Solving Data-centric Tasks using Large Language Models

Anonymous ACL submission

Abstract

Large language models (LLMs) are rapidly replacing help forums like StackOverflow, and are especially helpful for non-professional programmers and end users. These users are often interested in *data-centric tasks*, such as spreadsheet manipulation and data wrangling, which are hard to solve if the intent is only communicated using a natural-language description, without including the data. But how do we decide how much data and which data to include in the prompt?

This paper makes two contributions towards answering this question. First, we create a dataset of real-world NL-to-code tasks manipulating tabular data, mined from StackOverflow posts. Second, we introduce a *cluster-then-select* prompting technique, which adds the most representative rows from the input data to the LLM prompt. Our experiments show that LLM performance is indeed sensitive to the amount of data passed in the prompt, and that for tasks with a lot of syntactic variation in the input table, our cluster-then-select technique outperforms a random selection baseline.

1 Introduction

Code-generating large language models (LLMs) promise to empower end users interested in *data-centric tasks*, ranging from string manipulations in spreadsheets to data cleaning and analysis in computational notebooks. For example, consider the following task on tabular data: given a column with *full names*, generate a new column with *user names*, by combining the first initial and last name, in lowercase. This task can be solved by a Pandas program that: 1) splits the full name into a list of strings, 2) extracts the first and last string from the list, 3) converts both to lowercase and joins the first letter of one string to the other. The challenge in generating this program is that data rows often have varied formats, *e.g.* most rows only have two names ("John Smith"), but some have multiple middle names ("Jake L Woodhall", "Jo Anna Emily Gray"). If an LLM prompt does not include any data or only includes rows with two names, the LLM is more likely to generate a program that does not generalize (*e.g.* one that extracts the last name as the *second* element of the list instead of *last*).

In this paper, we focus on solving such tasks that involve multi-step computations on the input columns to generate additional columns. Towards this goal, we mine StackOverflow to construct a new dataset, dubbed SOFSET, of data-centric tasks, equipped with a natural-language query and a small input table. Using this dataset, we conduct experiments on generating Pandas programs using GPT-4 and an open-source alternative CODELLAMA, with the goal of analysizing LLMs' sensitivity to the amount of input data provided in the prompt.

Unlike input tables in StackOverflow posts, real-world data tables are often large, hence sending the entire table to the LLM is likely impractical, expensive, or detrimental to performance. *How do we best convey the structure of a large input table to the LLM?* To address this question, we propose a *cluster-then-select* prompting technique that clusters input rows based on their syntactic structure and then selects representative rows from each cluster; *e.g.* in our "user name" example, the technique would include a row for each number of middle names. To evaluate this technique, we perform experiments on SOFSET augmented with larger input tables extracted from Kaggle.

In summary, this paper contributes:

- a real-world dataset of complex tasks for evaluating data-centric code generation;
- a *cluster-then-select* technique for selecting rows to prompt with, from large input tables;
- an analysis that shows LLMs are sensitive to the data quantity, choice and position of rows.

2 Related work

Large language models for tabular data Code-generating LLMs like Codex (Chen et al., 2021) and PaLM (Chowdhery et al., 2022) have been fine-tuned for code-specific tasks and adapted for data-centric domains like SQL (Trummer, 2022; Rajkumar et al., 2022). (Li et al., 2020) investigate the ability of language models like BERT to perform entity matching on tabular data. (Narayan et al., 2022) use GPT-3 for data cleaning, error detection and entity matching tasks. (Hegselmann et al., 2023) focus on tabular classification tasks and investigate parameter-efficient tuning of LLMs.

Prompting for data-centric tasks In this paper, we ask the question: *how does data context impact code generation for data-centric tasks?* Previous works have explored prompting with data: (Jain et al., 2022) provide both input and expected output tables (which might not be available in a realistic setting). (Gemmell and Dalton, 2023) prompt with transformed tables after filtering out rows that are not relevant for their question-answering tasks. (Hegselmann et al., 2023) serialize data tables into a textual representation for tabular classification tasks. (Yin et al., 2022) focuses on data-centric tasks in computational notebooks. These works focus on prompting for data analysis, classification and wrangling tasks (in-place data transformations) whereas we focus on multi-step data manipulation. We propose a new *cluster-then-select* prompting technique that clusters the input data and adds representative rows to the prompt.

3 The SOFSET Dataset

We collect a new dataset fashioned from real-world data-centric tasks from StackOverflow (SOFSET). We sample tasks deterministically from the highest rated posts with the tag "ExcelFormulas" in StackOverflow (as of March 2022). These tasks are representative of real problems spreadsheet users face frequently since they correspond to the highest-rated posts. We manually check that the posts are genuine tasks and also remove post identifiers for anonymization. This gives us a total of 201 tasks.

3.1 Dataset Annotation

Each datapoint in our dataset is annotated with a concise textual query, a data input (column-major-flat table), an expected correct output (extra columns), a pandas solution and metadata. We manually write the textual queries, summarising the original verbose StackOverflow question. Each query is annotated and verified by at least 3 internal annotators. For the data input, we use the table from the original StackOverflow post (if available), and add extra rows and corner cases until we have at least 10 rows. As the NL query and tabular data are not verbatim copies from StackOverflow and we have a different target language (Pandas instead of Excel Formulas), the evaluation data should not be present in the training data. We choose Pandas as the target language since LLMs are especially good at generating Python but our methods and dataset are programming-language agnostic.

3.2 Dataset Properties

What makes our dataset different from existing ones? First, our dataset consists of complex data-centric tasks with several input columns. Prior python datasets like (Hendrycks et al., 2021; Chen et al., 2021) are not data-centric. Second, our dataset is larger than existing data-centric datasets, JIGSAW (Jain et al., 2022) and CERT (Zan et al., 2022). JIGSAW has 79 unique tasks (median of 7 data rows) and CERT has 100 unique tasks (median of 3 rows). Our dataset has 201 unique tasks, with a median of 10 rows. The SPIDER dataset (Yu et al., 2018) is a text-to-SQL dataset which focuses on relational query tasks whereas we focus on fine-grained data wrangling and manipulation tasks. Finally, we propose a *taxonomy* of data-centric tasks, classifying them into data-independent (IND), data-dependent (DEP), and external-dependent (EXT), based on the data required to produce a solution.

Data-independent tasks These tasks can be solved using the query alone without any data access. An example is the query "create a new column that includes only the first 5 characters from Filename".

Data-dependent tasks These tasks cannot be solved using the query alone: the model needs access to the input table. For example, the query "create a new column with the number of days between the two date columns" requires data access to identify the correct column names and date format, both absent from the query.

External-dependent tasks These tasks can only be solved with external world knowledge in addition to data access. The query "create a new column that counts how many US holidays are between the dates in Start Date and End Date", requires the model to know about US holidays.

Following this taxonomy, SOFSET consists of

126 IND tasks, 44 DEP tasks and 31 EXT tasks. These tasks span diverse domains including string manipulation, date and time, math, address, and complex conditionals among others.

3.3 Cluster-then-select prompting technique

To solve tasks on large tables, we propose a *cluster-then-select* technique which prompts the model with a representative sample of the input data. In order to capture the syntactic variation in the input data, we rely on an existing tool (Padhi et al., 2018), which takes as input a set of strings and synthesizes a small set of regular expressions (regexes), such that each input string matches one of the regexes. In our "user name" example from the introduction, it would synthesize separate regexes for rows with zero, one, and two middle names. These regexes are then used to cluster the input strings, and we select some number of rows from each cluster.

If the input table only has one column, selecting n representative rows based on the clustering results is trivial: simply pick one row each from the top-n most populous clusters. In cases where the input contains more than one column, they may be clustered differently. We then select n rows that together cover as many strings as possible across all the columns. We frame this as a *weighted maximal coverage problem* (max), which can be solved approximately in a greedy manner. In each iteration, the algorithm selects the rows whose elements maximize cluster coverage.

Kaggle-augmented dataset In order to evaluate our *cluster-then-select* technique on larger datasets, we expand the 44 data-dependent tasks by adding more rows from open-source Kaggle datasets (kag), bringing the total to 1000. We first identify the specific data domains in the original SOFSET rows (such as names, numbers, address, date, time etc) and then source comparable open-source datasets from Kaggle of the same domain. We then post-process the Kaggle data to maintain the original rows format, while also introducing greater variation which increases the number of data clusters. 62% of our DEP tasks have at least 2 clusters and we have tasks with up to 10 clusters. Since the Kaggle data is post-processed and is not tied to the task query in any way, it is unlikely to bias the LLM evaluation by being part of the training data. This larger dataset allows for a thorough evaluation, better mirroring real-world conditions.

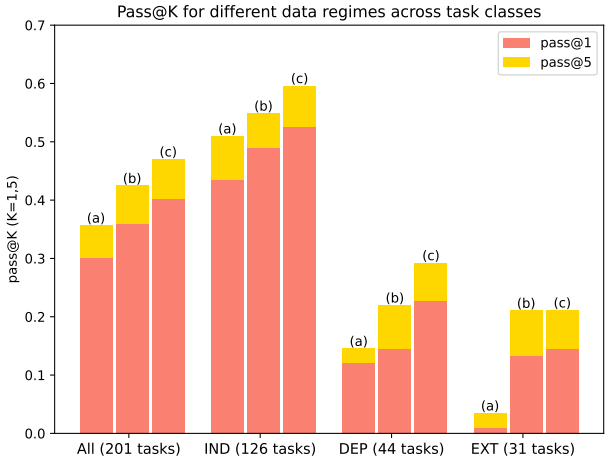

Figure 1: pass@k with (a) no-data, (b) first-row, and (c) ten-rows passed to the model. The leftmost group of bars represent pass@k with all classes followed by separate pass@k for IND, DEP and EXT tasks.

4 Evaluation of data-centric tasks

We perform an analysis of the role of data on model performance in data-centric tasks. We first use the original SOFSET dataset to examine three data regimes with increasing amounts of data: (a) no-data (b) first-row and (c) ten-rows and the taxonomy of task classes of increasing difficulty in terms of data required: IND, DEP and EXT. We then use Kaggle-augmented DEP tasks to compare our *cluster-then-select* technique (which selects representative rows from the top-n most dense clusters) against a random baseline (which selects random rows from the input). For each data setting, we construct a prompt which contains the task query and selected rows as a pandas dataframe to generate code using GPT-4. Correctness is reported based on whether the code produces the expected output in terms of pass@k, the probability that at least one of k samples of generated code produces the correct output (Chen et al., 2021).

Does model performance vary with the amount of data passed for different task classes? Figure 1 shows the impact of the amount of data on LLM performance, first for the entire dataset and then split by task classes. We see a larger drop in performance with reduced (and no) data on DEP (and EXT) tasks compared to IND tasks. Specifically, the performance gap (pass@5) between first-row and no-data regimes is larger for the DEP and EXT classes (33.8% and 83.5% resp) compared to only 7.1% for IND tasks. The fact that there is any performance drop for IND tasks indicates that having

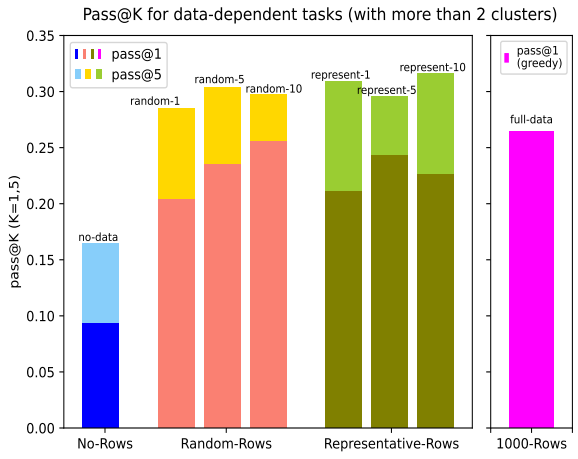

Figure 2: pass@k for 39% (17 out of 44) DEP tasks (with more than two clusters) with no-data, random selection (random-n), representative selection (represent-n) and pass@1 with greedy sampling for full-data (1000 rows). Completions are evaluated on 1000 rows.

data helps the model even when the problem can be solved independently of data. In the absence of data, almost no EXT task is solved (pass@1) but performance improves when a single row is passed.

Is our cluster-then-select technique effective on larger input tables? We evaluate our *cluster-then-select* technique on Kaggle-augmented DEP tasks (with 1000 rows) since we expect to see the benefit of our approach more clearly on tasks dependent on data. In order to do so, we compare our representative selection strategy against *random* selection where the rows are randomly selected from the input table. Among DEP tasks, we further only focus on 17 (out of 44) that have input columns with at least three clusters, since with two clusters or fewer we don't expect to see much difference between the representative and random samples. We also evaluate against two baselines: no-data (0 rows) and full-data (all 1000 rows). We run random selection experiments five times.

Figure 3 shows that the model performs best with 10 most representative rows added to the prompt (pass@5 for represent-10 = 0.32). Representative selection performs slightly better than random for the same number of rows. Specifically, represent-1 and represent-10 outperform random-1 and random-10 by 8% and 6% resp. In addition, random selection has *high variance*, especially for a small number of rows (*e.g.* pass@1 for random-1 varies from 0.20 to 0.31 across the five runs), which is not surprising, since the random strategy might select rows from different clusters or from the same one. Thus, while random selection gives comparable results on average, our cluster-then-select technique offers a more consistent approach to provide the model a representative sample of the data. Further, the low pass@k for our no-data baseline suggests that our dataset was not part of the training data, as then the model would likely perform well even without data input. We note that while we evaluate on 1000 rows, the same cluster-then-select technique could easily scale to datasets with over 100K rows without much overhead.

Does the position of data rows in the prompt also affect performance? For the full-data baseline, we used a longer-context version of GPT-4 (32k) with temperature 0 (*greedy selection to eliminate variance in the generations*) for the same 17 DEP tasks. The right side of Figure 2 shows pass@1 for this setting with ten runs: we permute the 1000 rows in the dataframe ten times, in order to measure the sensitivity of the model to row positioning. We observe a high variance in pass@1 values, ranging from 0.20 to 0.32 with an average of 0.26. This shows that the position of rows in the dataframe influences completions quality, which aligns with previous findings about positional biases in prompts (Liu et al., 2023). Surprisingly, the full-data setting (irrespective of row ordering) performs worse than selecting one random row in some cases (pass@1 for one random row ranges from 0.12 to 0.27 with an average of 0.20). [1]

5 Conclusion and Future Work

Our work highlights the importance of data for code generation on data-centric tasks and proposes a new dataset for evaluation of data-centric tasks. We show that providing even one data row to the model boosts performance compared to a no-data baseline. Since providing the entire input data is often infeasible, we propose a *cluster-then-select* prompting technique that selects representative rows from the data to be added to the prompt. While randomly selecting rows also performs well, for data with a high degree of syntactic variation, it is more beneficial to add representative rows to the prompt. For future work, handling a broader problem space (*e.g.*, multi-table inputs, hierarchical table inputs) raises interesting challenges.

[1] Experiments with all DEP tasks on GPT-4 are in Figure 3 and CODELLAMA experiments are Figure 6, Figure 7, Figure 8.

6 Limitations

We discuss the limitations of our work in terms of the SOFSET dataset, the cluster-then-select prompting technique and the models used for evaluation. Although starting from actual user-specified problems gives our results greater alignment with real spreadsheet user problems, the form that such queries take pose some potential limitations to our analysis. Users usually only show relevant columns of data in their queries when in actuality there might be many more unrelated columns in real spreadsheets. We have seen good results applying LLMs to spreadsheets with many columns that are extraneous to the query but we do not perform a rigorous evaluation of the same. Furthermore, since we have collected only English queries from StackOverflow, our results may not generalize to other languages.

Since we draw our conclusions from the generations produced by the GPT-4 model, future models might invalidate our conclusions. Furthermore access to models such as GPT-4 cannot be taken for granted and the costs of running our evaluation are considerable. Even open source models like CODELLAMA require available GPU resources to evaluate.

Finally, our cluster-then-select prompting technique is based on the regular expression synthesis algorithm from (Padhi et al., 2018). Given that the clusters for the input data columns are defined by the specificity of this regex synthesis, using a different synthesis algorithm could potentially result in a different set of clusters.

7 Broader Research Impact

To the best of our knowledge, research on prompting large language models to solve data-centric tasks with tabular data is infrequent, despite the considerable importance of such scenarios. Solving the problem of how to help LLM reason over large amounts of data is essential to the future of assisted decision making. Generating multi-step programs that require reasoning is the beginning of this journey and to make progress the community needs challenging real-world datasets to evaluate on. By releasing our new dataset, sharing the analysis results of our experiments and releasing our prototype tool[2], we offer valuable benchmarks and a baseline to the wider research community which promises to encourage further exploration.

[2]discussed in Appendix C

8 Ethics Statement

There are broad ethical impacts resulting from the creation of AI models that attempt to generate code solutions from natural language descriptions and these are discussed in detail in previous papers including Codex (Chen et al., 2021), AlphaCode (Li et al., 2022), and PaLM (Chowdhery et al., 2022). These impacts include over-reliance, misalignment between what the user expressed and what they intended, potential for bias and under/over representation in the model results, economic impacts, the potential for privacy and security risks, and even environmental considerations. All of these considerations also apply to the work described here. Our focus is to highlight how the presence of data improves the performance of these models but it is important to note that the quality of the data used in the prompt will impact whether the resulting generation exhibits bias, exposes private data, etc. We explore the overall impact of providing data as part of the prompt but do not conduct a more focused analysis of determining how bias in the prompt data might influence the resulting code generation, a task we leave for future work.

There is the question of the sources of data and of consent to use the data in the manner exhibited in this paper. We have reviewed each of the datasets we have included in this paper to ensure that our use is compatible with the intent of the authors and publishers. Our datasets have also been reviewed by our institution's ethics board to review that this is an ethical use.

We are wary of using the word "understand" in this paper. It has been correctly argued that language models do not really "understand" language in the sense of connecting language's syntactic content with the semantics of the physical world (Bender and Koller, 2020; Webson and Pavlick, 2022). There have long been critics of the use of such terms in AI research (Agre et al., 1997). Nonetheless, large language models have shown themselves in certain situations to be capable of the syntactic manipulation of language which in humans we take to be commonsense evidence of understanding. This is the less contentious manner in which we use the word. Thus our intention in using the word is not to claim that models can connect data with real-world concepts, but rather that the model can manipulate language about data in a useful manner, where "useful" is defined by our quantitative benchmarks.

This paper does not directly contribute to a tool built on the assumed capabilities of language models to understand data, but nonetheless, it is motivated by their potential applications in such tools. These tools may be deployed in many data applications such as databases, spreadsheets, and business intelligence applications. Depending on the audience of the tool, various interaction design concerns arise. Explainability of the model is a key consideration, and the tool should offer decision support to evaluate mispredictions and potential next steps (Sarkar, 2022). Previous research of non-experts using inference driven tools for data manipulation has shown the importance of tool design in the critical appreciation of the model and its limitations, and in the potential cost of errors (Williams et al., 2020; Sarkar et al., 2015). As an exploratory paper without a concrete application, we do not encounter these issues, but the project has nonetheless been reviewed by our institution's ethics board.

References

Kaggle datasets.

Maximum coverage problem.

Philip E Agre et al. 1997. Lessons learned in trying to reform AI. *Social science, technical systems, and cooperative work: Beyond the Great Divide*, 131.

Emily M Bender and Alexander Koller. 2020. Climbing towards nlu: On meaning, form, and understanding in the age of data. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pages 5185–5198.

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

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek B Rao, Parker Barnes, Yi Tay, Noam M. Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Benton C. Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier García, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Oliveira Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Díaz, Orhan Firat, Michele Catasta, Jason Wei, Kathleen S. Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. Palm: Scaling language modeling with pathways. *ArXiv*, abs/2204.02311.

Michael Droettboom, Roman Yurchak, Hood Chatham, Dexter Chua, Gyeongjae Choi, Marc Abramowitz, casatir, Jan Max Meyer, Jason Stafford, Madhur Tandon, Michael Greminger, Grimmer Kang, Chris Trevino, Wei Ouyang, Joe Marshall, Adam Seering, Nicolas Ollinger, Ondřej Staněk, Sergio, Teon L Brooks, Jay Harris, Alexey Ignatiev, Seungmin Kim, Paul m. p. P., jcaesar, Carol Willing, Cyrille Bogaert, Dorian Pula, Frithjof, and Michael Jurasovic. 2022. Pyodide: A Python distribution for WebAssembly (0.19.0).

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

Stefan Hegselmann, Alejandro Buendia, Hunter Lang, Monica Agrawal, Xiaoyi Jiang, and David Sontag. 2023. Tabllm: Few-shot classification of tabular data with large language models. In *Proceedings of The 26th International Conference on Artificial Intelligence and Statistics*, volume 206 of *Proceedings of Machine Learning Research*, pages 5549–5581. PMLR.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Xiaodong Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *ArXiv*, abs/2105.09938.

Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *International Conference on Software Engineering (ICSE)*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago,

et al. 2022. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*.

Yuliang Li, Jinfeng Li, Yoshihiko Suhara, AnHai Doan, and Wang-Chiew Tan. 2020. Deep entity matching with pre-trained language models. *arXiv preprint arXiv:2004.00584*.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts.

Avanika Narayan, Ines Chami, Laurel Orr, and Christopher Ré. 2022. Can foundation models wrangle your data? *arXiv preprint arXiv:2205.09911*.

Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd Millstein. 2018. Flashprofile: a framework for synthesizing data profiles. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28.

Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. 2022. Evaluating the text-to-sql capabilities of large language models. *ArXiv*, abs/2204.00498.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Advait Sarkar. 2022. Is explainable AI a race against model complexity? *arXiv preprint arXiv:2205.10119*.

Advait Sarkar, Mateja Jamnik, Alan F Blackwell, and Martin Spott. 2015. Interactive visual machine learning in spreadsheets. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 159–163. IEEE.

Immanuel Trummer. 2022. Codexdb: Generating code for processing sql queries using gpt-3 codex. *ArXiv*, abs/2204.08941.

Albert Webson and Ellie Pavlick. 2022. Do prompt-based models really understand the meaning of their prompts? In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2300–2344, Seattle, United States. Association for Computational Linguistics.

Jack Williams, Carina Negreanu, Andrew D Gordon, and Advait Sarkar. 2020. Understanding and inferring units in spreadsheets. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–9. IEEE Computer Society.

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*.

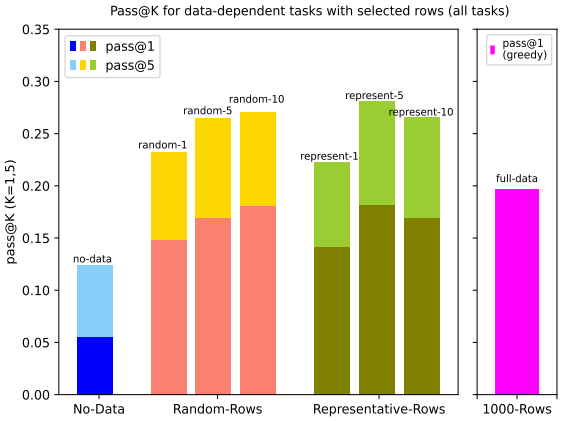

Figure 3: pass@k for DEP tasks with no-data, and n=1, 5 and 10 rows passed to the model, using random (random-n), representative selection (represent-n). The completions are evaluated on 1000 rows.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. 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, Brussels, Belgium, October 31 - November 4, 2018*, pages 3911–3921. Association for Computational Linguistics.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. Cert: Continual pre-training on sketches for library-oriented code generation. *ArXiv*, abs/2206.06888.

A Community Data License Agreement - Permissive - Version 2.0

This is the Community Data License Agreement - Permissive, Version 2.0 (the "agreement"). Data Provider(s) and Data Recipient(s) agree as follows:

A.1 Provision of the Data

- A Data Recipient may use, modify, and share the Data made available by Data Provider(s) under this agreement if that Data Recipient follows the terms of this agreement.

- This agreement does not impose any restriction on a Data Recipient's use, modification, or sharing of any portions of the Data that are in the public domain or that may be used, modified, or shared under any other legal exception or limitation.

A.2 Conditions for Sharing Data

- A Data Recipient may share Data, with or without modifications, so long as the Data Recipient makes available the text of this agreement with the shared Data.

A.3 No Restrictions on Results

- This agreement does not impose any restriction or obligations with respect to the use, modification, or sharing of Results.

A.4 No Warranty; Limitation of Liability

- All Data Recipients receive the Data subject to the following terms:

THE DATA IS PROVIDED ON AN "AS IS" BASIS, WITHOUT REPRESENTATIONS, WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NONINFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

NO DATA PROVIDER SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE DATA OR RESULTS, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

A.5 Definitions

- "Data" means the material received by a Data Recipient under this agreement.
- "Data Provider" means any person who is the source of Data provided under this agreement and in reliance on a Data Recipient's agreement to its terms.
- "Data Recipient" means any person who receives Data directly or indirectly from a Data Provider and agrees to the terms of this agreement.
- "Results" means any outcome obtained by computational analysis of Data, including for example machine learning models and models' insights.

B Software License Agreement

MIT License

All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

C Our Prototype Tool

This section explains the workflow of our system using a running example in Fig. 5. In this example, there is an input table full names and wants to create a new column with user names, by concatenating the first initial and the last name and converting them to lowercase.

The high-level workflow of our tool is depicted in Fig. 4 and formalized in Algorithm 1. The tool takes as input a user query Q expressed in natural language, user input table T as a Pandas dataframe, and the target cardinality k of distinct completions to generate. To ensure termination within a reasonable time, we set a limit k_{max} on the number of calls to LLM ($k_{max} = 8k$). For our running example, k is 1, Q is "create a new column in lowercase that concatenates the first initial and the last name.", and T is `Data({"Names":["John Smith", "Jack Will Anders", ...]})`. At a high-level, the algorithm first clusters the data in T based on automatically synthesized regular expressions and

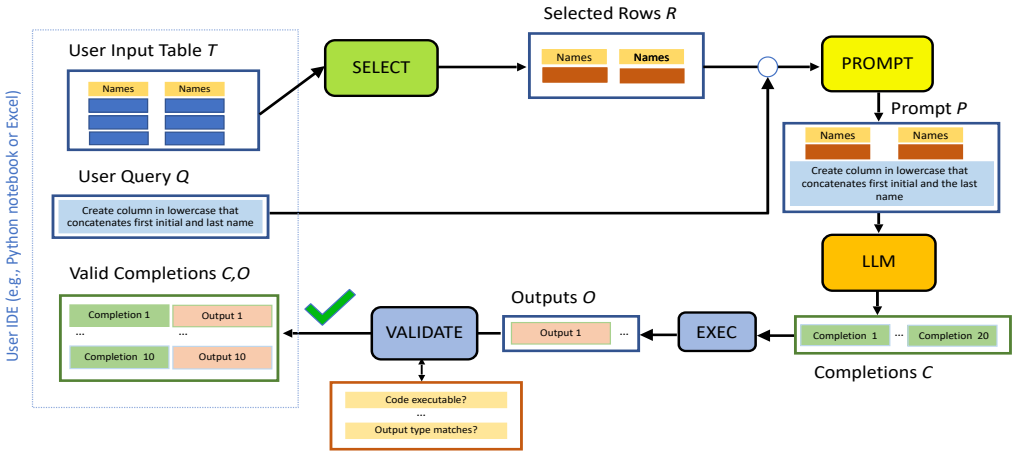

Figure 4: Our tool transforms an input table and a user query into a list of valid completions. The input data is used to extract the selected rows R. The resulting rows and query are used to construct a prompt which is fed to a code synthesis LLM, such as GPT-4 or CODELLAMA, generating multiple possible completions. The outputs of these completions are then validated and the first k valid completions (along with the outputs) are returned.

Algorithm 1 Inference Algorithm

Input: Explicit: user query Q, input table T, cardinality k. Implicit: completion limit k_{max} (with $k \leq k_{max}$), number n of rows to be selected.
Output: Pair of lists (C, O), with $|C| = |O| \leq k$, of unique completions and their corresponding outputs.

1: **procedure** INFER(Q, T, k)
2: $M \leftarrow$ CLUSTER(T) ▷ map elements into clusters
3: $R \leftarrow$ SELECT(T, n, M) ▷ select n most representative rows from T
4: $P \leftarrow$ PROMPT(Q, R) ▷ prompt creation
5: $B, C, O \leftarrow k_{max}, [], []$ ▷ initialize budget, caches
6: **while** $B > 0 \wedge |C| < k$ **do**
7: $c \leftarrow$ LLM(P) ▷ sample completion
8: $B \leftarrow B - 1$ ▷ decrement budget
9: $o \leftarrow$ EXEC(c, T) ▷ execute against T
10: **if** VALIDATE(o) $\wedge$ ($c \notin C$) **then**
11: $C \leftarrow C + [c]$ ▷ append completion to C
12: $O \leftarrow O + [o]$ ▷ append output to O
13: **return** (C, O)

stores them in a map M (line 2). It then extracts *representative rows* of the table using SELECT (line 3); combines the query Q and the rows R to create a prompt P using PROMPT (line 5); and then queries LLM repeatedly using this prompt until the target completions are reached or we exceed the budget of calls (lines 7-13). Each completion c (line 8) is executed on the input table (line 10) using an EXEC procedure, and if the completion is new and its output o satisfies a VALIDATE procedure, the two are accumulated in C and O. The lists of completions and outputs are returned to the user (line 14). We now describe the key steps of the algorithm in more detail.

CLUSTER Consider the input table T in Fig. 5. As is common in data-centric tasks, different rows have slightly format: some only have first and last name, while others have one or two middle names, and some last names are hyphenated. It is important to capture this syntactic variation in the prompt, to make sure that the solution generalizes to all rows (or as many rows as possible). For example, if the LLM were only exposed to rows with two names, it could attempt to extract the last name as the suffix *after the first space*, which would not generalize to rows with more than two names. To capture such syntactic variation, are rely on an existing tool FLASHPROFILE (Padhi et al., 2018), which takes as input a set of strings and synthesizes a set of regular expressions (regexes) from a restricted class, such that each input string matches one of the regexes. In our running example, FLASHPROFILE would synthesize four regexes: `[A-Z][a-z]+[\\s][A-Z][a-z]+`, which matches rows with just two proper-case names, `[A-Z][a-z]+[\\s][A-Z][a-z]+-[A-Z][a-z]+`, which matches rows with dashed last names, and two more (for three and four names, respectively). Then, the output of FLASHPROFILE can be used to cluster input strings.

SELECT If the input table has only one column, like in our running example, selecting representative rows based on the clustering results is trivial: given the budget of n rows to be included into the signature, simply pick one row each from at most n largest clusters. In Fig. 5, assuming our budget

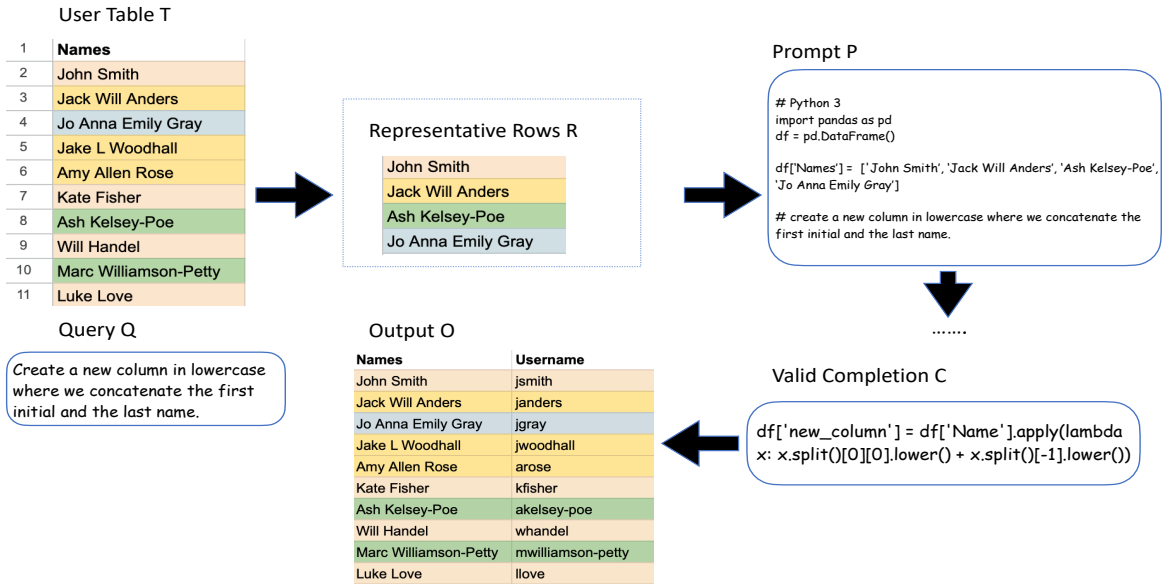

Figure 5: An example run, with the input table T and query Q. The tool extracts the four most representative rows R and uses them, along with the query Q, to create a prompt to pass to the model. The first valid completion is used to create an output column, which is shown to the user.

is $n = 5$, we pick one row from each of the four clusters generated by FLASHPROFILE (depicted with different colors), leading to four representative rows. In case the input contains more than one column, FLASHPROFILE might cluster different columns differently. In this case we would like to select n rows that together cover as many strings as possible across all the columns. We frame this as a *weighted maximal coverage problem*— a well-known NP-complete problem (max) that can be solved approximately using the greedy algorithm described in Algorithm 2. The algorithm takes as input the table T, a map M from the rows of the table to the set of clusters covered by the element in each column of the row. It also takes as input the row budget n. The algorithm iterates over all the rows in T not already in R (line 3) and in each iteration selects the row whose elements maximize the size of the clusters covered (line 4), adding this row to R.

Algorithm 2 Rows Coverage Algorithm SELECT

1: **procedure** SELECT(T, n, M)
2: **while** $|R| < n$ **do**
3: **for** $r \in T_r \wedge r \notin R$ **do**
4: BEST $\leftarrow$ **argmax**$(\sum \{ |c_i| \text{ s.t. } c_i \in M[r] \})$
▷ greedily increase coverage
5: $R \leftarrow R \cup$ BEST
return R

PROMPT Prompt creation procedure PROMPT creates a textual prompt by concatenating the NL query and the representative rows of T. The selected rows R are in the form of a Pandas dataframe. A concrete example of the resulting prompt is shown in Fig. 5.

LLM The completion procedure LLM queries GPT-4 (or another code-generating model), passing the prompt P and predefined stop sequences. We use stop sequences that we have found to allow the LLM to generate at least one solution while typically not using the entire token budget. Note that the LLM needs to produce multiple completions, because it will filter out invalid completions, which are not considered at all. A naive approach would be to request a single completion, validate it, and repeat the process until k distinct valid completions are obtained; this, however, requires sending the prompt to the LLM every time, which incurs a monetary cost. An alternative approach is to *batch* the completions, *i.e.* request some number b of completions in parallel; if the batch size b is too large, however, this also incurs unnecessary cost, since we are requesting more output tokens than we need.

Further details are available in Appendix D.4.

EXEC The procedure EXEC turns each LLM completion into a stand-alone executable program and runs it to obtain the final output o. There are two main challenges to be addressed in this step. First, LLM completions do not have a consistent way

of identifying the final output: for example, the last line of the completion might be an expression that computes the output, or an assignment to a "result" variable, or a print statement. So our tool uses a predefined set of rewrite rules, which we developed by analyzing the patterns in completions. The second challenge is that executing arbitrary LLM-generated code poses a security risk; for this reason, we execute completions in a sandbox. Further details are available in Appendix D.5.

VALIDATE Finally, the procedure VALIDATE checks that the output value o is a dataframe with the right dimensions. The completions that executed without runtime errors during EXEC and passed the output validation are deemed *valid* and are presented to the user. Further details are available in Appendix D.6.

D Experimental Setup

D.1 Evaluation Parameters

We report all results with GPT-4 as the LLM with a temperature of 0.5. We also do a performance comparison for no-data, first-row and full-data regimes and the different selection strategies with CODELLAMA (Roziere et al., 2023) as the LLM.

D.2 Evaluation Metrics

The probability that at least one of k inferred outputs is correct is called pass@k (Chen et al., 2021). More formally, pass@k is the probability that with a sample of k code completions, at least one is correct. To measure this probability empirically for each datapoint, we compute up to m valid programs by sampling from GPT-4 or CODELLAMA. We count the number s of correct completions, and hence compute an estimate of pass@k as $1 - \binom{m-s}{k} / \binom{m}{k}$ (Chen et al., 2021). By computing $m > k$ completions the estimate has lower variance than by simply computing k completions. Each pass@k on a whole dataset is the average of pass@k over all its datapoints. All evaluation results are averaged over tasks, computing m valid completions to estimate pass@k or pass@$k(X\%)$. In practice, we set $m = 20 * k$ when we report results for $k = 1$ or $k = 5$.

D.3 Prompt Template

For each task, we generate prompts according to the data regimes and selection strategies as described above. Below is an example prompt for the query "Create a new column with the difference in hours, minutes and seconds between the two timestamps in the format HH:MM:SS" with one selected row:

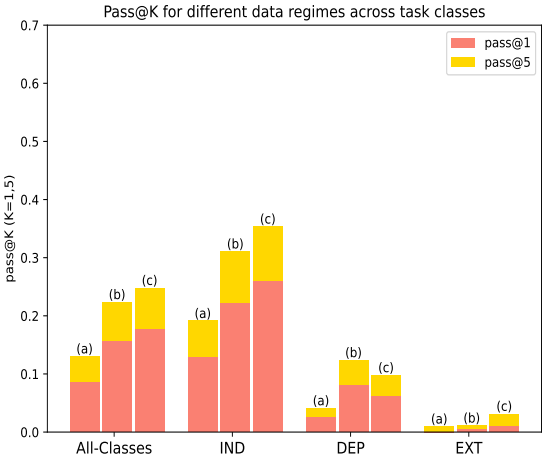

Figure 6: pass@k (CODELLAMA) with (a) no-data, (b) first-row, and (c) full-data (10 rows) passed to the model. The leftmost group of bars represent pass@k with all classes followed by separate pass@k for IND, DEP and EXT tasks. Smaller models have a huge performance drop. But the trend of performance improving with the amount of data passed to the model is seen.

```
# Python 3
import pandas as pd
df = pd.DataFrame()
df['Start'] = ['2/22/2015 1:06:20 PM']
df['End'] = ['2/23/2015 3:08:20 PM']

# Create a new column with the difference
in hours, minutes and seconds between the
two timestamps in the format HH:MM:SS"
```

D.4 Completions Generation

Parallelization. For efficiency, we request multiple completions from GPT-4 per iteration in Alg.1. To try to minimize both inference time and the load on OpenAI's servers, we adapt the batch size to an estimate of the probability that the next completion is valid. The batch size used in each iteration is $n = \min(\lceil r/p \rceil, B, L)$, where $r = k - |C|$ is the number of valid completions still to obtain, B is the remaining completion budget, and L is a parallelization limit enforced by the CHATGPT API. The probability estimate p is updated after each iteration by counting the number of valid and invalid completions in that iteration's batch.

Since pass@k is calculated only from valid completions, it is not influenced by either parallelization or batch size adaptation. We additionally report the average "pool" size (valid and invalid completions) to measure the cost of retrieving valid completions using the above approach in all our

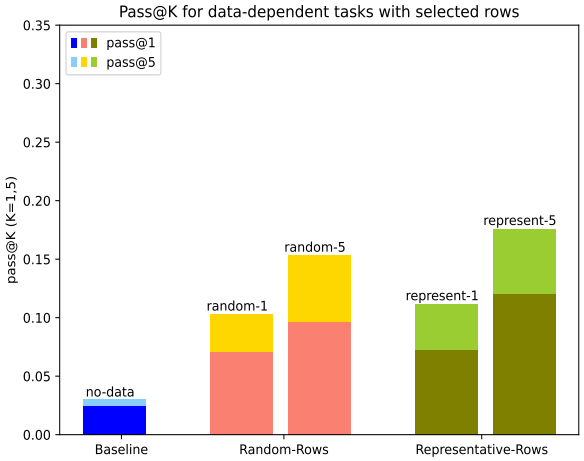

Figure 7: pass@k (CODELLAMA) for DEP tasks with no-data, and n=1 and 5 rows passed to the model, using random (random-n) selection, representative selection (represent-n) and full-data (1000 rows). Completions are evaluated on 1000 rows.

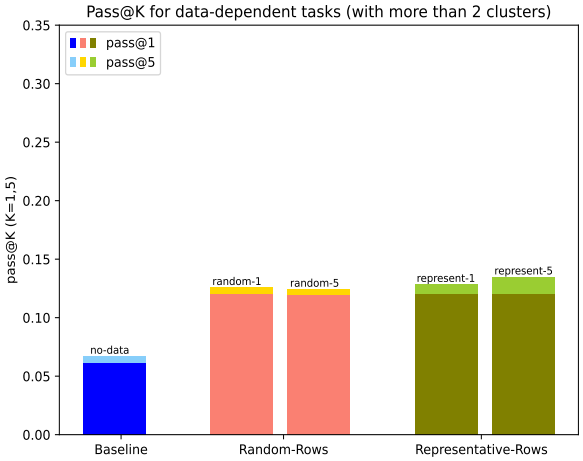

Figure 8: pass@k (CODELLAMA) for 17 out of 44 DEP tasks (more than two clusters) with no-data, random selection (random-n) and representative selection (represent-n). Completions are evaluated on 1000 rows.

experiments.

Stop sequences. The most effective stop sequence we found that allows GPT-4 to generate at least one solution while not usually using the entire token budget is a blank line followed by a line comment; i.e. `\n\n#`. Further, to keep GPT-4 from generating what appears to be the rest of a forum post after a code snippet, we also use the stop sequence `</code>`.

Completion cleanup. Having forum posts apparently in GPT-4's training data means some completions would raise `SyntaxError` exceptions when executed due to formatting artifacts, and therefore be invalid. Instead, to make the most of the completion budget, we replace formatting artifacts. In particular, we replace HTML escape sequences such as `<` and `&quot;` with Python operators and delimiters. Cleanup additionally removes unnecessary whitespace, blank lines and comments, and truncates completions at `\n#` when it appears after executable code.

D.5 Execution of Completions

Rewriting. Completions returned by GPT-4 do not clearly indicate which variables or expressions are intended to be the answer to a query. This must be inferred from the shape of the code. We found that an effective way to identify and expose the likely answer is to search backwards to find the last unindented (i.e. top-level) statement that has one of a few forms, and rewrite the completion so that its last statement is an assignment to a fresh identifier var_{out}. The statement forms and rewrites are

- var `=` $expr$: append the statement var_{out} `=` var to the completion
- var`[`$expr_i$`]` `=` $expr$: append the statement var_{out} `=` var to the completion
- `print(`$expr$, ...`)`: replace this statement and the rest of the completion with var_{out} `=` $expr$
- $expr$: replace this statement and the rest of the completion with var_{out} `=` $expr$

Rewriting also inserts `import` statements for common libraries (e.g. `import numpy as np`). The rewritten completion is appended to the code that defines the input dataframe to create a completed program. The completed program and the output variable name var_{out} are sent to a sandbox for execution.

Sandboxing. Because of security risks inherent in running the LLM-generated code, we run completed programs in a sandbox. Our sandbox is a JavaScript web service that runs Python programs in Pyodide (Droettboom et al., 2022), a Python distribution for WebAssembly. While Python programs running in Pyodide have access to the host's network resources, they at least are isolated from other host resources including its filesystem, offering some level of protection from malicious or accidentally harmful completions. After running the code, the sandbox returns the value of var_{out}.

D.6 Validation of Completions

For a completion to be considered a correct solution in the calculation of pass@k, its actual output must match the expected output. Matching cannot be the same as equality and still conform to a reasonable notion of correctness; for example, the natural breakdown of a solution might generate intermediate columns in the actual output that are not in the expected output. The actual output is allowed to vary from the expected output in the following ways and still match the expected output:

1. Extra columns
2. Different column order
3. Different column headers
4. Number expected; actual is a number within small relative error (default 0.01)
5. Number expected; actual is a string that parses as a number within small relative error
6. Boolean expected; actual is number 0 or 1
7. Boolean expected; actual is a string that represents a truth value
8. String expected; actual is a string that differs only in case

Allowed string truth value representations, allowed relative error, and whether string matching is case-sensitive are (optionally) overridden per data point as appropriate.