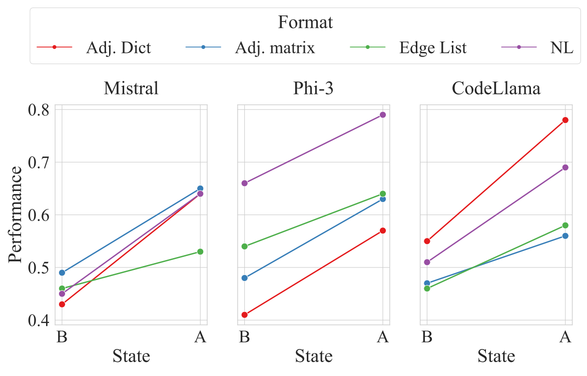


(b) Python

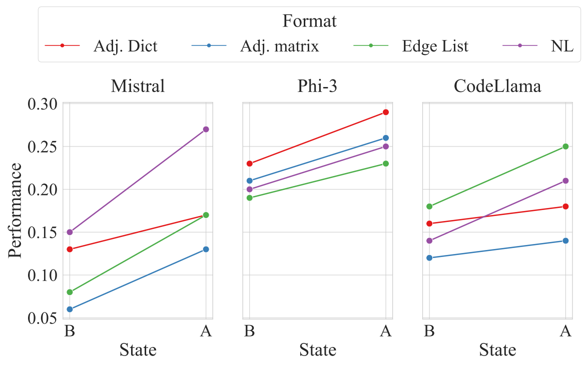


(c) SQL

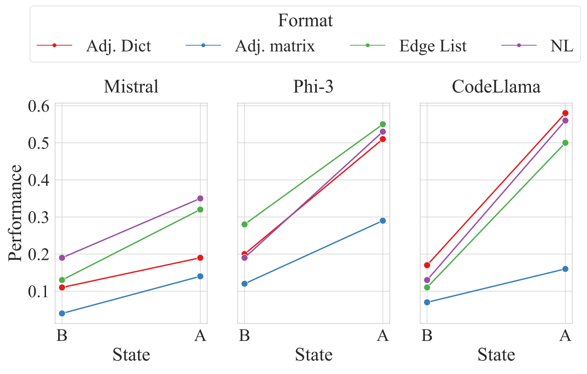
Figure 6: Performance before and after fine-tuning on text-to-code tasks. In 7/9 settings, we can select the right format from the performance before fine-tuning. In one setting (Phi-3 on Python) the best performance before fine-tuning is tied with a winner after fine-tuning. Surprisingly, CodeLlama does not learn anything for the SQL code format, allowing it to be surpassed by the open bracket format after fine-tuning.



(a) Cycle Detection



(b) Flow estimation



(c) Shortest Path

Figure 7: Performance comparison of different formats on Graph Q&A tasks before fine-tuning (base) and after fine-tuning. In all the settings we can directly choose the right format for fine-tuning based on pre-trained model performance.

invert format shows improved performance after fine-tuning, despite being the lowest-performing format in the base model. This suggests that the model learns to better recognize its structure, likely because it closely resembles the markdown format. Interestingly, in all cases, Row and Row-invert formats show same performance after fine-tuning across all models, even though their performance differs before fine-tuning likely because, after fine-tuning, the position of columns and rows no longer significantly affects the model's performance.

Python (Figure 6b) We observe that the best performance for Mistral, both on the base model and after fine-tuning, comes from the Column format, consistent with the Formula task. CodeLlama performs best with the Record format, both before and after fine-tuning. These results hold true for both the Formula and Python tasks, indicating that the Column format for Mistral and the Record-column format for CodeLlama are generally well-suited for tabular understanding tasks. For Phi-3, the performance remains almost same after fine-tuning (except for Row format for which there is a slight decline). This could be because the model was already exposed to similar data during its training, resulting in minimal additional learning during fine-tuning. However, the record format remains best before and after fine-tuning for Phi-3. For the Python task, the trend holds that the format yielding the highest performance on the base model continues to do so post fine-tuning. We also see that it is possible for multiple formats to achieve the highest performance post fine-tuning, but the top-performing format on the base model is always among the leading candidates (as seen with case of record and column Format for Phi-3).

SQL (Figure 6c) For Mistral, the best performance both before and after fine-tuning is achieved with the Closed bracket format. Phi-3 gives equal and best performance for three formats before fine-tuning: closed bracket, SQL code and open column, out of which 2 formats remain the best after fine-tuning: closed Bracket & SQL code. In the case of CodeLlama, the performance either remains the same or improves only for the open column format. Since the base model performance for CodeLlama is already comparable to the fine-tuned performance of the other models, it's likely that the model has encountered this data during training. There is an inconsistency with CodeLlama: the best-performing format before fine-tuning is SQL code, but after fine-tuning, it is the open column format. One hypothesis is that the open column format has the fewest notation to learn, which enables more effective learning during fine-tuning. Overall, the very close performance of different formats on the base model makes it challenging to distinguish the best representation for this task.

Graphs (Figure 7) In all settings, the format that performs best before fine-tuning performs best after fine-tuning. Interestingly, different models have different preferences for different tasks, even if the same model is fine-tuned. Even though the NL format seems the most natural for a *language* model, it does not always outperform the structured formats—perhaps because the models also tend to favor more compact structures over verbose inputs for certain tasks. CodeLlama does better with adjacency dictionary on 2/3 tasks, which relates with it best performance with record format (dictionary like structures) in text-to-code tasks. Adjacency matrix seems to be the least performing format for all models, which shows it is not a suitable structure for shortest path task.

4.2 Conclusion

In summary, performance before fine-tuning allows to predict performance after fine-tuning in **16/18** settings (7/9 for text-to-code and 9/9 for graphs). We conclude that the **pre-trained model allows us to select which format to use for fine-tuning**.

5 Related Work

Recent studies have explored various techniques to represent complex structures, such as tables, graphs, and database schemas, for prompting or in-context learning in large language models (LLMs). These representations are important for enabling LLMs to understand structural information effectively. Research has shown that the performance of LLMs can be sensitive to the choice of format (Fang et al., 2024; Fatemi et al., 2023) which highlights the need to determine optimal representations for fine-tuning tasks.

Tabular data representation For tabular data, Sui et al. (2024) proposed a method where LLMs generate explanations for table structures, which are then used to re-prompt the model for improved performance. Other approaches such as Gong et al. (2020), employed a template-based approach to convert table records into natural language sentences, concatenating them for final representation. Singha et al. (2023), showed that certain formats, like JSON or df-loader, work best for particular table understanding tasks, while (Ye et al., 2023b) and (Wang et al., 2024b) demonstrated the effectiveness of the PIPE format for table reasoning tasks. Furthermore, Jaitly et al. (2023) explored LaTeX-based serialization for table classification tasks. Despite these insights, it remains unclear whether a generalized approach can be adopted for different models and different tasks.

Graph structure representation Similarly, for graph-based tasks, various methods have been proposed to encode graph structures. Earlier works, such as Wang et al. (2024a) and Ye et al. (2023a), employed natural language descriptions to represent graph edges and nodes uniquely for each subtask. However, these verbalized graphs can become lengthy, unstructured, and difficult for both humans and models to process (Jin et al., 2023). While Guo et al. (2023) suggested that appending explanations to the graph structure can improve performance, our results have been inconsistent. Alternative approaches, like Chai et al. (2023), introduced encoder-decoder architectures specifically designed to learn graph encodings. However, our study aims to assess the impact of format representation within the LLM itself for different GraphQA tasks, without relying on external encoders. Guo et al. (2023) evaluated three common formats—edge lists, adjacency matrices, and GraphML descriptions—for their effectiveness in graph tasks. However, it is difficult for a non-expert practitioner to be aware of all possible formats.

Other representations In the context of database schema representation for Text-2-SQL tasks, Gao et al. (2023) explored different formats inspired by external knowledge sources such as OpenAI prompt demonstrations and Alpaca SFT prompts. While their work leverages predefined formats, our study seeks to derive the schema representations directly from the model's knowledge.

Data Selection The work by (Liu et al., 2022) show that selecting data from the pre-training data that has a similar distribution to the fine-tuning data increases the value of starting from a pre-trained model, because it can reduce the effect of catastrophic forgetting. However, in our work when we extract formats encountered in the pre-training data—by letting the model auto-complete the data, which informally optimizes the probability of that format given the task—the most common formats are not the best formats. This likely happens because pre-training data for language models is typically unstructured, causing a global bias towards formats that are more common and ignoring the task.

6 Conclusion

Our study shows that language models can implicitly suggest candidate formats that are effective for fine-tuning reducing the developer's reliance on personal experience with structured formats. Given that base model performance varies across different formats for representing structures, we investigate whether fine-tuning teaches the model to use other formats, or if the difference in performance persists. To this end, we examine the correlation between base model performance and post-finetuning outcomes across formats. Notably, the format that performs best on the base model consistently ranks among the top candidates after fine-tuning. Through experiments on various data structures, we show that these findings are broadly applicable. This approach offers a practical way to select appropriate formats for fine-tuning without relying on trial and error, saving both time and computational resources during training.

7 Limitations

In this study we consider only easily accessible models. There are two main dimensions to the cost of fine-tuning an LLM: the GPU requirement and the time to train on that GPU. We fix the first dimension to accessible models that can be trained on a single A100.

While we have shown a correlation between the performance of the base model and the fine-tuned model, this analysis is limited to a single piece of structured data in the prompt. We have not evaluated other parts of the prompt, nor the combination of different sources. Since our focus was specifically on structured data, we have restricted our analysis to that area.

Additionally, we observed two instances in our study—text-to-Python for Phi-3 and text-to-SQL for CodeLlama—where fine-tuning did not yield significant improvements over base model performance. We understand that these are common tasks, and it is possible that the base model was already trained on similar tasks. However, we cannot definitively determine whether this is the case.

References

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.

Ziwei Chai, Tianjie Zhang, Liang Wu, Kaiqiao Han, Xiaohai Hu, Xuanwen Huang, and Yang Yang. 2023. Graphllm: Boosting graph reasoning ability of large language model. *arXiv preprint arXiv:2310.05845*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming

Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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

Bahare Fatemi, Jonathan Halcrow, and Bryan Perozzi. 2023. Talk like a graph: Encoding graphs for large language models. *arXiv preprint arXiv:2310.04560*.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. Text-to-sql empowered by large language models: A benchmark evaluation. *arXiv preprint arXiv:2308.15363*.

Heng Gong, Yawei Sun, Xiaocheng Feng, Bing Qin, Wei Bi, Xiaojiang Liu, and Ting Liu. 2020. Tablegpt: Few-shot table-to-text generation with table structure reconstruction and content matching. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 1978–1988.

Jiayan Guo, Lun Du, Hengyu Liu, Mengyu Zhou, Xinyi He, and Shi Han. 2023. Gpt4graph: Can large language models understand graph structured data? an empirical evaluation and benchmarking. *arXiv preprint arXiv:2305.15066*.

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

Sukriti Jaitly, Tanay Shah, Ashish Shugani, and Razik Singh Grewal. 2023. Towards better serialization of tabular data for few-shot classification. *arXiv preprint arXiv:2312.12464*.

Jinhao Jiang, Kun Zhou, Zican Dong, Keming Ye, Wayne Xin Zhao, and Ji-Rong Wen. 2023. Structgpt: A general framework for large language model to reason over structured data. *arXiv preprint arXiv:2305.09645*.

Bowen Jin, Gang Liu, Chi Han, Meng Jiang, Heng Ji, and Jiawei Han. 2023. Large language models on graphs: A comprehensive survey. *arXiv preprint arXiv:2312.02783*.

Ziquan Liu, Yi Xu, Yuanhong Xu, Qi Qian, Hao Li, Xiangyang Ji, Antoni Chan, and Rong Jin. 2022. Improved fine-tuning by better leveraging pre-training data. *Advances in Neural Information Processing Systems*, 35:32568–32581.

Usneek Singh, José Cambronero, Sumit Gulwani, Aditya Kanade, Anirudh Khatry, Vu Le, Mukul Singh, and Gust Verbruggen. 2024. An empirical study of validating synthetic data for formula generation. *arXiv preprint arXiv:2407.10657*.

Ananya Singha, José Cambronero, Sumit Gulwani, Vu Le, and Chris Parnin. 2023. Tabular representation, noisy operators, and impacts on table structure understanding tasks in llms. *arXiv preprint arXiv:2310.10358*.

Ananya Singha, Bhavya Chopra, Anirudh Khatry, Sumit Gulwani, Austin Henley, Vu Le, Chris Parnin, Mukul Singh, and Gust Verbruggen. 2024. Semantically aligned question and code generation for automated insight generation. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, pages 127–134.

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.

Heng Wang, Shangbin Feng, Tianxing He, Zhaoxuan Tan, Xiaochuang Han, and Yulia Tsvetkov. 2024a. Can language models solve graph problems in natural language? *Advances in Neural Information Processing Systems*, 36.

Zilong Wang, Hao Zhang, Chun-Liang Li, Julian Martin Eisenschlos, Vincent Perot, Zifeng Wang, Lesly Miculicich, Yasuhisa Fujii, Jingbo Shang, Chen-Yu Lee, et al. 2024b. Chain-of-table: Evolving tables in the reasoning chain for table understanding. *arXiv preprint arXiv:2401.04398*.

Ruosong Ye, Caiqi Zhang, Runhui Wang, Shuyuan Xu, Yongfeng Zhang, et al. 2023a. Natural language is all a graph needs. *arXiv preprint arXiv:2308.07134*, 4(5):7.

Yunhu Ye, Binyuan Hui, Min Yang, Binhua Li, Fei Huang, and Yongbin Li. 2023b. Large language models are versatile decomposers: Decomposing evidence and questions for table-based reasoning. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 174–184.

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, et al. 2022. Natural language to code generation in interactive data science notebooks. *arXiv preprint arXiv:2212.09248*.

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*.

Alex Zhuang, Ge Zhang, Tianyu Zheng, Xinrun Du, Junjie Wang, Weiming Ren, Stephen W Huang, Jie Fu, Xiang Yue, and Wenhu Chen. 2024. Structlm: Towards building generalist models for structured knowledge grounding. *arXiv preprint arXiv:2402.16671*.