CodeXData: Do Code Generating Language Models Understand Data?

Anonymous ACL submission

Abstract

Large language models (LLMs) are effective at code generation. Certain code tasks, such as data wrangling or analysis, can be data-dependent. To study the extent to which code generating models condition on input data, we define two novel data-centric taxonomies that characterise (1) the data required to complete a task and (2) the data available for a given task. Our system CODEXDATA generates Python code under various taxonomy configurations, given an underlying LLM such as CODEX or INCODER. To evaluate CODEXDATA, we curate two new datasets for Python code generation from natural language for data-centric tasks. We evaluate these datasets by varying configurations over our taxonomies and find that performance varies based on the task class, data access, and prompting strategy. This is the first empirical measurement of the impact of data in the NL-to-code setting using LLMs for data-centric tasks.

1 Introduction

Large language models (LLMs) that generate code solutions based on natural language specifications have evolved rapidly and demonstrated effectiveness in practice (Li et al., 2022). Existing evaluations of code generating LLMs, such as OpenAI's CODEX model (Chen et al., 2021a), often include a small number of input-output examples as part of the prompt to the model. However, for data-centric tasks such as those in spreadsheets and computational notebooks, entire tables of data are present and have the potential to positively influence the model behavior. An understanding of the structure of such input data has been exploited in previous program synthesis approaches (Singh, 2016). LLMs have the ability to surpass previous work because *they have access to relevant world knowledge (from training) not present in the prompt*. Our work investigates the impact of data on the effectiveness of LLMs for generating code and provides objective measures of the performance of LLMs for a variety of data-dependent tasks that require a sophisticated understanding of the data context in which they are operating. For instance, the query *"extract the street name from each address"* is under-specified, because without knowing the address format, it is not possible to know which part is the street name. However, if a data sample is given (*e.g. "1024 Turing Way, Boston MA 88588"*), LLMs have the potential to synthesize the correct solution – as the example captures the *data requirement* for this task.

Although prompting with the required data might improve code generation, there are practical considerations that limit *data availability*. Resources such as bandwidth, memory, and time all impose constraints; the application can be limited to a subset of a large dataset, or the prompt can be limited to a small set of tokens. Privacy concerns can restrict the type of data that can be used, such as personally identifiable information, and where or how the data can be transmitted. The data source itself may impose constraints, such as data obtained from ever-changing streams where only a snapshot can be provided to a model. Therefore: *the success of data-centric code generation using LLMs is dependent on both the data required by a given task, and the data available for that task.*

We introduce two new taxonomies for code generation from natural language to capture the nuances of these data requirements. The first concerns the extent to which data is required to fully interpret a task. A task can be *data-independent*, where the query itself provides sufficient information for the model, such as "*return the unique values from column location*". A task can be *data-dependent*, such as extracting the street name from an address (described previously), where the query lacks data-specific information required by the model, such as the input structure or format. Finally, a task can be *external-dependent*, where the

query requires world knowledge outside the task, *e.g.*, "*create a new column that verifies that the year is an election year in the US*".

Our second taxonomy concerns redaction of task data. We focus on queries over tabular data and so we define our taxonomy in terms of the column headers and data rows of a table. Following data redaction, a task has access to either the original column headers or simply an anonymous schema (generated headers such as "Col1"), and none, some, or all of the table rows.

To explore the impact of our taxonomies on code generation, we curate two new datasets. Each datapoint in our datasets consists of a textual query, a data input (column-major-flat table), and an expected correct output (extra columns). Our first dataset SOFSET is sourced from questions on StackOverflow[1] that request help to solve a spreadsheet task. The dataset is challenging because it consists of real-word multi-step data manipulation problems. Additionally, whilst we generate Python pandas code to solve each task, the tasks were formulated in the context of spreadsheets and formulas. We argue that because the tasks were not designed to be solved with Pandas, this presents an additional challenge over existing datasets. To further evaluate model performance on tasks relevant to specific data-types (*e.g.* addresses and dates) and the impact of noise in the data, we create a second synthetic dataset TYPESET with curated types and queries based on questions from StackOverflow and Mr. Excel[2].

Using these datasets and taxonomies we evaluate CODEXDATA, our system to convert natural language into code for querying tabular data. CODEXDATA queries an LLM (CODEX or INCODER (Fried et al., 2022) in our experiments) with a prompt that combines both task description and task data to generate Python code that uses the Pandas API as shown in Figure 1. We evaluate our approach using CODEX because of its state-of-the art performance and practical usage in the code completion tool, GitHub Copilot[3]. We also evaluate the INCODER model (an opensource alternative to CODEX) which is trained from scratch on publicly available code and StackOverflow posts.

Our main contribution is the first empirical study that measures the impact of data on the code generated by LLMs for data-centric tasks. To support this study, we make the following additional contributions:

1. We define two novel data-centric task taxonomies. The first characterises the data required to complete a task. The second characterises the data redacted for a given task.

2. We curate two new datasets to explore the impact of our taxonomies on data-centric code generation. The first dataset is a collection of real spreadsheet problems from StackOverflow. The second dataset is a synthetic benchmark designed to evaluate the impact of different data types and noise on performance. Our datasets have 200 and 64 unique tasks, with a median of 10 and 20 data rows respectively, making them larger and more diverse than prior data-centric datasets.

3. We study how data redaction and prompting strategies affect the performance (pass@k) for different task classes. We find that data redaction can lead to a significant performance drop (as shown in Table 1, depending on the configuration, we can see a drop of over 50% for pass@10), but generally, with just one example row and our prompting method, we restore performance to within 5% of the full data performance (Figure 2).

2 Background and Related work

Code generating LLMs like Codex (Chen et al., 2021a), PaLM (Chowdhery et al., 2022), InCoder (Fried et al., 2022), CodeGen (Nijkamp et al., 2022), CodeT5 (Wang et al., 2021) and AlphaCode (Li et al., 2022) have been trained or fine-tuned for code-specific tasks and can be adapted for data-centric domains such as SQL (Trummer, 2022) and data wrangling (Narayan et al., 2022).

Data-centric Code Generation. We explore the question: *how does data impact code generation for data-centric tasks?* We focus on Pandas tasks with column-major-flat tables to produce new data columns. Related work on data-centric code generation has focused on SQL tasks (Rajkumar et al., 2022) from the Spider benchmark (Yu et al., 2018) or in-place data transformations that do not yield a new dataframe column (Narayan et al., 2022). To evaluate our data-centric approach, we curate two new Pandas datasets with queries and input dataframes. Python datasets like

[1] https://stackoverflow.com/questions/tagged/excel-formula
[2] https://www.mrexcel.com/board/forums/excel-questions.10/
[3] https://github.com/features/copilot/

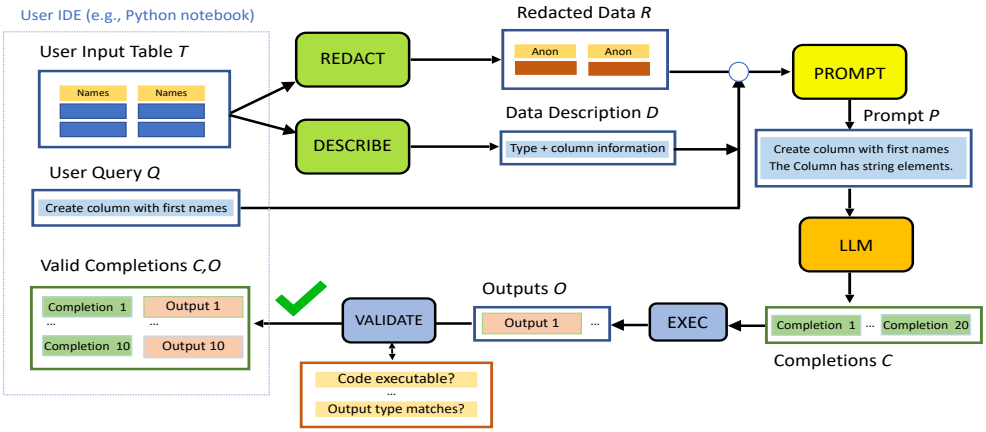


Figure 1: The figure illustrates components of CODEXDATA that transform the user input table and query into a list of valid completions. The data is redacted to anonymize private information. A data description is extracted from the input table. The resulting redacted data, data description and query are used to construct a prompt which is fed to a code synthesis LLM such as CODEX or INCODER, generating multiple possible completions. The outputs of these completions are then validated and the first k valid completions (along with the outputs) are returned.

APPS (Hendrycks et al., 2021) and HumanEval (Chen et al., 2021a)) are not data-centric, while the tasks from prior Pandas datasets (e.g. Jain et al. (2022) or Zan et al. (2022)) do not have comparable complexity and the data available is insufficient for detailed analysis. We also evaluate the impact of data redaction and noisy data on data-centric code-generation. Narayan et al. (2022) explore the role of noisy table attributes in LLM performance for entity matching. They also identify the challenges of data privacy, but do not empirically explore the impact of privacy-related decisions on performance.

Prompt Engineering. Careful prompt engineering for LLMs is effective (Reynolds and McDonell, 2021). We explore prompting techniques specifically for code generation. Instruction-style prompts have been effective for NLP tasks (Mishra et al., 2021). We construct prompts that instruct the model to generate solutions with a certain property, *e.g.*, type-correctness. Prompts based on summarising have also been effective (Kuznia et al., 2022). We propose *data-description prompts* that summarize information about the dataframes in our tasks. Chain-of-thought prompting combined with symbolic reasoning (Gao et al., 2022) could be an interesting future direction.

3 CodeXData: Technical Approach

Algorithm 1 presents our inference algorithm, with an associated overview in Figure 1. It takes as input a *user query* Q as text, *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 CODEX ($k_{max} = 8k$). For our example, Q is "create a new column with the first names", T is `Data({"Name":["Jay M","Emma G"]})`, and k is 10.

Algorithm 1 CodeXData Inference Algorithm

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

1: **procedure** CODEXDATA(Q, T, k)
2: $R \leftarrow$ REDACT(T) ▷ redact the input table
3: $D \leftarrow$ DESCRIBE(T) ▷ extract data description
4: $P \leftarrow$ PROMPT(Q, R, D) ▷ 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)

At a high-level the algorithm *redacts* the input table (line 2) using REDACT; extracts a data *description* of the table using DESCRIBE (line 3); combines the query Q, redacted data R, and data description D to create a prompt P using PROMPT (line 4); queries LLM repeatedly using this prompt until the target completions are reached or we exceed the budget of calls (lines 6-12). Each comple-

tion c (line 7) is executed on the input table (line 9) 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 13). We describe the key components.

REDACT. A data redactor procedure REDACT creates redacted data R from the input table T. For our running example, redacted data R in the no data (+ anonymous headers) and no data (+ actual headers) cases would be `Data({"Anon":[]})` and `Data({"Names":[]})` resp. REDACT can alternatively provide a subset of rows as R = `Data({"Names":["Jay M"]})` or the full dataframe with no redaction.

DESCRIBE. A descriptor procedure DESCRIBE extracts a data description D from the input table T. The description can consist of: number of elements in a column, column type or basic predicates satisfied by all the column elements (*e.g.* all lowercase). It can also be a regular expression that captures a pattern supported by a substantial portion of the column elements (Padhi et al., 2018), *e.g.*, *all elements in column Names match the regex format [A-Z][a-z]+ [A-Z]+*.

PROMPT. A prompt creation procedure PROMPT creates a textual prompt by combining the user's query, redacted data, and data description. The baseline prompt concatenates redacted data R, and textual query Q. We consider three more strategies for creating a prompt. *Instruction prompts* (Mishra et al., 2021) append instructions to the baseline prompt, and emphasize that the completion should have a particular property: *write an executable and type correct pandas solution for the following query: create a new column with the first names* (Table 2 in Appendix). *Example-usage* prompts add concrete Pandas examples in a few-shot format. They prepend to the baseline prompt examples that emphasize a property (See Table 3 in Appendix). *Data-description* prompts augment the baseline prompts with the data description D. For example, if D is column type information, the prompt would be: *create a new column with the first names. Returns pd.Dataframe with new columns. Data columns are as follows: First Names (as str)* (Table 4 in Appendix).

LLM. The completion procedure LLM queries CODEX's Da Vinci engine (or any other code generating LLM, *e.g.* INCODER), passing the prompt P, a temperature derived from k, 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. The completion then undergoes cleanup, such as truncating at the first comment after executable code, and removing whitespace and comments. Further details are in Appendix E.5.

EXEC. Execution procedure EXEC constructs an executable program by (1) rewriting the completion c into c' to expose the likely answer by finding a top-level statement that has a particular form and assigning it to a fresh variable and (2) appending the rewritten completion c' to the code that defines the input table T to create a completed program. It executes the program in a sandbox to obtain the final output o. Further details are in Appendix E.6.

VALIDATE. We validate the extracted output (*e.g.*, for expected output type) using an *output validator* VALIDATE. The completions that pass the validator, *i.e. valid completions*, are returned to the user.

4 Experiments

We design our experiments to study how model performance changes with (a) level of data redaction, (b) prompting strategy, and (c) noise level of the data, across the three task classes in our taxonomy. For redaction, we consider no-data access (with anonymized and properly named column headers), full-data access and subset-data access (in all our experiments we use the first row only as subset). The task classes are of increasing difficulty in terms of data required: data-independent (IND), data-dependent (DEP) and external-dependent (EXT). We evaluate how prompting can bridge the gap between data required and the data available. In particular, we evaluate the performance of the data-description prompting strategies to understand whether data description information in the query can help us mitigate the impact of data redaction. We also investigate the performance of the three prompting strategies: instruction, example-usage and data-description in presence of full data.

4.1 Task Classes

Data-independent tasks. These tasks can be solved using the query alone without any data access. For example, the query "create a new column that counts how many holidays (7/3/22, 8/2/22, 15/10/22) are between the dates in Start Date and End Date" can be solved without knowledge of the

actual data. Pandas offers functions that can operate on default formats like "dd/mm/yy". If the date format was not directly specified, the model would need to *access* the data to *infer* the correct format.

Data-dependent tasks. These tasks cannot be solved using the query alone: the model needs to have access to the data. For instance, 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. For example, 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.

4.2 Our Datasets

We curate two novel Pandas datasets, SOFSET and TYPESET, which consist of real-world problems across diverse domains to answer our research questions. Our datasets are larger and more diverse than either of two existing Pandas datasets, JIGSAW (Jain et al., 2022) and CERT (Zan et al., 2022). In addition to the three task classes, we classify the tasks based on data types (strings, numbers, names, dates, units, addresses and mixed types) that are popular for data-centric tasks (Table 5). All the datasets in the paper were collected and manually annotated by the authors.

SOFSET. This dataset is a collection of 200 real-world tasks from StackOverflow where the user has asked about spreadsheets. We sample these tasks deterministically from the highest rated posts in StackOverflow (as of March 2022) after we filter with the tag "ExcelFormulas". These tasks are representative of real problems users have since they correspond to the highest rated posts. We manually check that the posts are genuine tasks. Tasks in this dataset, especially those in the DEP and EXT task classes, represent complex problems.

Each task in the dataset consists of a query, data and hand-annotated metadata. We manually write the queries, either from scratch or summarise the verbose ask into a concise query based on the original StackOverflow description. We either use the data the user has added to their query, or if it is ambiguous, we create new data that matches the query. We also add extra rows and corner cases to make the data have at least 10 rows. The metadata we add includes column names, type of query, type of data access, question type and type of column referencing. We provide the expected output (values in the newly generated column). We remove post identifiers for anonymization.

TYPESET. We construct a synthetic dataset TYPESET with type-specific tasks to further evaluate performance across the three task classes. This dataset is a collection of 64 tasks mostly over date and address types. The tasks consist of synthetic queries that are common for each type (*e.g.* for type date - extract the year from a date) paired with type-specific data columns sourced from Sherlock (Hulsebos et al., 2019), AutoType (Yan and He, 2018) and Wikidata (Vrandecic and Krötzsch, 2014). We manually annotate the queries along with Pandas solutions and expected outputs.

TYPESETNOISY. We create a noisy version of the TYPESET dataset with 131 tasks to evaluate how performance changes in the presence of noisy data. Specifically, we alter approximately 20% of the values in the input columns. We induce four noisy scenarios: **corruption**: inserts a space, dash, or a new line at a random position in the input strings; **missing**: replaces values in an input column with an empty string; **mixformat**: changes column formats (for *e.g.* mixes ints and floats or combine different date/address formats); and **additional columns**: adds redundant anonymous and appropriately named columns based on the column type of the data frame. The authors observed these scenarios commonly occur in StackOverflow questions used for the dataset creation[4]. Further, similar robustness measures have been used in Text-to-SQL parsing (Pi et al., 2022) against adversarial table perturbations.

4.3 Evaluation Metrics

We report correctness based on whether the generated code produces the expected output. Consider a single datapoint consisting of a query Q, an input table T, and an expected correct output o. The probability that at least one of k inferred outputs is correct is called pass@k (Chen et al., 2021a). More formally, pass@k is the probability that $o \in O$ if we sample (C, O) from $\text{CODEXDATA}(Q, T, k)$. To measure this probability empirically for each datapoint, we compute up to $2k$ valid programs by sampling (C, O) from

[4] https://support.microsoft.com/en-us/office/top-ten-ways-to-clean-your-data-2844b620-677c-47a7-ac3e-c2e157d1db19

CODEXDATA$(Q, T, 2k)$. We count the number s of occurrences of o in O, and hence compute an estimate of pass@k as $1 - \binom{2k-s}{k} / \binom{2k}{k}$ (Chen et al., 2021a). By computing $2k$ 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.

We also calculate a partial form of the metric written as pass@$k(X\%)$. It is defined similarly to pass@k except the correctness of an output column depends on the $X\%$ of inferred output that matches the expected output. So, pass@$k(100\%)$ is the same as pass@k. This partial metric is required to distinguish solutions that fail on an edge case from solutions that are inherently wrong (*e.g.* reference the wrong column or API). We conduct a sensitivity analysis of pass@k and report observed standard deviations for each task class with and without data redaction (Appendix Figure 15).

4.4 Empirical Insights

We assess performance across tasks for different data redactions and prompting strategies using the datasets SOFSET and TYPESET. We use anonymized versions of column names for the no-data (anonymous columns) redaction and proper column names for the other three redactions. We report results for CODEXDATA with CODEX and INCODER-6B (Fried et al., 2022) as the underlying LLMs. The prompting experiments are conducted with full-data access unless specified otherwise. All evaluation results are averaged over tasks (and correspond to a single CODEXDATA run), computing $2k$ valid completions to estimate pass@k or pass@$k(X\%)$. In the appendix, we report stability across several runs and detailed error analyses including invalid completions.

Performance varies with the underlying LLM. Table 1 shows that in general CODEX performs considerably better than INCODER across all task categories, datasets and redaction levels. The gap for IND tasks in SOFSET is consistent with prior results (Fried et al., 2022). We hypothesize that the gap between LLMs in the DEP and EXT tasks is due to CODEX having enhanced world knowledge. However, the tasks in SOFSET EXT are too difficult for either model to address successfully.

The level of data redaction has a high impact on pass@k. Table 1 shows that in most settings, pass@k drops when the data redaction is stricter, *i.e.* pass@k is highest in the full data case (d). The performance degradation from no redaction (d) to partial redaction (only first row (c)) is relatively small in most cases, which implies that sampling rows is almost as effective as providing full data.

Performance varies with dataset complexity. Tasks in SOFSET are more complex than in TYPESET because of the difficulty of the user query (e.g. involves multi-step computations) and number of corner cases in the data. In particular, in DEP and EXT tasks the complexity gap between datasets is high which directly translates to a big performance gap as seen in Table 1. The pass@k for DEP and EXT tasks is higher than IND tasks for TYPESET— we attribute this to the uneven distribution of problems across classes and data-types. For example, there are more IND tasks of data-type address which is the most difficult category. The x-axis in Figure 2 represents how pass@10 varies with percentage of examples passed. Interestingly, we see a sharper decrease in pass@10($X\%$) in Figure 2 for SOFSET compared to TYPESET due to presence of complex corner cases in the former.

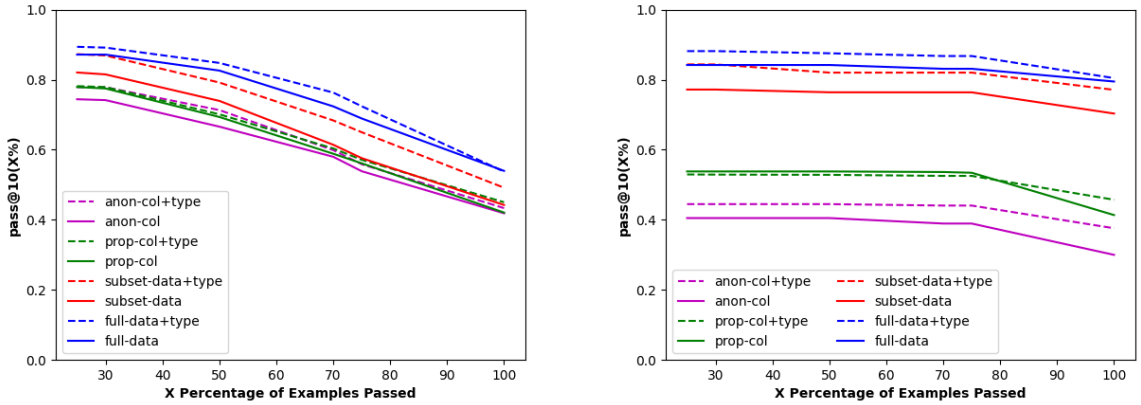
Data-description prompts counter data redaction. Figure 2 shows the interplay of data descriptions in the prompt under different data redactions. For each pair of same colored lines, the dashed line represents the result when a data description (here type information) is added to the original prompt. The data-description prompt improves performance in all cases but is particularly valuable in the cases where (a) a subset of the data is present or (b) all data is removed and columns anonymized, and helps more in (a). So we can use data-description prompts to counter data redaction. (INCODER plots in Appendix Figure 14).

Careful prompt engineering helps. We conduct more prompting experiments (using PROMPT from section 3). The full results for both CODEX and INCODER, are in (Figure 10, Figure 11, Figure 12) in the Appendix. Figure 3 presents the prompting strategies that outperform baseline for CODEX. We see that carefully extracting the data description (in the form of regular expressions, number of column elements and type information) helps for both datasets. Interestingly, instructing the model to be concise, executable and generate iterative solutions also improves performance across the two datasets. The best performing strategy for SOFSET (regular expressions + type information) has a pass@10 of 0.56 compared to the baseline pass@10 of 0.53 (5.6%). TYPESET has a

SOFSET		Data-Independent (IND)			Data-Dependent (DEP)			External-Dependent (EXT)		
Model	Data Redaction	#	pass@5	pass@10	#	pass@5	pass@10	#	pass@5	pass@10
CODEX	(a) no data, anon names	133	0.50	0.57	37	0.08	0.14	31	0.08	0.11
	(b) no data, prop names	132	0.50	0.58	38	0.08	0.12	31	0.05	0.10
	(c) subset data	132	0.55	0.60	38	0.15	0.14	31	0.13	0.14
	(d) full data	132	0.61	**0.67**	38	0.19	**0.30**	31	0.17	**0.21**
INCODER	(a) no data, anon names	133	0.05	0.05	37	0.06	0.05	31	0.00	0.00
	(b) no data, prop names	132	0.16	0.17	38	0.12	**0.12**	31	0.13	**0.20**
	(c) subset data	132	0.10	0.18	38	0.08	0.08	31	0.12	0.13
	(d) full data	132	0.17	**0.20**	38	0.08	0.08	31	0.10	0.14

TYPESET		Data-Independent (IND)			Data-Dependent (DEP)			External-Dependent (EXT)		
Model	Data Redaction	#	pass@5	pass@10	#	pass@5	pass@10	#	pass@5	pass@10
CODEX	(a) no data, anon names	15	0.44	0.49	13	0.31	0.38	36	0.23	0.25
	(b) no data, prop names	19	0.40	0.42	8	0.22	0.35	37	0.38	0.43
	(c) subset data	19	0.52	**0.62**	8	0.76	0.81	37	0.61	0.76
	(d) full data	19	0.52	0.58	8	0.78	**0.92**	37	0.78	**0.83**
INCODER	(a) no data, anon names	15	0.43	0.50	13	0.34	0.35	36	0.10	0.11
	(b) no data, prop names	19	0.36	0.36	8	0.24	0.24	37	0.21	**0.23**
	(c) subset data	19	0.46	0.47	8	0.48	0.50	37	0.14	0.17
	(d) full data	19	0.42	**0.50**	8	0.58	**0.62**	37	0.20	0.21

Table 1: Impact on pass@k with data redaction for SOFSET (top) and TYPESET (bottom) for both CODEX and INCODER. The results are grouped by different task classes: IND, DEP and EXT. # indicates the number of examples (anonymization can change task class).



SOFSET TYPESET

Figure 2: Interplay between the varying amount of data redaction and how data description in the prompt helps with performance for CODEX. For each redaction, we evaluate the performance by adding a data description to the prompt in the form of type information. Type information is especially helpful when we only have access to anonymous columns and a subset of the data. The extracted data descriptions do not depict private information.

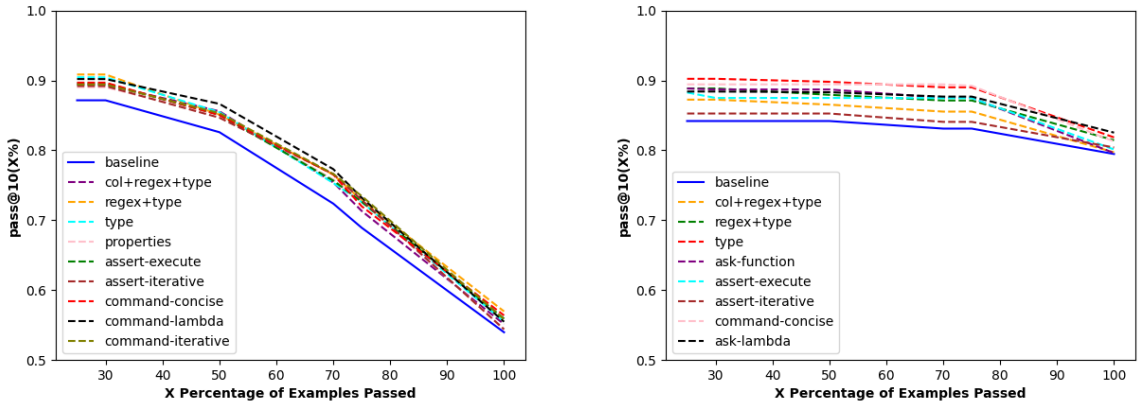
relative performance improvement of 4%.

Prompting can also sometimes hurt as seen in case of regex (subset). This prompt adds too much data description (one regex per element) which likely ends up confusing the model. So prompts can be beneficial but only when curated carefully.

Noisy data impacts performance. We evaluate the impact of noise in TYPESETNOISY. Addition of approximately 20% noise to the data leads to a drop in pass@10(75%) and higher (Figure 4). INCODER is more affected by noise (Appendix Figure 17). This is consistent with prior insight from text-to-SQL tasks (Pi et al., 2022).

Performance varies considerably per data-type. Performance across data-types differs in pass@k, length of completions, number of retrieved completions, and the extent to which those completions are executable. Tasks with string data-type have an average pass@k of 0.83 compared to 0.48 for tasks with dates data-type. Overall, the model performs best on units tasks, worst on addresses (Tab 14 in Appendix).

Generation efficiency depends on redaction and task class. We compute the number of total completions required by CODEX to produce 20 valid completions (as determined by the VALIDATE procedure). We perform this experiment on TYPESET, across task classes and redaction levels. Figure 5 shows the pass@10 performance as a function of total completion count. Irrespective of



SOFSET TYPESET

Figure 3: Impact of prompting strategies on SOFSET and TYPESET. We present a subset of prompting strategies that perform better than the baseline, full results in Figure 10 and Figure 12. Data-description prompts with type information and regular expression patterns that match column elements outperform the baseline for both datasets.

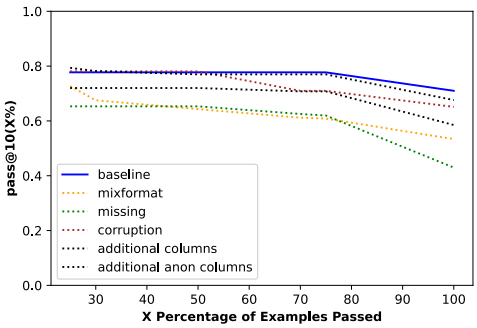


Figure 4: Impact pass@10($X\%$) for the different noisy scenarios (e.g. corruption) in TYPESETNOISY compared to the original TYPESET (baseline) for CODEX.

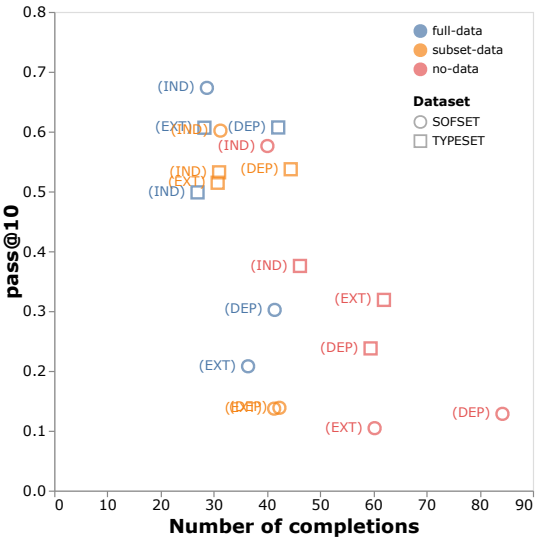


Figure 5: Interplay between pass@10 and total number of retrieved completions from CODEX for the different datasets (shapes), degree of data redaction (color) and task classes (labels). **Lower right** (Low pass@10 and many completions): most results for runs with highly redacted data and DEP tasks lie here. **Upper left** (High pass@10 and few completions): most results for runs with no redaction and/or IND tasks lie here. **Lower left-/middle** (Low pass@10 and few completions): most results for runs for EXT tasks lie here.

task class, access to data greatly reduces the number of generations needed; for example, with full-data we need fewer than 50 completions. We need more completions for data-dependent tasks than data-independent tasks. In general, sampling more completions results in a poorer pass@10. (INCODER plots in Appendix Figure 16).

5 Conclusion and Future Work

Our work highlights the importance of data for code generation on data-centric tasks. We empirically show that data influences both the system's capability to generate good solutions and its efficiency, and that optimizing prompts improves performance. Our study is a first step towards understanding the relationship between code generation and data in LLMs. As a result, we scoped our exploration to enable a focused analysis. Handling a broader problem space (e.g., multi-table inputs, hierarchical table inputs, new table outputs) raises interesting challenges in how to best encode the data. We hypothesise that the impact of data redaction can not be fully eliminated so it would be interesting to investigate how to integrate with approaches to privacy preservation.

References

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. 2021a. Evaluating large language models trained on code.

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. 2021b. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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).

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida I. Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *ArXiv*, abs/2204.05999.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2022. Pal: Program-aided language models. *arXiv preprint arXiv:2211.10435*.

Patrick M. Haluptzok, Matthew Bowers, and Adam Tauman Kalai. 2022. Language models can teach themselves to program better. *ArXiv*, abs/2207.14502.

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.

Madelon Hulsebos, Kevin Hu, Michiel Bakker, Emanuel Zgraggen, Arvind Satyanarayan, Tim Kraska, Çağatay Demiralp, and César Hidalgo. 2019. Sherlock: A deep learning approach to semantic data type detection. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM.

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)*.

Kirby Kuznia, Swaroop Mishra, Mihir Parmar, and Chitta Baral. 2022. Less is more: Summary of long instructions is better for program synthesis. *arXiv preprint arXiv:2203.08597*.

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*.

Swaroop Mishra, Daniel Khashabi, Chitta Baral, Yejin Choi, and Hannaneh Hajishirzi. 2021. Reframing instructional prompts to gptk's language. *arXiv preprint arXiv:2109.07830*.

Avanika Narayan, Ines Chami, Laurel Orr, and Christopher R'e. 2022. Can foundation models wrangle your data? *ArXiv*, abs/2205.09911.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Haiquan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *ArXiv*, abs/2203.13474.

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.

Xinyu Pi, Bing Wang, Yan Gao, Jiaqi Guo, Zhoujun Li, and Jian-Guang Lou. 2022. Towards robustness of text-to-SQL models against natural and realistic adversarial table perturbation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2007–2022, Dublin, Ireland. Association for Computational Linguistics.

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

Laria Reynolds and Kyle McDonell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–7.

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.

Rishabh Singh. 2016. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proceedings of the VLDB Endowment*, 9(10):816–827.

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

Denny Vrandecic and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

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.

Cong Yan and Yeye He. 2018. Synthesizing type-detection logic for rich semantic data types using open-source code. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 35–50, New York, NY, USA. Association for Computing Machinery.

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 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., 2021a), AlphaCode (**?**), and PaLM (Chowdhery et al., 2022). These impacts include overreliance, 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.

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 'understand' 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. These benchmarks aim to be reflective of a qualitative notion of utility, but as with all quantitative benchmarks the process of operationalising a qualitative notion inevitably requires some reductionism.

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, which checks for compatibility with Microsoft's Responsible AI standard.[5]

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.

[5] https://www.microsoft.com/en-us/ai/responsible-ai

B Datasheets for Datasets

B.1 MOTIVATION

For what purpose was the dataset created? Was there a specific task in mind? Was there a specific gap that needed to be filled? Please provide a description.
Answer: We created the new dataset to be able to evaluate how well LLMs perform on data-centric tasks when they are given various degrees of access to data. LLMs have been evaluated on Python generations in prior work - on puzzles, e.g. (Haluptzok et al., 2022), on programming contests, e.g. (Li et al., 2022), on Python problems, e.g. (Chen et al., 2021b) or (Hendrycks et al., 2021), and even Pandas tasks, e.g. (Zan et al., 2022) or (Jain et al., 2022). We are contributing this dataset as it contains larger dataframes and more diverse tasks than prior work, allowing us to draw meaningful conclusions for our research questions.

Who created this dataset (e.g., which team, research group) and on behalf of which entity (e.g., company, institution, organization)?
Answer: to be added after ARR reviews to not break anonymity

What support was needed to make this dataset? (*e.g.* who funded the creation of the dataset? If there is an associated grant, provide the name of the grantor and the grant name and number, or if it was supported by a company or government agency, give those details.)
Answer: to be added after ARR reviews to not break anonymity

B.2 COMPOSITION

What do the instances that comprise the dataset represent (e.g., documents, photos, people, countries)? Are there multiple types of instances (e.g., movies, users, and ratings; people and interactions between them; nodes and edges)? Please provide a description.
Answer: Queries and data from StackOverflow or synthetic data.

How many instances are there in total (of each type, if appropriate)?
Answer: We have collected a benchmark of 780 datapoints.

Does the dataset contain all possible instances or is it a sample (not necessarily random) of instances from a larger set? If the dataset is a sample, then what is the larger set? Is the sample representative of the larger set (e.g., geographic coverage)? If so, please describe how this representativeness was validated/verified. If it is not representative of the larger set, please describe why not (e.g., to cover a more diverse range of instances, because instances were withheld or unavailable).
Answer: Part of the dataset (SOFSET) is a sample from all the queries in Stackoverflow under the tag "ExcelFormula". The dataset was sampled such that we prioritize highly rated questions/answers, so we believe they are representative for real problems users have. TYPESET was synthetically created with queries that are common for each type (e.g. extracting the year from a date) - as the dataset has fine-grained types (e.g. names and addresses) we have a small set of possible queries available.

What data does each instance consist of? "Raw" data (e.g., unprocessed text or images) or features? In either case, please provide a description.
Answer: Each datapoint comprises of a query (e.g. "count the number of clothing items"), data (e.g. a table of products and sales) and hand-annotated metadata (e.g. column names, type of query, type of data access).

Is there a label or target associated with each instance? If so, please provide a description.
Answer: We provide the expected output (values in the newly generated column) and for TYPESET we also provide a Pandas solution.

Is any information missing from individual instances? If so, please provide a description, explaining why this information is missing (e.g., because it was unavailable). This does not include intentionally removed information, but might include, e.g., redacted text.
Answer: No

Are relationships between individual instances made explicit (e.g., users' movie ratings, social network links)? If so, please describe how these relationships are made explicit.
Answer: No

Are there recommended data splits (e.g., training, development/validation, testing)? If so, please provide a description of these splits, explaining the rationale behind them.
Answer: This dataset is hand-curated for evaluation only.

Are there any errors, sources of noise, or redundancies in the dataset? If so, please provide a description.
Answer: Part of the dataset has been augmented (the data has been corrupted) to help us explore the impact of naturally occuring noise in users' work. We added missing data, corrupted data values, surplus columns and mixed formats.

Is the dataset self-contained, or does it link to or otherwise rely on external resources (e.g., websites, tweets, other datasets)? If it links to or relies on external resources, a) are there guarantees that they will exist, and remain constant, over time; b) are there official archival versions of the complete dataset (i.e., including the external resources as they existed at the time the dataset was created); c) are there any restrictions (e.g., licenses, fees) associated with any of the external resources that might apply to a future user? Please provide descriptions of all external resources and any restrictions associated with them, as well as links or other access points, as appropriate.
Answer: Self-contained

Does the dataset contain data that might be considered confidential (e.g., data that is protected by legal privilege or by doctor-patient confidentiality, data that includes the content of individuals' non-public communications)? If so, please provide a description.
Answer: No

Does the dataset contain data that, if viewed directly, might be offensive, insulting, threatening, or might otherwise cause anxiety? If so, please describe why.
Answer: No

Does the dataset relate to people? If not, you may skip the remaining questions in this section.
Answer: No

Does the dataset identify any subpopulations (e.g., by age, gender)? If so, please describe how these subpopulations are identified and provide a description of their respective distributions within the dataset.
Answer: No

Is it possible to identify individuals (i.e., one or more natural persons), either directly or indirectly (i.e., in combination with other data) from the dataset? If so, please describe how.
Answer: No

Does the dataset contain data that might be considered sensitive in any way (e.g., data that reveals racial or ethnic origins, sexual orientations, religious beliefs, political opinions or union memberships, or locations; financial or health data; biometric or genetic data; forms of government identification, such as social security numbers; criminal history)? If so, please provide a description.
Answer: No

B.3 COLLECTION

How was the data associated with each instance acquired? Was the data directly observable (e.g., raw text, movie ratings), reported by subjects (e.g., survey responses), or indirectly inferred/derived from other data (e.g., part-of-speech tags, model-based guesses for age or language)? If data was reported by subjects or indirectly inferred/derived from other data, was the data validated/verified? If so, please describe how. *Answer*: Directly observable

Over what timeframe was the data collected? Does this timeframe match the creation timeframe of the data associated with the instances (e.g., recent crawl of old news articles)? If not, please describe the timeframe in which the data associated with the instances was created. Finally, list when the dataset was first published.
Answer: The StackOverflow posts date back to 2006 and we curated the dataset throughout 2022.

What mechanisms or procedures were used to collect the data (e.g., hardware apparatus or sensor, manual human curation, software program, software API)? How were these mechanisms or procedures validated?
Answer: We manually sourced the datapoints and manually annotated with metadata.

What was the resource cost of collecting the data? (e.g. what were the required computational resources, and the associated financial costs, and energy consumption - estimate the carbon footprint. See Strubell *et al.*(?) for approaches in this area.)
Answer: The costs were negligible (no computation cost) and the annotations were sourced among the research group.

If the dataset is a sample from a larger set, what was the sampling strategy (e.g., deterministic, probabilistic with specific sampling prob-

abilities)?
Answer: Deterministic - we sampled based on highest rated posts in StackOverflow after filtering to the tag ExcelFormula and checking manually that the posts were genuine tasks (not related to Excel behaviour).

Who was involved in the data collection process (e.g., students, crowdworkers, contractors) and how were they compensated (e.g., how much were crowdworkers paid)?
Answer: Researchers in the team (no extra compensation).

Were any ethical review processes conducted (e.g., by an institutional review board)? If so, please provide a description of these review processes, including the outcomes, as well as a link or other access point to any supporting documentation.
Answer: Yes, the dataset passed our institution's on-boarding and publishing reviews

Does the dataset relate to people? If not, you may skip the remainder of the questions in this section.
Answer: No

Did you collect the data from the individuals in question directly, or obtain it via third parties or other sources (e.g., websites)?
Answer: Third party (StackOverflow)

Were the individuals in question notified about the data collection? If so, please describe (or show with screenshots or other information) how notice was provided, and provide a link or other access point to, or otherwise reproduce, the exact language of the notification itself.
Answer: No, all data was public and we didn't infringe any licensing

Did the individuals in question consent to the collection and use of their data? If so, please describe (or show with screenshots or other information) how consent was requested and provided, and provide a link or other access point to, or otherwise reproduce, the exact language to which the individuals consented.
Answer: Not applicable

If consent was obtained, were the consenting individuals provided with a mechanism to revoke their consent in the future or for certain uses? If so, please provide a description, as well as a link or other access point to the mechanism (if appropriate)
Answer: Not applicable

Has an analysis of the potential impact of the dataset and its use on data subjects (e.g., a data protection impact analysis)been conducted? If so, please provide a description of this analysis, including the outcomes, as well as a link or other access point to any supporting documentation.
Answer: No

B.4 PRE-PROCESSING / CLEANING / LABELING

Was any preprocessing/cleaning/labeling of the data done(e.g.,discretization or bucketing, tokenization, part-of-speech tagging, SIFT feature extraction, removal of instances, processing of missing values)? If so, please provide a description. If not, you may skip the remainder of the questions in this section.
Answer: We manually created the queries (either from scratch or based on the original StackOverflow description), we augmented the data (we added more rows, corner cases, augmentations), we extracted metadata (e.g. questions type, type of column referencing).

Was the "raw" data saved in addition to the preprocessed/cleaned/labeled data (e.g., to support unanticipated future uses)? If so, please provide a link or other access point to the "raw" data.
Answer: No.

Is the software used to preprocess/clean/label the instances available? If so, please provide a link or other access point.
Answer: We will provide access to the software we built for data processing, but the majority of the work carried involved manual annotations.

B.5 USES

Has the dataset been used for any tasks already? If so, please provide a description.
Answer: The current work is the first use of the dataset.

Is there a repository that links to any or all papers or systems that use the dataset? If so, please provide a link or other access point.
Answer: Not applicable

What (other) tasks could the dataset be used for?
Answer: The dataset could be used the evaluate code generation for other languages (*e.g.* R or SQL) on data-centric tasks, query generation (from the code).

Is there anything about the composition of the dataset or the way it was collected and preprocessed/cleaned/labeled that might impact future uses? For example, is there anything that a future user might need to know to avoid uses that could result in unfair treatment of individuals or groups (e.g., stereotyping, quality of service issues) or other undesirable harms (e.g., financial harms, legal risks) If so, please provide a description. Is there anything a future user could do to mitigate these undesirable harms?
Answer: No.

Are there tasks for which the dataset should not be used? If so, please provide a description.

Answer: No.

B.6 DISTRIBUTION

Will the dataset be distributed to third parties outside of the entity (e.g., company, institution, organization) on behalf of which the dataset was created? If so, please provide a description.
Answer: Yes, it will be distributed widely.

How will the dataset will be distributed (e.g., tarball on website, API, GitHub)? Does the dataset have a digital object identifier (DOI)?

Answer: Github

When will the dataset be distributed?
Answer: At the end of the anonymity period.

Will the dataset be distributed under a copyright or other intellectual property (IP) license, and/or under applicable terms of use (ToU)? If so, please describe this license and/or ToU, and provide a link or other access point to, or otherwise reproduce, any relevant licensing terms or ToU, as well as any fees associated with these restrictions.
Answer: Yes, TBC

Have any third parties imposed IP-based or other restrictions on the data associated with the instances? If so, please describe these restrictions, and provide a link or other access point to, or otherwise reproduce, any relevant licensing terms, as well as any fees associated with these restrictions.
Answer: No.

Do any export controls or other regulatory restrictions apply to the dataset or to individual instances? If so, please describe these restrictions, and provide a link or other access point to, or otherwise reproduce, any supporting documentation.
Answer: No.

B.7 MAINTENANCE

Who is supporting/hosting/maintaining the dataset?
Answer: The authors, but we will also accept PRs from the community if they would like to extend the dataset and they pass our quality checks.

How can the owner/curator/manager of the dataset be contacted (e.g., email address)?
Answer: via Github

Is there an erratum? If so, please provide a link or other access point.
Answer: No.

Will the dataset be updated (e.g., to correct labeling errors, add new instances, delete instances)? If so, please describe how often, by whom, and how updates will be communicated to users (e.g., mailing list, GitHub)?
Answer: Yes, we will consider any requests and if any mistakes are pointed out we will update regularly (monthly to start with).

If the dataset relates to people, are there applicable limits on the retention of the data associated with the instances (e.g., were individuals in question told that their data would be retained for a fixed period of time and then deleted)? If so, please describe these limits and explain how they will be enforced.
Answer: Not applicable.

Will older versions of the dataset continue to be supported/hosted/maintained? If so, please describe how. If not, please describe how its obsolescence will be communicated to users.
Answer: No.

If others want to extend/augment/build on/contribute to the dataset, is there a mechanism for them to do so? If so, please provide a description. Will these contributions be validated/verified? If so, please describe how. If not, why not? Is there a process for communicating/distributing these contributions to other users? If so, please provide a description.
Answer: Yes, via PRs.

C Examples of problems for different scenarios

Description	Prompt Example

Data-Independent (IND)	`import pandas as pd`
	`df = pd.DataFrame()`
	`df["String"] = ["Anna"]`
	# create a new column with the String column capitalized

Data-Dependent (DEP)	`import pandas as pd`
	`df = pd.DataFrame()`
	`df["Name"] = ["Anna"]`
	`df["DoB"] = ["20/10/1989"]`
	# extract the year from the date

External knowledge (EXT)	`import pandas as pd`
	`df = pd.DataFrame()`
	`df["State"] = ["WA"]`
	# expand the state name.

Anonymous header and no data	`import pandas as pd`
	`df = pd.DataFrame()`
	`df["Col1"] = []`
	# Create a column with the first number

Proper header	`import pandas as pd`
	`df = pd.DataFrame()`
	`df["Address"] = []`
	# Create a column with the first number from the address

Proper header and data sample	`import pandas as pd`
	`df = pd.DataFrame()`
	`df["Address"] = ["6088 184A ST"]`
	# Create a column with the first number

Full data	`import pandas as pd`
	`df = pd.DataFrame()`
	`df["Address"] = ["6088 184A ST", "4234 32A ST"]`
	# Create a column with the first number

Figure 6: Prompt examples for the different taxonomies. Top three rows cover task taxonomies. For class IND, the query contains all information required for the completion: the input column name and the task. For class DEP, the query does not specify the date format or column. For class EXT, the model needs external world knowledge about US states. Bottom four rows cover data availability.

Description	Prompt & Completion Example

Completion for a prompt with no data in the dataframe	`import pandas as pd`
	`df = pd.DataFrame()`
	`df["Col1"] = []`
	# Create a column with the first number
	`df["Col2"] = [1]`

Completion for a prompt with only the first row of the dataframe	`import pandas as pd`
	`df = pd.DataFrame()`
	`df["Col1"] = ["6088 184A ST"]`
	# Create a column with the first number
	`df['Col2'] = df['Col1'].str.split().str[0]`

Completion for a prompt with only a direct reference to the columns	`import pandas as pd`
	`df = pd.DataFrame()`
	`df["Address"] = []`
	# Create a column with the first number from the address
	`df['Number'] = df['Address'].str.extract(`
	`  '(\d+)', expand = False)`

Figure 7: Prompt examples and completions results for type errors in the TypeGoldSet using different data availability.

Description	Prompt & Completion Example

Completion for a prompt with no data in the dataframe	`import pandas as pd`
	`df = pd.DataFrame()`
	`df["Col1"] = []`
	# Create a new column with the time stamp
	`df['New_Col'] = pd.datetime.now()`
	`print(df)`

Completion for a prompt with only the first row of the dataframe	`import pandas as pd`
	`df = pd.DataFrame()`
	`df["Col1"] = ["Fri, 06 Feb 1987 06:00:37 GMT"]`
	# Create a new column with the time stamp
	`df['Col2'] = pd.to_datetime(df.Col1)`

Completion for a prompt with only a direct reference to the columns	`import pandas as pd`
	`df = pd.DataFrame()`
	`df["Date"] = []`
	# Create a new column with the time stamp from the dates
	`df['Time Stamp'] = pd.to_datetime(df['Date']).apply(`
	`  lambda x:  x.timestamp())`

Figure 8: Prompt examples and completions results for value errors in the TypeGoldSet using different data availability.

Description	Prompt & Completion Example

Original	`import pandas as pd`
	`df = pd.DataFrame()`
	`df["Col1"] = [41, 163, ...]`
	`df["Col2"] = [83, 60, ...]`
	# Create a new column with the sum of the numeric columns
	`df['Col3'] = df['Col1'] + df['Col2']`

Mixed types and numeric delta	`import pandas as pd`
	`df = pd.DataFrame()`
	`df["Col1"] = [41, "163", ...]`
	`df["Col2"] = ["83", 59.9, ...]`
	# Create a new column with the sum of the numeric columns
	`df['sum'] = df['Col1'].astype(float) +`
	`df['Col2'].astype(float)`

Empty strings	`import pandas as pd`
	`df = pd.DataFrame()`
	`df["Col1"] = [41, "", ...]`
	`df["Col2"] = ["", "", ...]`
	# Create a new column with the sum of the numeric columns
	`df['Sum'] = df.apply(pd.to_numeric,`
	`errors='coerce').sum(axis=1)`
	`print(df)`

Original	`import pandas as pd`
	`df = pd.DataFrame()`
	`df["String"] = ["mailto: andrew_wilson@gmx.net",`
	`... "mailto: emily_marie_lopez@gmx.net"]`
	# Create a new column with the email address extracted from the string column
	`df['Email'] = df['String'].str.extract(`
	`'([a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-.]+)',`
	`expand=True)`

Text corruption	`import pandas as pd`
	`df = pd.DataFrame()`
	`df["String"] = ["andrew_wilson@gmx.net",`
	`... "-mailto: emily_marie_lopez@gmx.net"]`
	# Create a new column with the email address extracted from the string column
	`df['Email'] = df['String'].str.extract(`
	`r'(?<=mailto:) (\\S+)')`

Figure 9: Prompt examples and completions results for corruptions in TypeGoldSet

D Prompt Experiments

This section includes concrete prompt examples for the different prompting schemes. Refer to tables Table 2 for instruction prompts, Table 3 for example-usage prompts, Table 4 for data description prompts.

E Implementation details

E.1 Statistics about our datasets

E.2 Task classes

We distinguish three task classes:

(IND) data-independent

For these tasks there is a solution that can be applied to the data without knowing the content or format. Hence, if there is a Pandas or default Python function that can be applied with default parameters the task is of class (IND). Examples are simple date operations, like to extract days or hours, with default formats that can be inferred by the pandas function "to_date" are of class A.

(DEP) data-dependent

For these tasks the model needs to know the data content and/or format to find the solutions. For example, if we would need a special parameter, filtering or any kind of parameterization derived from the content and the format of the data the task it is of class (DEP). Examples are data operations on dates with non-default format, mixed formats or corrupted values. In these cases we need either a special parameter (for example errors=coerce) or other transformations derived from the data.

(EXT) external-dependent

The model can not find a solution of the task only knowing the data content and the format. Its need additional open world knowledge. This knowledge needs to be beyond the knowledge about the pandas and python syntax, APIs and libraries. Examples are operations in names where we need knowledge about what a surname, middle name or first name is. Or operations on addresses where we might need to know what a zip code or a state is.

E.3 Anonymous columns

We say a column header (or column name) is anonymous when it does not give any information about the content or type of the columns. For example the column name "Names" indicates that the column contains names or the column name "Dates" indicates that the column contains date values. On the other hand a column name like "Data" or "Info" does give no information about the content or type of the column.

E.4 Adding noise

For our investigations on how much use the mode is making of the actual data in the dataframes and to check the quality of the solution on non-perfect data, we added additional noise. This additional noise is:

Corrupting values:

With a probability of 20% a value in an input column will have a corrupted value. For string like objects this corruptions will be an insertion of a space, a dash or a new line in a random position in the corresponding string.

Mixing formats:

For numerical values, we mix integer and float, for date values we mix different date formats and for names and addresses with permute the tokens.

Missing values:

With a probability of 20% a value in an input column will be replaced with a empty string which represents a missing value in all our settings.

E.5 Obtaining completions

Parallelization. For efficiency, we request multiple completions from Codex 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 Codex 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 experiments.

Temperature. We use the same k-dependent temperature t as described in Chen et al. (2021a); i.e. for $k = 1$, $t = 0.2$; for $1 < k \leq 5$, $t = 0.4$, for $5 < k \leq 50$, $t = 0.6$; otherwise $t = 0.8$.

Prompt Name	Prompt Example (Query Q)
instruct-concise	*write a concise, short and idiomatic pandas solution for the following query:* create a new column with the last names.
instruct-execute	*write an executable and type correct pandas solution for the following query:* create a new column with the last names.
instruct-function	*write a pandas function that* create a new column with the last names.
instruct-iterative	create a new column with the last names. *Prioritize the iterative programming style with use of for loops, while loops and iterative data structures*
instruct-lambda	create a new column with the last names. *Prioritize the functional programming style with use of lambdas, comprehensions, and generators*
instruct-domain	create a new column with the last names. *Prioritize the use of pandas operators that deal with string manipulation.*
assert-concise	create a new column with the last names. *The following example demonstrates how to write a concise, short and idiomatic pandas solution.*
assert-execute	create a new column with the last names. *The following example demonstrates how to write an executable and type correct pandas solution.*
assert-function	*The following example demonstrates how to write a pandas function that* create a new column with the last names.
assert-iterative	create a new column with the last names. *The following example demonstrates how to prioritize the iterative programming style with use of for loops, while loops and iterative data structures.*
assert-lambda	create a new column with the last names. *The following example demonstrates how to prioritize the functional programming style with use of lambdas, comprehensions, and generators.*
assert-domain	create a new column with the last names. *The following example prioritizes the use of pandas operators that deal with string manipulation.*
ask-concise	*how to* create a new column with the last names*? how to write a concise, short and idiomatic pandas solution?*
ask-execute	*how to* create a new column with the last names*? how to write an executable and type correct pandas solution?*
ask-function	*how to write a pandas function that* create a new column with the last names*?*
ask-iterative	*how to* create a new column with the last names*? how to prioritize the iterative programming style with use of for loops, while loops and iterative data structures?*
ask-lambda	*how to* create a new column with the last names*? how to prioritize the functional programming style with use of lambdas, comprehensions, and generators?*
ask-domain	*how to* create a new column with the last names*? how to prioritize the use of pandas operators that deal with string manipulation?*

Table 2: Instruction prompt experiments include command-style prompts, assert-style prompts and ask-style prompts illustrated on the query *create a new column with the last names.* The different prompts vary along two dimensions: the style in which the model is queried and the property we want the completions to emphasize. The phrasing style we explore are instruct, assert and ask. Examples of properties include executable, type correctness and using operators from the problem domain.

Stop sequences. The most effective stop sequence we found that allows Codex 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 Codex 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 Codex'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 `&lt;` 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.

E.6 Executing completions

Rewriting. Completions returned by Codex 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

Prompt Name	Prompt Example

usage-function	`import pandas as pd`
	`def startswith_at(txt):`
	`  return txt.startswith('@')`
	`df = pd.DataFrame("Names":["Charles Moore", "Anna Green"])`
	# create a new column with the last names.

usage-iterative	`import pandas as pd`
	`df = pd.DataFrame("a":[1, 2, 3])`
	# loop over the rows of a df using pd.iloc, create a function to iterate over the df.
	`for i in df.index:`
	`  val = df[col].iloc[i]`
	
	`df = pd.DataFrame("Names":["Charles Moore", "Anna Green"])`
	# create a new column with the last names.

usage-lambda	`import pandas as pd`
	`df = pd.DataFrame("a":[1, 2, 3])`
	# group df on column b and keep half of the elements at random
	`dfout = df.groupby('b').apply(lambda x:x.sample(frac=0.5))`
	
	`df = pd.DataFrame("Names":["Charles Moore", "Anna Green"])`
	# create a new column with the last names.

Table 3: Example-usage prompt examples illustrated on the query *create a new column with the last names.* Example-usage prompts append example code snippets to the query that demonstrate how to solve the pandas problem in various ways.

- 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 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}.

E.7 Evaluation

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:

- Extra columns
- Different column order
- Different column headers
- Number expected; actual is a number within small relative error (default 0.01)
- Number expected; actual is a string that parses as a number within small relative error
- Boolean expected; actual is number 0 or 1
- Boolean expected; actual is a string that represents a truth value

Prompt name	Prompt template
column-info	`df = pd.DataFrame("Position":[3, 14, 25])` Given position number n, create a new column with the nth letter of the alphabet. **There is 1 column in the input data. The column Position has 10 entries, 10 of which are unique**.
type-info	`df = pd.DataFrame("Position":[3, 14, 25])` Given position number n, create a new column with the nth letter of the alphabet. **Returns** **----------** **pd.Dataframe with new columns. Data columns are as follows:** **Position integers (as `int`)** **Letter strings (as `str`)**
regex-info (exact)	`df = pd.DataFrame("Position":[3, 14, 25])` Given position number n, create a new column with the nth letter of the alphabet. **10 of 10 elements in column Position match the regex format `[0-9]+` where examples of these elements include 8 and 23.**
regex-info (partial)	`df = pd.DataFrame("Body Fat":["4%", "14%"])` create a new column that writes Body Builder if Body Fat is less than 5%, else if it is less than 13% writes Athletic, else writes No Data Yet. **9 of 10 (90%) elements in column Body Fat match the regex format [0-9]+% where an example of these elements is (40%). 1 of 10 (10%) elements in column Body Fat match the regex format NA where an example of these elements is NA.**
regex-type-info	`df = pd.DataFrame("Position":[3, 14, 25])` Given position number n, create a new column with the nth letter of the alphabet. **Returns** **----------** **pd.Dataframe with new columns. Data columns are as follows:** **Position integers (as `int`) that match the regex `[0-9]+`** **Letter strings (as `str`)**
regex-type-col-info	`df = pd.DataFrame("Position":[3, 14, 25])` Given position number n, create a new column with the nth letter of the alphabet. **There is 1 column in the input data. The column Position has 10 unique entries.** **Returns** **----------** **pd.Dataframe with new columns. Data columns are as follows:** **Position integers (as `int`) that match the regex `[0-9]+`** **Letter strings (as `str`)**
properties (textual)	`df = pd.DataFrame("Position":[3, 14, 25])` Given position number n, create a new column with the nth letter of the alphabet. **The column Position has the following properties: All elements contain digits.**
properties (format)	`df = pd.DataFrame("Position":[3, 14, 25])` Given position number n, create a new column with the nth letter of the alphabet. **The column Position satisfies ContainsDigits()**

Table 4: Data-description prompts illustrated on the queries *Given the position number n, create a new column with the nth letter of the alphabet*. Data-description prompts aim to extract task-specific structural information about the data input provided by the user. We automatically compute information about the number of elements in the columns, the type of each of the column elements (one of string, integer, boolean) and properties satisfied by the elements. In addition, we compute a regular expression that captures a pattern of the elements in the column.

- 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.

F Detailed evaluation

F.1 Stability analysis

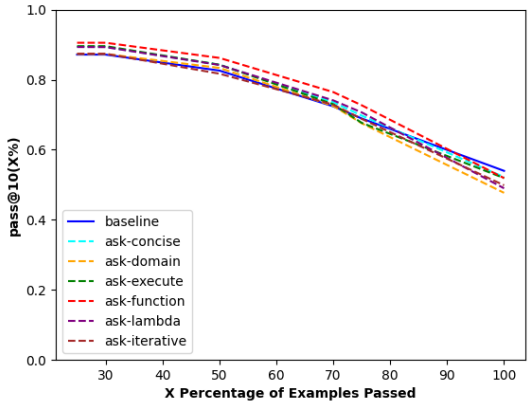
To estimate the stability of the CODEXDATA algorithm on pass@k we ran each task in TYPESET and SOFSET 10 times and estimated the pass@5. In Fig.15 we show the corresponding means and standard deviations for SOFSET (top) and TYPESET (bottom). Due to the effort to run CODEX for each example we were only able to calculate these numbers for k=5.

Data type	Example of task	TYPESET	SOFSET
strings	sub-string extracts: df["col1"].str[4:10]	16	116
numbers	divisions e.g. df["col1"] / df["col2"]	14	34
names	surname in upper-case df["Givenname"].str.upper()[0]	12	13
dates	extract month e.g. pd.DatetimeIndex(df['date']).month	14	32
units	conversions	40	1
addresses	extract the state df["col1"].str[:-2]	18	4
mixed types	combine columns df["col1"] + df["col2"].astype(str)	14	0

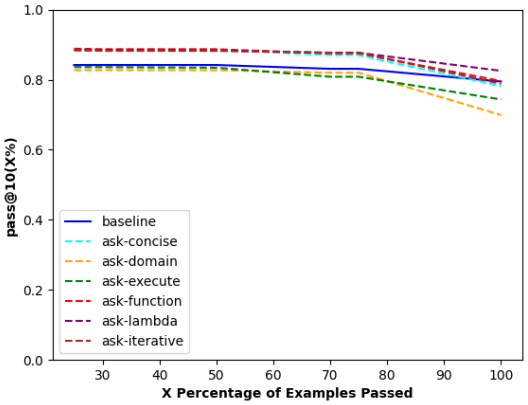
Table 5: Number of tasks per data type in TYPESET and SOFSET with an example.

G Remaining InCoder experiments

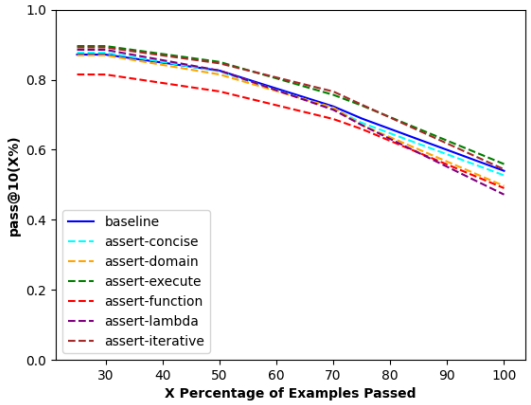
G.1 Detailed error analyses



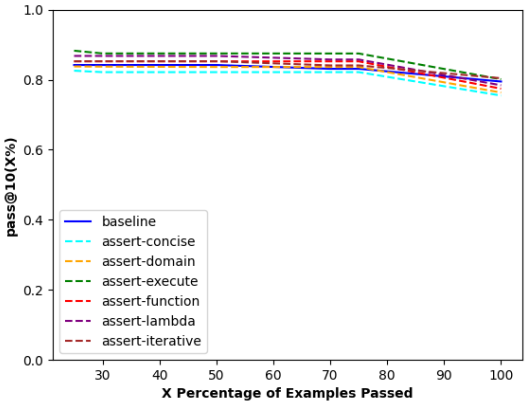
SOFSET Instruction Prompts (Questions)



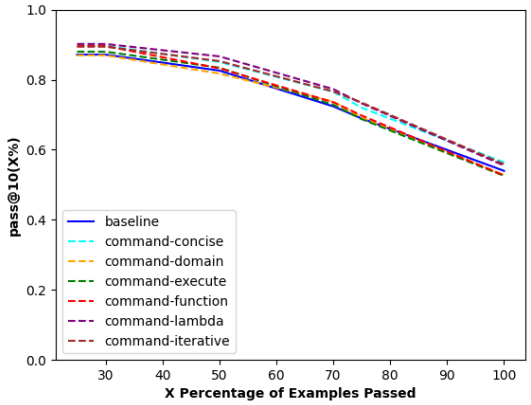
TYPESET Instruction Prompts (Questions)



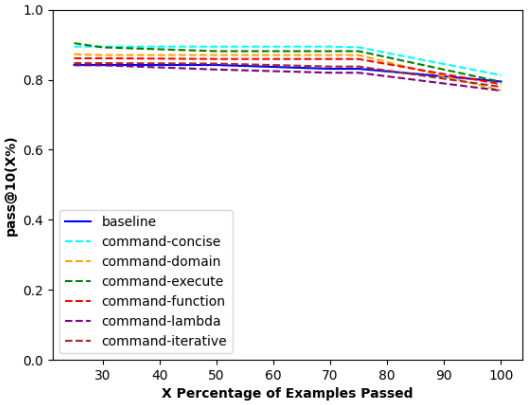
SOFSET Instruction Prompts (Assertions)



TYPESET Instruction Prompts (Assertions)

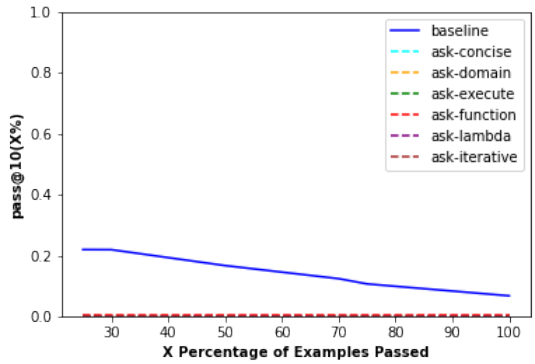


SOFSET Instruction Prompts (Commands)

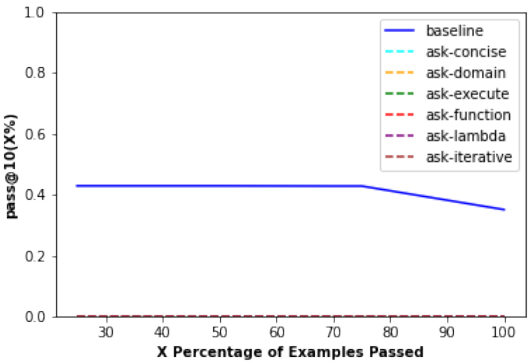


TYPESET Instruction Prompts (Commands)

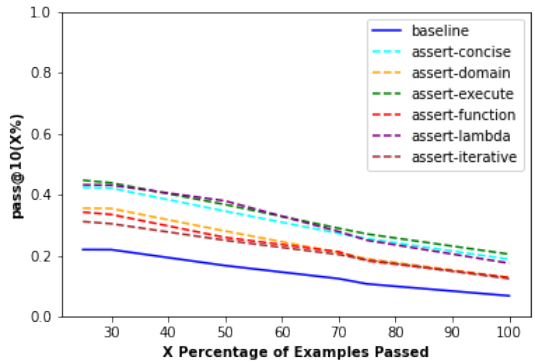
Figure 10: This figure presents prompt results for the instruction type prompts for SOFSET and TYPESET datasets for CODEX. We evaluate the different ways of phrasing the instruction: as a command, as an assertion and as a question. We observe that instruction prompts that are phrased as a command perform slightly better than instruction prompts phrased as an assertion or question. A speculation can be made about the impact of training data: not many natural language comments are typically phrased as questions.



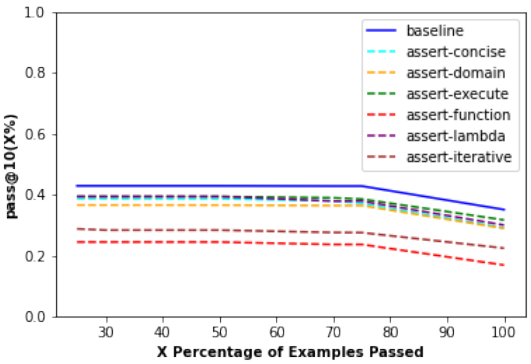
SOFSET Instruction Prompts (Questions)



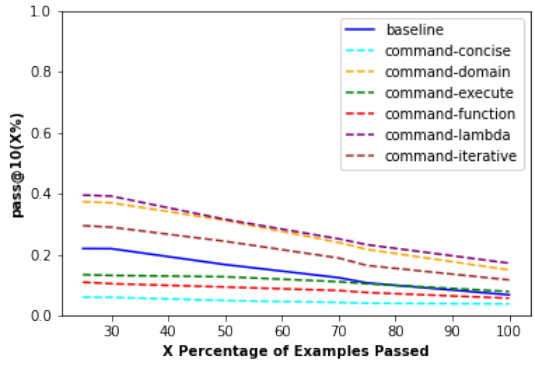
TYPESET Instruction Prompts (Questions)



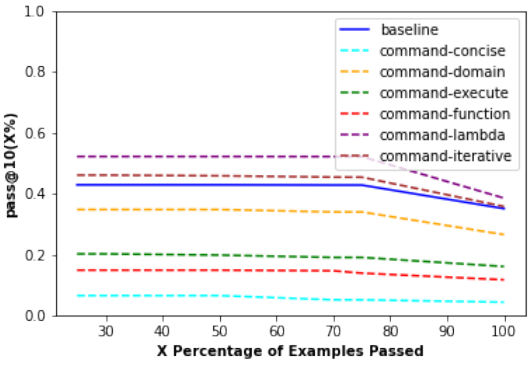
SOFSET Instruction Prompts (Assertions)



TYPESET Instruction Prompts (Assertions)

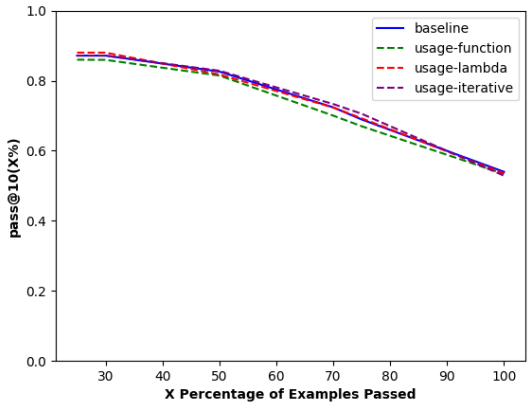


SOFSET Instruction Prompts (Commands)

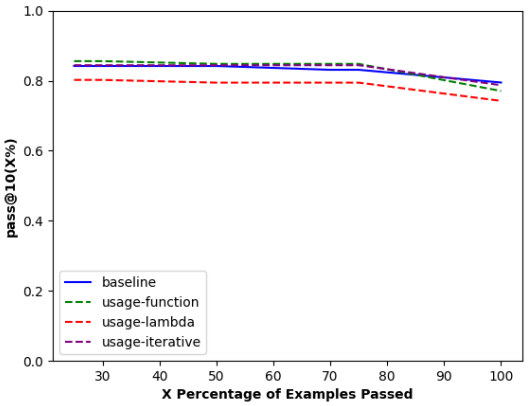


TYPESET Instruction Prompts (Commands)

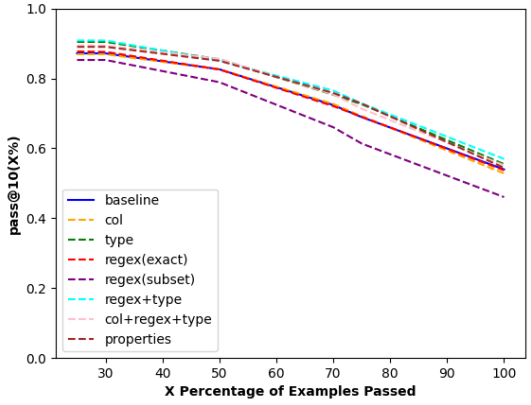
Figure 11: This figure presents prompt results for the instruction type prompts for SOFSET and TYPESET datasets for INCODER. We evaluate the different ways of phrasing the instruction: as a command, as an assertion and as a question. We observe that instruction prompts that are phrased as a command perform slightly better than instruction prompts phrased as an assertion or question. A speculation can be made about the impact of training data: not many natural language comments are typically phrased as questions.



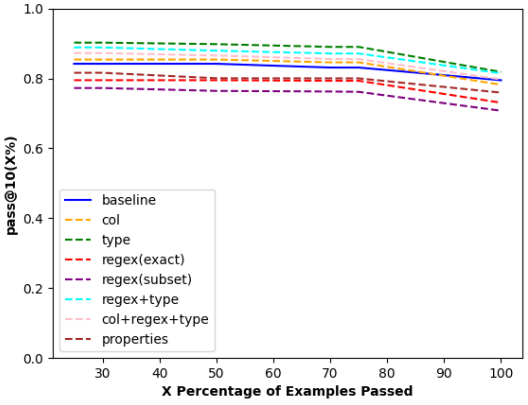
SOFSET Example-Usage Prompts



TYPESET Example-Usage Prompts

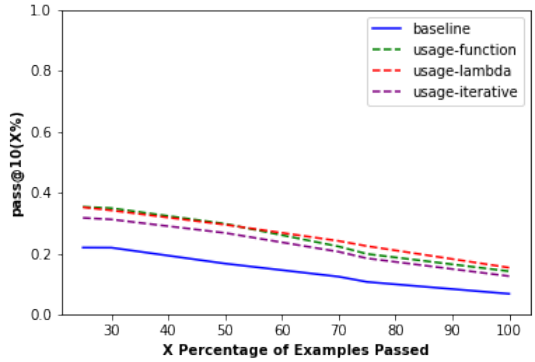


SOFSET Data Description Prompts

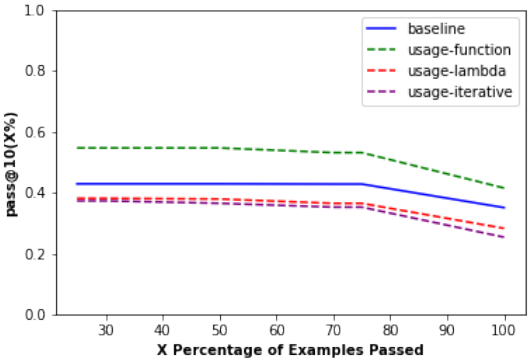


TYPESET Data Description Prompts

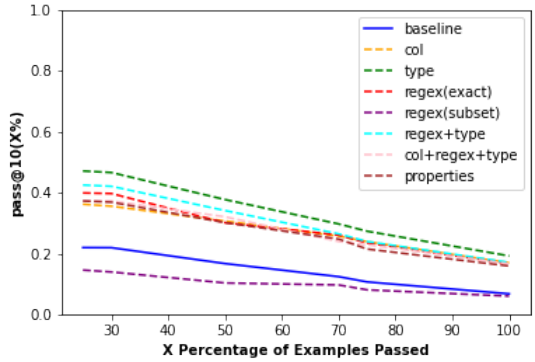
Figure 12: This figure presents prompt results for the example usage and data description prompting strategy for SOFSET and TYPESET datasets for CODEX. We observe that example usage prompts always perform worse or same as the baseline. Amongst the data description prompts, augmenting type information leads to better performance in both the datasets.



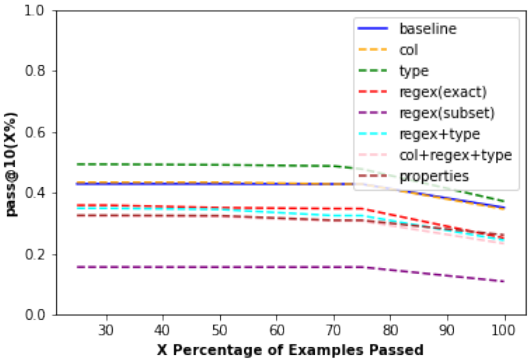
SOFSET Example-Usage Prompts



TYPESET Example-Usage Prompts

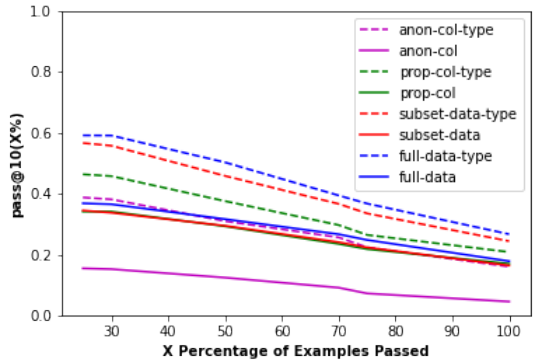


SOFSET Data Description Prompts

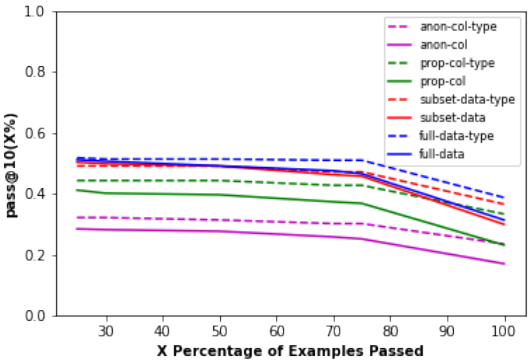


TYPESET Data Description Prompts

Figure 13: This figure presents prompt results for the example usage and data description prompting strategy for SOFSET and TYPESET datasets for INCODER. We observe that example usage prompts always perform worse or same as the baseline. Amongst the data description prompts, augmenting type information leads to better performance in both the datasets.



SOFSET



TYPESET

Figure 14: Interplay between the varying amount of data redaction and how data description in the prompt helps with performance for INCODER. For each redaction, we evaluate the performance by adding a data description to the prompt in the form of type information. Type information is especially helpful when we only have access to anonymous columns and a subset of the data. The extracted data descriptions do not directly depict private information.

Notation

Metric	Description	Type
pass@k	see metrics	unit test
pass@k(%)	see metrics	unit test
NumUniqueValid	Number of unique completions that are valid (have no runtime errors and have the correct output type).	code
NumSuccess	Number of completions that produce the correct output type and values (pass unit test).	unit test
NumError	Number of runtime errors (completions with no valid output produced).	runtime
NumSyntaxErrors	Number of syntax errors (completion generated is no valid python code).	runtime
NumTypeErrors	Number of type errors in the generated completions (operations/function not allowed for type).	runtime
NumValueErrors	Number of value errors (value caused error in operation/-function).	runtime
NumIndexErrors	Number of error involving the dataframe index.	runtime
NumAttributeErrors	Number of attribute errors (referred object attribute does not exists for example).	runtime
NumNameErrors	Number of name errors (function/variable does not exists).	runtime
NumKeyErrors	Number of key errors (column does not exist).	runtime
NumRegexErrors	Number of regular expression errors.	runtime
NumAssertionErrors	Number of assertion errors in the completions.	runtime
NumCorrectOutputType	Number of completion that produce the correct output type (a new column).	semantic
NumTypeMismatch	Number of mismatches of the output type of the completion and the ground truth output. Please note that the mismatch values are only computed for completions that are valid/execute.	semantic
NumValueMismatch	Number of mismatches of the column values from the generated by the completion and the ground truth.	semantic
DiffColNumber	Number of completions with different output column number than ground truth output.	semantic
InputColumnsError	Number of completions that do not use the input columns from the dataframe.	semantic
ExtraColumnsError	Number of completions that use columns that are not relevant for the task.	semantic
NumCompletions	Number of completions retrieved.	stats
CompletionLenght	String length of generated completions.	stats

Table 6: Short description of the used metrics. For each experiment we calculate the above metrics and report them as average over all completions and all examples per evaluation. Please see the Metrics section for further details.

Full-data vs no-data experiments SOFSET

	full-data	no-data prop-col	subset-data	no-data anon-col
Pass@K	0.673	0.583	0.6016	0.5687
Pass@K75pct	0.8273	0.7533	0.7512	0.7217
Pass@K70pct	0.8519	0.7761	0.7874	0.7709
Pass@K50pct	0.933	0.8584	0.8642	0.8259
Pass@K30pct	0.9524	0.906	0.9069	0.8625
Pass@K25pct	0.9531	0.9123	0.9152	0.8662
NumUniqueValid	0.8086	0.6397	0.7529	0.6604
NumSuccess	0.3135	0.2292	0.2952	0.2242
NumError	0.1833	0.3454	0.2384	0.3274
NumSyntaxErrors	0.0072	0.0159	0.0074	0.0167
NumTypeErrors	0.0214	0.0306	0.0252	0.0327
NumValueErrors	0.0405	0.0655	0.0714	0.0632
NumIndexErrors	0.007	0.0109	0.007	0.0088
NumAttributeErrors	0.0098	0.0146	0.0106	0.0209
NumNameErrors	0.0045	0.0094	0.0062	0.0087
NumKeyErrors	0.0081	0.018	0.0068	0.0715
NumRegexErrors	0.0	0.0	0.0	0.0
NumAssertionErrors	0.0	0.0	0.0	0.0
NumCorrectOutputType	0.8086	0.6397	0.7529	0.6604
NumTypeMismatch	0.0081	0.0149	0.0087	0.0122
NumValueMismatch	0.4951	0.4105	0.4577	0.4362
DiffColNumber	0.0264	0.0313	0.0201	0.0301
DiffTableNumber	0.0264	0.0313	0.0201	0.0301
CompletionLenght	9.5091	9.5446	9.5564	9.6114
NumCompletions	28.7045	40.0606	31.25	40.1053

Table 7: Detailed results for SOFSET for task class **(IND)** and different levels of data redaction. Metrics shown are average numbers across the sampled completions.

Full-data vs no-data experiments SOFSET

	full-data	no-data prop-col	subset-data	no-data anon-col
Pass@K	0.302	0.1213	0.1384	0.1354
Pass@K75pct	0.4031	0.1739	0.2294	0.1564
Pass@K70pct	0.4352	0.1859	0.2793	0.1901
Pass@K50pct	0.6094	0.376	0.4767	0.3741
Pass@K30pct	0.7121	0.4835	0.605	0.4546
Pass@K25pct	0.7121	0.4835	0.605	0.4546
NumUniqueValid	0.6071	0.2997	0.5895	0.2926
NumSuccess	0.0828	0.0296	0.0497	0.0243
NumError	0.3761	0.6936	0.398	0.7017
NumSyntaxErrors	0.0112	0.0207	0.0105	0.0209
NumTypeErrors	0.172	0.3263	0.1473	0.259
NumValueErrors	0.0653	0.123	0.1023	0.0928
NumIndexErrors	0.0018	0.0063	0.0049	0.0083
NumAttributeErrors	0.0398	0.0876	0.0333	0.0865
NumNameErrors	0.0029	0.0061	0.0036	0.0034
NumKeyErrors	0.0074	0.0162	0.0052	0.1619
NumRegexErrors	0.0	0.0	0.0	0.0
NumAssertionErrors	0.0	0.0003	0.0	0.0
NumCorrectOutputType	0.6071	0.2997	0.5895	0.2926
NumTypeMismatch	0.0168	0.0068	0.0126	0.0057
NumValueMismatch	0.5244	0.2701	0.5398	0.2683
DiffColNumber	0.0328	0.0413	0.0419	0.0261
DiffTableNumber	0.0328	0.0413	0.0419	0.0261
CompletionLenght	9.6305	9.8837	9.6271	9.9495
NumCompletions	41.4474	80.3421	42.3421	88.3243

Table 8: Detailed results for SOFSET for task class **(DEP)** and different levels of data redaction. Metrics shown are average numbers across the sampled completions.

Full-data vs no-data experiments SOFSET

	full-data	no-data prop-col	subset-data	no-data anon-col
Pass@K	0.2078	0.0962	0.1369	0.1124
Pass@K75pct	0.4314	0.2181	0.2546	0.2097
Pass@K70pct	0.4641	0.2835	0.2869	0.2258
Pass@K50pct	0.6079	0.3806	0.5297	0.3277
Pass@K30pct	0.7273	0.5732	0.6814	0.5658
Pass@K25pct	0.7273	0.5732	0.6814	0.5658
NumUniqueValid	0.6669	0.4943	0.6079	0.4372
NumSuccess	0.0559	0.011	0.0314	0.0244
NumError	0.3193	0.4991	0.3826	0.5566
NumSyntaxErrors	0.0118	0.0156	0.0126	0.0174
NumTypeErrors	0.0492	0.1167	0.0433	0.1196
NumValueErrors	0.1019	0.1487	0.1449	0.1448
NumIndexErrors	0.0025	0.013	0.0058	0.0211
NumAttributeErrors	0.0248	0.0834	0.0283	0.0688
NumNameErrors	0.0126	0.0145	0.0114	0.0127
NumKeyErrors	0.0073	0.0106	0.0064	0.0937
NumRegexErrors	0.0	0.0	0.0	0.0
NumAssertionErrors	0.0	0.0	0.0	0.0
NumCorrectOutputType	0.6669	0.4943	0.6079	0.4372
NumTypeMismatch	0.0138	0.0066	0.0095	0.0061
NumValueMismatch	0.611	0.4833	0.5765	0.4128
DiffColNumber	0.0487	0.077	0.0621	0.0548
DiffTableNumber	0.0487	0.077	0.0621	0.0548
CompletionLenght	9.5894	9.8389	9.6326	9.8971
NumCompletions	36.4516	55.2581	41.3548	65.2258

Table 9: Detailed results for SOFSET for task class **(EXT)** and different levels of data redaction. Metrics shown are average numbers across the sampled completions.

Full-data vs no-data experiments TYPESET

	full-data	no-data prop-col	subset-data	no-data anon-col
Pass@K	0.55	0.4	0.5881	0.5663
Pass@K75pct	0.7741	0.6263	0.7013	0.6915
Pass@K70pct	0.7741	0.6263	0.7013	0.6915
Pass@K50pct	0.7741	0.6263	0.7013	0.6923
Pass@K30pct	0.7741	0.6263	0.7263	0.6923
Pass@K25pct	0.7741	0.6263	0.7263	0.6923
NumUniqueValid	0.8481	0.6374	0.793	0.6897
NumSuccess	0.4008	0.2669	0.3914	0.3257
NumError	0.1399	0.3498	0.2006	0.2987
NumSyntaxErrors	0.0025	0.0064	0.0083	0.0104
NumTypeErrors	0.0455	0.1061	0.069	0.0583
NumValueErrors	0.0173	0.0277	0.0146	0.1008
NumIndexErrors	0.0	0.0004	0.0	0.0
NumAttributeErrors	0.0	0.0197	0.0006	0.0097
NumNameErrors	0.0	0.0047	0.0	0.0039
NumKeyErrors	0.0042	0.002	0.0	0.0034
NumRegexErrors	0.0	0.0	0.0	0.0
NumAssertionErrors	0.0	0.0	0.0	0.0
NumCorrectOutputType	0.8481	0.6374	0.793	0.6897
NumTypeMismatch	0.012	0.0128	0.0063	0.0116
NumValueMismatch	0.4474	0.3705	0.4016	0.364
DiffColNumber	0.0087	0.0169	0.0043	0.0103
DiffTableNumber	0.0087	0.0169	0.0043	0.0103
InputColumnsError	0.0961	0.0539	0.0721	0.0222
ExtraColumnsError	0.0	0.0	0.0	0.0
CompletionLenght	9.5944	9.7442	9.6465	9.6858
CompletionSize	25.15	45.35	28.15	41.8

Table 10: Detailed results for TYPESET for task class **(IND)** and different levels of data redaction. Metrics shown are average numbers across the sampled completions.

Full-data vs no-data experiments TYPESET

	full-data	no-data prop-col	subset-data	no-data anon-col
Pass@K	0.9243	0.3454	0.8071	0.3333
Pass@K75pct	0.9243	0.3454	0.8071	0.4876
Pass@K70pct	0.9243	0.3618	0.8071	0.4876
Pass@K50pct	0.9243	0.3748	0.8071	0.5293
Pass@K30pct	0.9243	0.375	0.8071	0.5293
Pass@K25pct	0.9243	0.375	0.8071	0.5293
NumUniqueValid	0.7122	0.325	0.6844	0.5172
NumSuccess	0.4548	0.1282	0.386	0.1739
NumError	0.2822	0.674	0.3156	0.4828
NumSyntaxErrors	0.0	0.0159	0.0	0.0
NumTypeErrors	0.1031	0.1846	0.0854	0.2164
NumValueErrors	0.0567	0.1867	0.0875	0.0808
NumIndexErrors	0.0	0.0213	0.0	0.0129
NumAttributeErrors	0.0138	0.1038	0.0202	0.0851
NumNameErrors	0.0	0.0	0.0	0.0023
NumKeyErrors	0.0	0.0	0.0	0.0181
NumRegexErrors	0.0	0.0	0.0	0.0
NumAssertionErrors	0.0	0.0	0.0	0.0
NumCorrectOutputType	0.7122	0.325	0.6844	0.5172
NumTypeMismatch	0.0056	0.001	0.0	0.0
NumValueMismatch	0.2574	0.1968	0.2984	0.3432
DiffColNumber	0.0054	0.013	0.006	0.006
DiffTableNumber	0.0054	0.013	0.006	0.006
InputColumnsError	0.0889	0.0448	0.0464	0.1982
ExtraColumnsError	0.0	0.0	0.0	0.0
CompletionLenght	9.5041	9.8964	9.7128	9.8413
CompletionSize	30.5	68.75	35.25	47.25

Table 11: Detailed results for TYPESET for task class **(DEP)** and different levels of data redaction. Metrics shown are average numbers across the sampled completions.

Full-data vs no-data experiments TYPESET

	full-data	no-data prop-col	subset-data	no-data anon-col
Pass@K	0.854	0.4407	0.7824	0.2383
Pass@K75pct	0.8947	0.5174	0.8131	0.2677
Pass@K70pct	0.8947	0.5174	0.8131	0.2677
Pass@K50pct	0.8975	0.5174	0.8131	0.2677
Pass@K30pct	0.8976	0.5174	0.8131	0.2677
Pass@K25pct	0.8976	0.5174	0.8131	0.2677
NumUniqueValid	0.7303	0.4097	0.6981	0.292
NumSuccess	0.3118	0.1299	0.2794	0.0728
NumError	0.2639	0.5855	0.2994	0.7016
NumSyntaxErrors	0.0087	0.0186	0.0049	0.0053
NumTypeErrors	0.0807	0.2443	0.1063	0.131
NumValueErrors	0.0677	0.0889	0.0493	0.0846
NumIndexErrors	0.0036	0.0009	0.0004	0.0023
NumAttributeErrors	0.0057	0.0245	0.0063	0.0145
NumNameErrors	0.0521	0.1623	0.0821	0.0083
NumKeyErrors	0.0027	0.0095	0.0042	0.0057
NumRegexErrors	0.0	0.0	0.0	0.0
NumAssertionErrors	0.0	0.0002	0.0	0.0
NumCorrectOutputType	0.7303	0.4097	0.6981	0.292
NumTypeMismatch	0.0058	0.0048	0.0025	0.0064
NumValueMismatch	0.4186	0.2798	0.4187	0.2192
DiffColNumber	0.0174	0.0182	0.0089	0.0103
DiffTableNumber	0.0174	0.0182	0.0089	0.0103
InputColumnsError	0.3928	0.1325	0.3077	0.0901
ExtraColumnsError	0.0	0.0	0.0	0.0
CompletionLenght	9.4444	9.8718	9.5836	10.1252
CompletionSize	29.0	72.6111	32.5556	95.3784

Table 12: Detailed results for TYPESET for task class **(EXT)** and different levels of data redaction. Metrics shown are average numbers across the sampled completions.

Noise level experiment

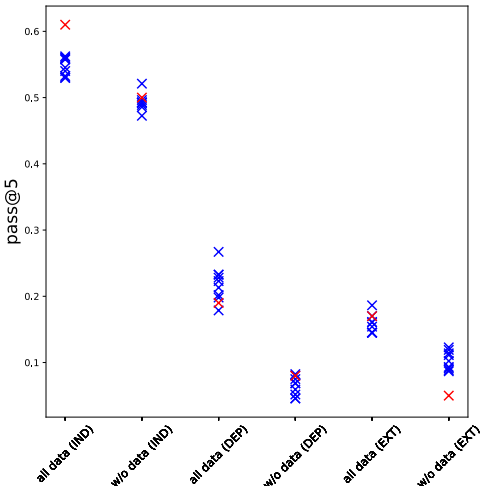
	missing	mixformat	corruption	baseline
Pass@K	0.2189	0.4613	0.2557	0.6423
Pass@K75pct	0.477	0.5435	0.6777	0.7017
Pass@K70pct	0.5868	0.5435	0.6777	0.7017
Pass@K50pct	0.6563	0.5436	0.7044	0.7127
Pass@K30pct	0.724	0.5436	0.7044	0.7189
Pass@K25pct	0.7713	0.5436	0.7044	0.7303
NumUniqueValid	0.7905	0.8084	0.7503	0.8112
NumSuccess	0.0552	0.1788	0.0987	0.3083
NumError	0.2037	0.1864	0.2476	0.1833
NumSyntaxErrors	0.0038	0.0056	0.0081	0.0095
NumTypeErrors	0.0369	0.038	0.0122	0.0358
NumValueErrors	0.0674	0.0467	0.0869	0.0678
NumIndexErrors	0.033	0.0021	0.0034	0.0004
NumAttributeErrors	0.0034	0.0433	0.0049	0.0115
NumNameErrors	0.0023	0.0022	0.0	0.0
NumKeyErrors	0.0021	0.0028	0.0	0.0025
NumRegexErrors	0.0	0.0	0.0	0.0
NumAssertionErrors	0.0	0.0	0.0	0.0
NumCorrectOutputType	0.7905	0.8084	0.7503	0.8112
NumTypeMismatch	0.0058	0.0052	0.0021	0.0055
NumValueMismatch	0.9283	0.7826	0.8667	0.6284
DiffColNumber	0.0207	0.0311	0.0289	0.0214
DiffTableNumber	0.0207	0.0311	0.0289	0.0214
InputColumnsError	0.0716	0.0276	0.0864	0.1536
ExtraColumnsError	0.0	0.0	0.0	0.0
CompletionLenght	9.6438	9.5977	9.7633	9.5553
NumCompletions	27.9167	27.1875	37.2143	26.9318

Table 13: Code completions results on TYPESETNOISY for different levels of noise. Metrics shown are average numbers across the sampled completions.

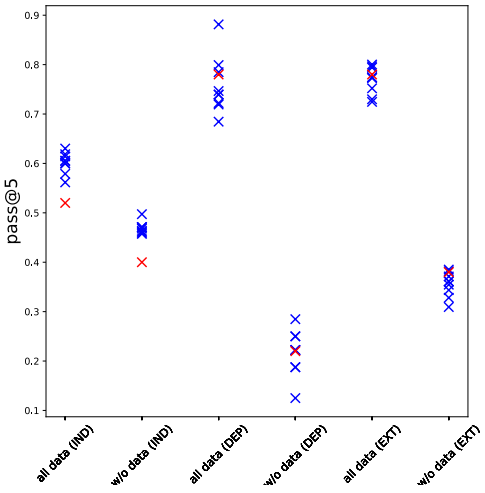
Performance per data-type on TYPESET

	units	strings	numeric	names	mixed	dates	address
Pass@K	0.9652	0.8333	0.5461	0.4809	0.5	0.4892	0.3205
Pass@K75pct	0.9798	0.9722	0.7128	0.7181	0.5587	0.5799	0.6193
Pass@K70pct	0.9798	0.9722	0.713	0.7181	0.5587	0.58	0.6646
Pass@K50pct	0.9798	0.9722	0.7938	0.7266	0.5587	0.5877	0.6928
Pass@K30pct	0.9798	0.9722	0.7938	0.7303	0.5587	0.5877	0.7354
Pass@K25pct	0.9798	0.9722	0.7938	0.7303	0.5587	0.62	0.7543
NumUniqueValid	0.7004	0.9084	0.543	0.8675	0.6703	0.6406	0.8546
NumSuccess	0.2792	0.5707	0.246	0.2807	0.2835	0.1515	0.1791
NumError	0.2963	0.0879	0.4507	0.1311	0.3192	0.3559	0.1365
NumSyntaxErrors	0.0032	0.0	0.0017	0.0028	0.0037	0.0035	0.0136
NumTypeErrors	0.1273	0.0	0.209	0.0028	0.1575	0.1229	0.0
NumValueErrors	0.0264	0.0239	0.1493	0.021	0.0276	0.0704	0.0638
NumIndexErrors	0.0018	0.0023	0.0	0.0087	0.0	0.0061	0.0074
NumAttributeErrors	0.0053	0.0051	0.0011	0.0016	0.0048	0.0401	0.0016
NumNameErrors	0.0938	0.0	0.0	0.0	0.003	0.0013	0.0023
NumKeyErrors	0.0	0.0024	0.0	0.0014	0.0	0.0004	0.0036
NumRegexErrors	0.0	0.0	0.0	0.0	0.0	0.0	0.0
NumAssertionErrors	0.0	0.0	0.0	0.0	0.0	0.0	0.0
NumCorrectOutputType	0.7004	0.9084	0.543	0.8675	0.6703	0.6406	0.8546
NumTypeMismatch	0.0033	0.0037	0.0062	0.0014	0.0105	0.0035	0.0088
NumValueMismatch	0.4212	0.3377	0.297	0.5868	0.3868	0.4891	0.6755
DiffColNumber	0.0097	0.0138	0.0043	0.0355	0.0047	0.0071	0.0124
DiffTableNumber	0.0097	0.0138	0.0043	0.0355	0.0047	0.0071	0.0124
InputColumnsError	0.6608	0.0	0.0137	0.0067	0.3417	0.0088	0.0313
ExtraColumnsError	0.0	0.0	0.0	0.0	0.0	0.0	0.0
CompletionLenght	9.3607	9.7139	9.8027	9.6785	9.6382	9.5844	9.6848
NumCompletions	29.25	22.1389	47.25	23.1071	35.0769	40.9032	24.4872

Table 14: Code completions results on TYPESET by data-types. Metrics shown are averaged across the sampled completions.



SOFSET



TYPESET

Figure 15: pass@5 scores for 10 (the red mark corresponds to Table 1) CodeXData runs for data independent tasks full-data; data independent tasks no-data; data dependent tasks full-data, data dependent tasks no-data; external knowledge tasks full-data; external knowledge tasks no-data.

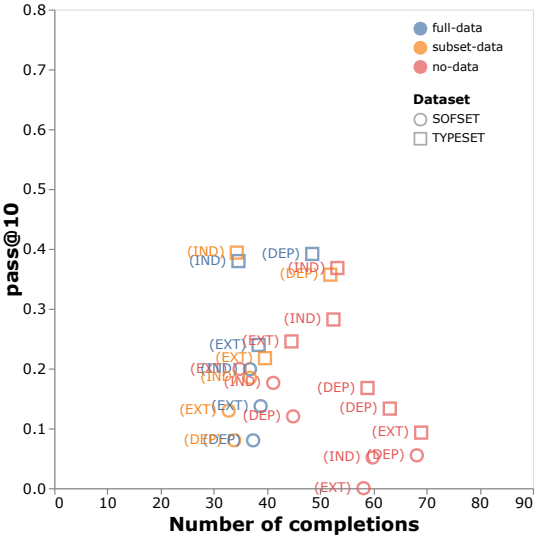


Figure 16: Interplay between pass@10 and total number of retrieved completions from INCODER for the different datasets (shapes), degree of data redaction (color) and task classes (labels). **Lower right** (Low pass@10 and many completions): most results for runs with highly redacted data and DEP tasks lie here. **Upper left** (High pass@10 and few completions): most results for runs with no redaction and/or IND tasks lie here. **Lower left/middle** (Low pass@10 and few completions): most results for runs for EXT tasks lie here.

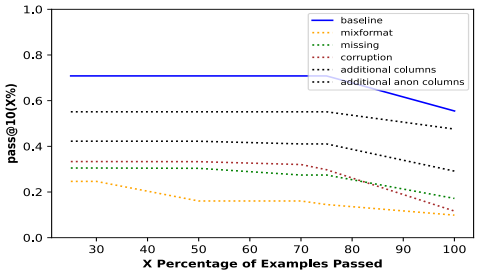
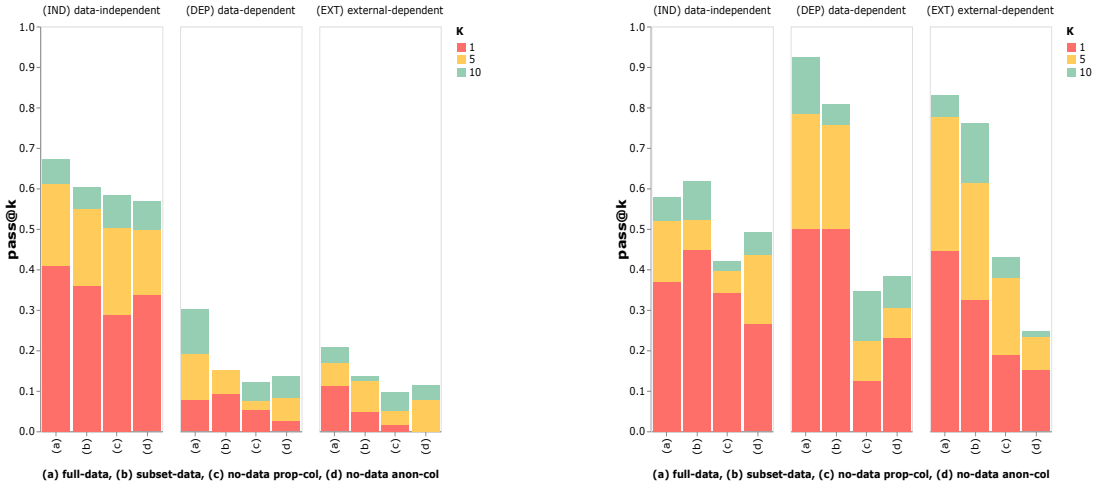
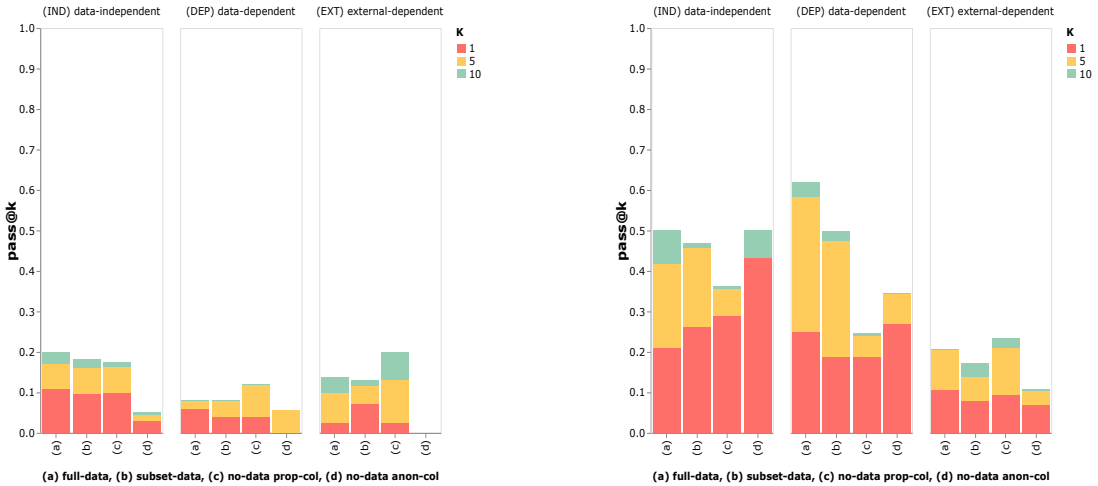


Figure 17: Impact on pass@10($X\%$) for the different noisy scenarios such as corruption in TYPESETNOISY compared to the original data in TYPESET (baseline) for INCODER.



SOFSET TYPESET

Figure 18: Impact on pass@k for CODEX with data redaction for SOFSET and TYPESET. The results shown are grouped by the different task classes: data-independent (IND), data-dependent (DEP) and external-dependent (EXT). In each group, the different bars represent different levels of redaction: (a) full data, (b) a subset of the data (only one row) (c) no data and proper column names and (d) no data and anonymous column names. Redaction has negligible effect on IND tasks, and a large impact on DEP and EXT tasks.



SOFSET TYPESET

Figure 19: Impact on pass@k for INCODER with data redaction for SOFSET and TYPESET. The results shown are grouped by the different task classes: data-independent (IND), data-dependent (DEP) and external-dependent (EXT). In each group, the different bars represent different levels of redaction: (a) full data, (b) a subset of the data (only one row) (c) no data and proper column names and (d) no data and anonymous column names. Redaction has negligible effect on IND tasks, and a large impact on DEP and EXT tasks.