Benchmarking Data Science Agents

Yuge Zhang[1] Qiyang Jiang[2] Xingyu Han[2] Nan Chen[1] Yuqing Yang[1] Kan Ren[2] *
[1]Microsoft Research, [2]ShanghaiTech University
Yuge.Zhang@microsoft.com, renkan@shanghaitech.edu.cn

Abstract

In the era of data-driven decision-making, the complexity of data analysis necessitates advanced expertise and tools of data science, presenting significant challenges even for specialists. Large Language Models (LLMs) have emerged as promising aids as data science agents, assisting humans in data analysis and processing. Yet their practical efficacy remains constrained by the varied demands of real-world applications and complicated analytical process. In this paper, we introduce DSEval – a novel evaluation paradigm, as well as a series of innovative benchmarks tailored for assessing the performance of these agents throughout the entire data science lifecycle. Incorporating a novel bootstrapped annotation method, we streamline dataset preparation, improve the evaluation coverage, and expand benchmarking comprehensiveness. Our findings uncover prevalent obstacles and provide critical insights to inform future advancements in the field.*

1 Introduction

Data science has become significant, as it helps individuals and organizations make informed decisions, predict trends, and improve processes by analyzing large volumes of data. Research on this topic continues to advance the field, driving innovations in machine learning, artificial intelligence, and big data analytics, thus enhancing its impact across various industries. However, data science requires extensive knowledge about analytical toolkits (e.g., NumPy and Pandas) and professional expertise to conduct analysis and correctly draw insights from data, which is challenging even for specialists.

Recent advancements in Large Language Model (LLM) (Brown et al., 2020; Touvron et al., 2023) and LLM-powered agents (Shen et al., 2023) have shown considerable potential in enhancing human

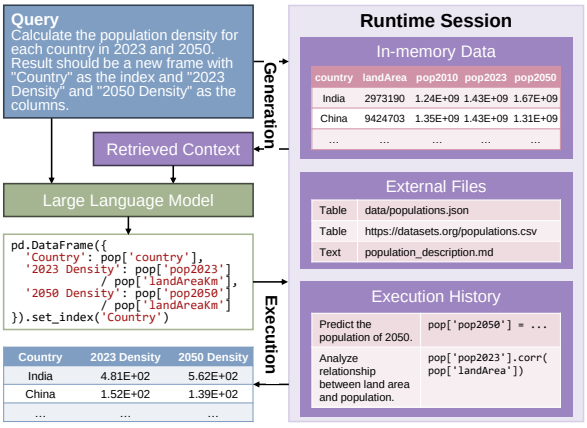


Figure 1: Illustration of a typical workflow of data science agents.

capabilities in data science. For instance, Code Interpreter[†] allows ChatGPT to perform data analysis and visualization by creating a sandboxed Python interpreter within the platform. Copilots integrated with Microsoft Excel and PowerBI[‡] assist users in exploring and understanding data and finding insights. Similar initiatives have also emerged in the open-source community, such as Jupyter AI (jupyterlab, 2023), Chapyter (chapyter, 2023), and CoML (Zhang et al., 2023a).

The tools mentioned are part of an emerging category of software known as *data science agents*, capable of executing a wide array of data-centric tasks, including manipulation, aggregation, visualization, and analysis, through natural language commands. These agents primarily utilize LLMs to produce and implement code within designated data science platforms, such as Excel. Essential to their operation is the ability to comprehend the context of data and files in the ongoing session, along with the capability to verify and amend outputs as necessary, as discussed in studies (Cheng et al., 2023; Zhang et al., 2023b; Tu et al., 2023; Chen

*Correspondence to Kan Ren.
*Source code and data are available at https://github.com/MetaCopilot/dseval.

[†]https://openai.com/blog/chatgpt-plugins
[‡]https://support.microsoft.com/en-us/copilot-excel

Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 5677–5700
August 11-16, 2024

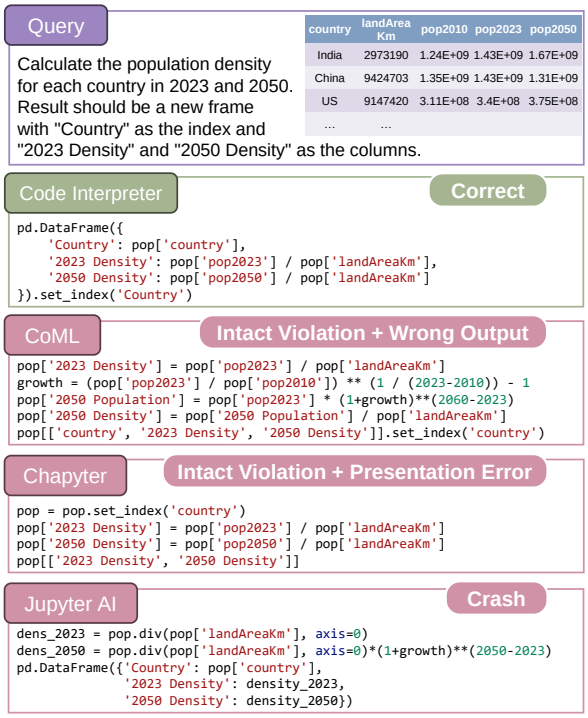


Figure 2: A sample problem to test current data science agents. Code Interpreter is the only agent that produces the correct code to answer the query. CoML neglects the existing "`pop2050`" column in the table and predicts the population of 2050 on its own, which is not desired. Chapyter fails to capitalize the index "`Country`" and unintentionally modifies the data ("pop"), violating intactness. Jupyter AI divides strings by integers and cannot automatically recover from such failures.

et al., 2023). Figure 1 depicts the typical workflow of a data science agent, highlighting its interactions with various components.

However, the reliability and accuracy of current data science agents can be inconsistent due to practical complexity of data science. For instance, when we subjected four different agents to the same query, as shown in Figure 2, only one provided the correct response. The errors made by the others ranged from overlooking a data frame column, misinterpreting data types, failing to adhere to specified output formats, to altering the original data. These discrepancies can stem from various issues, including LLM limitations, unclear or inaccessible context, or a lack of failure recovery mechanisms. Such shortcomings underscore the urgent need for focused research and enhancement of data science agents, with a particular emphasis on rigorous evaluation and benchmarking.

Evaluating data science agents is essential to pinpoint their capabilities and limitations, thereby informing future research trajectories. Yet, existing evaluation methodologies fall short of adequately addressing this need, being either insufficient or ill-suited for the task at hand. Some existing works (Zan et al., 2022; Lai et al., 2023) only deliver incomplete evaluations of simple code completion or in-filling capability of LLMs, neglecting the whole problem-solving ability of agents. Other recent works (Cheng et al., 2023; Dibia, 2023) perform evaluations either on a limited scale or in a biased manner, mainly due to the heavy human efforts for dataset construction and agent evaluation.

In this paper, we introduce a novel benchmarking framework designed specifically for evaluations of data science agents. Our contributions are three-fold. First, we propose DSEval, an evaluation paradigm that enlarges the evaluation scope to the full lifecycle of LLM-based data science agents. We also cover aspects including but not limited to the quality of the derived analytical solutions or machine learning models, as well as potential side effects such as unintentional changes to the original data. Second, we incorporate a novel bootstrapped annotation process letting LLM themselves generate and annotate the benchmarks with "human in the loop". A novel language (i.e., DSEAL) has been proposed and the derived four benchmarks have significantly improved the benchmark scalability and coverage, with largely reduced human labor. Third, based on DSEval and the four benchmarks, we conduct a comprehensive evaluation of various data science agents from different aspects. Our findings reveal the common challenges and limitations of the current works, providing useful insights and shedding light on future research on LLM-based data science agents.

2 Related Works

In this section, we provide a concise overview of the pertinent literature. Please refer to Appendix A for a detailed comparison table.

Evaluating Code Generation Models. The field of LLMs (Brown et al., 2020) has seen rapid progress, with many capable models that can generate high-quality natural language and codes for various domains and tasks (Chen et al., 2021; Roziere et al., 2023). Benchmarks for these models (Chen et al., 2021) have also emerged. Some of them are specifically designed for the data science domain, such as PandasEval / NumpyEval (Zan et al., 2022), DSP (Chandel et al., 2022) and DS-1000 (Lai et al., 2023). However, what these benchmarks provided

were pre-written prompts, mainly for a fair comparison of completion of in-filling abilities of LLMs themselves. They do not fully evaluate the skills of data science agents (Zhang et al., 2023a), such as handling natural language interactions, managing runtime sessions, and assembling prompts. Our evaluation scope is larger, which includes the full lifecycle of the agents.

Evaluating Agents. State-of-the-art LLMs (OpenAI, 2023) have been used to power autonomous agents (Significant-Gravitas, 2023; yoheinakajima, 2023; Wu et al., 2023), some of which get specialized in solving data science problems, such as data analysis, visualization and modeling (Li et al., 2023a; Qian et al., 2023a; Zhang et al., 2023a). However, there is a lack of rigorous and systematic evaluation methods for these agents. Some existing methods rely on huge human labor in problem collection and judgment, to assess the quality of the generated code or analysis (Cheng et al., 2023), which incurs significant cost and restricts the scalability of benchmarks. Some others resort to another more powerful LLM to score the output of the agent (Dubois et al., 2023; Dibia, 2023; Wang et al., 2023), which may introduce bias and overlook errors. Our work proposes a novel full-lifecycle evaluation paradigm to ensure robustness, and an additional LLM-bootstrapping annotation to enhance scalability and coverage.

3 DSEval: Evaluation Paradigm for Data Science Agents

To comprehensively and reliably evaluate a data science agent, we must first identify the evaluation scope, i.e., the "lifecycle" of an agent (§ 3.1). Then we propose a paradigm that monitors the full lifecycle for complete assessments (§ 3.2).

3.1 Evaluation Scope

We argue that a robust data science agent depends not solely on the LLM capabilities, but also on the design of its other constituent components. To identify the necessary scope for a comprehensive evaluation, we must first have a holistic perspective on the agent's lifecycle.

The lifecycle is depicted in the left part of Figure 3. First, the agent receives a "*query*" (expressed in natural language). Then it retrieves some additional contexts from a stateful "*runtime session*", which is usually hosted by a data analysis platform (e.g., Jupyter), containing information like variables, execution history, files, etc. A LLM-powered code generator then produces a code snippet based on the query and context. The code is sent back to the runtime session for execution to get the result. Optionally, a code fixer can help with error diagnosis and quality improvement (as done in tools like genai[§]). The lifecycle can repeat itself for multi-rounds, with the runtime session keeping track of the conversation and execution history.

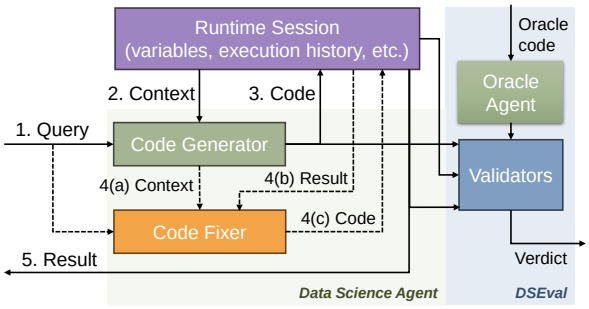


Figure 3: Agent lifecycle, monitored by DSEval. Our evaluation scope is the green-shadowed area. DSEval monitors the full lifecycle.

Our evaluation focuses on the holistic behavior of the data science agent, excluding implementation details of internal components such as code generators. We design benchmarks with queries and runtime sessions as the only inputs, which essentially differs from existing code generation benchmarks (Chen et al., 2021; Zan et al., 2022).

3.2 Full-Lifecycle Monitoring

The holistic view of the agent lifecycle also makes us rethink the evaluation paradigm, and we conclude that "*every step and component involved in the lifecycle must be continuously monitored*". For instance, imagine a query requiring in-place dataset modifications. Here, validating the runtime session is crucial to confirm the accurate execution. Hence, we design a validator module that is able to monitor the generated code, execution result, runtime session, etc. Meanwhile, the validator leverages an oracle agent equipped with a reference code snippet, provided by benchmarks for comparison. The process is illustrated in the right part of Figure 3.

The validator implementations within the validator module are fully modular, with each implementation focusing on a specific phase (e.g., data matching with fuzzy order, or evaluating trained model performance on a held-out test dataset). The full list and their usage frequencies are in § 5 and Appendix B. Notably, our focus is beyond correct-

[§] https://github.com/rgbkrk/genai

ness. For example, we implement an "Intact" validator, which tests whether the agent preserves the "intactness" of the session. We implement this due to the belief that minimizing unintended changes is one important criterion of safety and reliability.

4 Benchmarks based on DSEval

Building upon the DSEval evaluation paradigm, we initiated the data collection and benchmark development process. We came to realize that tremendous efforts were still required to properly rephrase queries, configure sessions, and adapt validators for each query. Simple format conversion proved insufficient due to limitations in existing data sources: some data sources lack real-world complexity (e.g., pandas-exercises (guipsamora, 2020)), while others address different-natured tasks (e.g., PandasEval (Zan et al., 2022)).

To ensure the benchmark coverage with limited human efforts, we developed an "LLM-bootstrapping annotation process", leveraging LLMs to automatically create problemsets based on a minimal "idea", while incorporating human input. This process is facilitated by the DSEAL (DSEval Annotation Language), which is designed to be compatible with the DSEval framework and easily comprehensible to LLMs. In this section, we first introduce DSEAL (§ 4.1), followed by a detailed description of the annotation process, including a Kaggle-inspired case study (§ 4.2).

4.1 DSEAL: DSEval Annotation Language

DSEAL is essentially a language to describe "problems". A *problem* in DSEAL corresponds to one iteration depicted in Figure 3, where a query is presented, and agents solve it and return results. We define a "*problemset*" as a sequence of interdependent problems, where later problems may have session or semantic dependencies on preceding ones. A benchmark comprises multiple "problemsets", each of which is self-contained and isolated.

The design of DSEAL is guided by three main objectives. Firstly, it must be compatible with the DSEval framework, ensuring that its components are expressive enough to fit within the framework. Secondly, it should be friendly to human annotators, for debuggability and ease of diagnosis. Lastly, it needs to be easily understandable by LLMs to leverage their power for annotation purposes.

To achieve these goals, we have designed DSEAL as an extended version of the Python language. Each problemset is represented as a Python (`*.py`) file, with problems separated by "`# %%`" (cell syntax[¶]). The code for oracle agents is written in Python, enabling direct execution and debugging using standard Python SDK. We use triple-quoted strings with YAML syntax inside to "configure" the problem, including the query, validator configurations, execution restrictions, and external data required. An example is provided in Figure 4.

```
# Previous problems...

# %%
"""
query: |
  Show the correlation between population
  density in 2023 and 2050, rounded to 2 decimals.
validator:
  template: basic
  namespace_intact:
    update: [pop]
  or:
    result:
      atol: 0
    output:
execution:
  forbid_names:
  - pop_heldout_test
  max_time: 0.5
data:
  pop.csv: https://.../pop.csv
"""
(pop['pop2023'] / pop['landAreaKm'])
  .corr(pop['pop2050'] / pop['landAreaKm']).round(2)

# Next problems...
```

Figure 4: An example problemset written in DSEAL (DSEval Annotation Language).

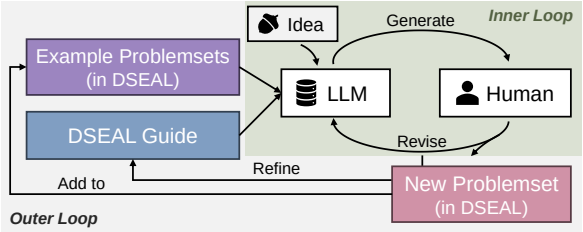


Figure 5: Illustration of the LLM-Bootstrapping Annotation Process.

4.2 LLM-Bootstrapping Annotation Process

To alleviate human labor, we leverage the capability of LLMs to automatically annotate the benchmark as bootstrapping. However, fully depending on LLMs may derive unreliable benchmarks even with state-of-the-art LLMs (further details are provided in the case study). Therefore, we incorporate "human-in-the-loop" to further enhance the annotation. The bootstrapping process involves an inner loop and an outer loop, as illustrated in Figure 5.

Inner Loop. To encourage LLMs to generate problemsets grounded in intended scenarios, we utilize "idea seeds". These seeds anchor the gen-

[¶] https://code.visualstudio.com/docs/datascience/jupyter-notebooks

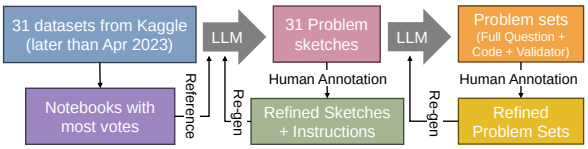


Figure 6: Illustration of the annotation process on DSEval-Kaggle.

Benchmark	# Sets	# Problems	Conversational	Realistic	Difficulty
DSEval-Exercise	21	187	✓	✗	17.3
DSEval-SO	202	202	✗	✓	16.1
DSEval-LeetCode	40	40	✗	✗	56.0
DSEval-Kaggle	31	396	✓	✓	35.9

Table 1: Overview of the four benchmarks.

erated problems to a specific scenario, promoting practicality and diversity across different outputs. Additionally, we prompt LLMs with a "guide" containing instructions to format the problemset with DSEAL and ensure clarity and challenge. Few-shot examples from existing problemsets further enhance quality (Kaplan et al., 2020).

Following the LLM's initial "bootstrapping" of a draft problemset, human experts step in for revision. Their focus lies in assessing clarity, diversity, and difficulty, and introducing necessary adjustments the LLM may struggle with independently. These adjustments can be partial, paving the way for the LLM to refine or enrich the problem set in an iterative loop.

Outer Loop. Once humans determine that no further adjustments are needed, the problemset is incorporated into the benchmark and serves as another example problemset of the LLM. Additionally, revision comments are leveraged to enhance the DSEAL guide, preventing similar issues in future. This loop, culminating in the accumulation of high-quality problem sets, exemplifies another form of "bootstrapping" within our process.

Case Study with DSEval-Kaggle. We selected 31 datasets published after April 2023, with data sizes less than 10 megabytes and more than 100 votes (by Sept. 2023). We attached the most-voted[‖] notebook associated with each dataset. These 31 datasets and notebooks serve as the "idea seeds".

The inner loop has two primary stages in this case. First, the targeted knowledge points and problemset sketches are created based on the dataset and notebook descriptions. Second, the full problemset is generated based on the sketch from the first stage. GPT-4 was used throughout the entire process. The illustration is in Figure 6.

In early experiments, we encountered the following issues when relying solely on GPT-4 to generate the problemset. (i) Lack of diversity due to repetitive generation results; for example, hypothesis test related queries appeared frequently. (ii) Deviation from the actual dataset content, neglecting crucial initial steps like data cleaning. (iii) Ambiguous queries resulting in vague or impossible-to-answer problems. (iv) Incorrect solutions or incorrect validator configurations. Interestingly, the first two issues can be effectively mitigated as the outer loop repeats. The other two require resolution within the inner loop (i.e., from human revisions).

We ensure that all problems are revised at least once by human annotators, thus guaranteeing the quality of the benchmark. The entire annotation process required approximately 2.32 million prompts and 187k completion tokens on GPT-4, as well as 20 human hours. We estimate a 3x reduction in human effort compared to purely manual methods like DS-1000 (Lai et al., 2023). More details about the annotation process can be found in subsection F.4.

5 Statistics and Coverage

Based on DSEval, we employed the annotation process to construct four benchmarks, detailed in Table 1. These benchmarks encompass problem sets with diverse properties, ranging from straightforward tasks to more intricate challenges. More technical details about how we created those benchmarks are available in Appendix F.

Validator Usages. Our evaluation process encompasses the entire lifecycle of data science agents. We employ a total of nine validators, each targeting distinct facets within the lifecycle. Details regarding their utilization are documented in Section B. Within our benchmarks, data science agents undergo validation through 6.5 validators per problem on average.

Problem difficulty. For a better understanding of the performance across different difficulty levels, similar to previous studies (Yu et al., 2018), we quantify code complexity by considering the number of function calls, expressions, conditions, and loops in the reference code for each problem. The distribution of problem difficulties is depicted in Figure 7, with the average difficulty detailed in Table 1. We observe that DSEval-LeetCode poses the highest level of difficulty, while DSEval-Kaggle exhibits the most diverse range of difficulty levels.

[‖] https://www.kaggle.com/code?sortBy=voteCount

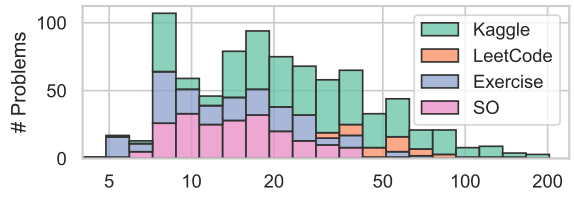
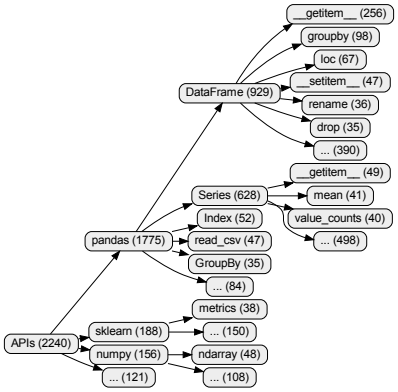
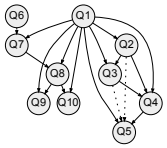


Figure 7: Difficulty distribution of the 4 benchmarks.



(a) Data science APIs involved in the problems. In the parenthesis are the number of appearances.



(b) Dependency graph of problemset "*student-performance*" in DSEval-Kaggle. Solid lines for "session dependencies" and dashed lines for "semantic dependencies".

Figure 8: API coverage and dependency graph example.

API coverage. Collectively, the four benchmarks covered 2240 API calls spanning 448 distinct APIs within the oracle code. These APIs are visualized in Figure 8a. Unsurprisingly, the most frequently utilized libraries are pandas, sklearn, and numpy. In total, 12 libraries are covered, with imblearn, nltk, statsmodels, and catboost being the least frequently employed. The most commonly occurring API is the `[]` operation of DataFrame, utilized for selecting indexes or columns.

Knowledge points coverage. We use GPT-3.5 to summarize the data science knowledge points essential for solving each problem. As illustrated in the word cloud of Figure 9, the benchmarks focus on fundamental data processing concepts such as data transformation, aggregation, filtering, sorting, and grouping, as well as encompassing machine learning concepts like outlier detection, imbalanced dataset handling, and feature selection.

Problem dependencies. DSEval-Kaggle and DSEval-Exercise are two conversational benchmarks where there could be interdependences among problems. We define "session dependency" as a scenario where a variable from a previous problem is used in a subsequent problem, and "semantic dependency" as a situation where the comprehension of a later query relies on the context of a preceding query. We visualize dependency graphs for

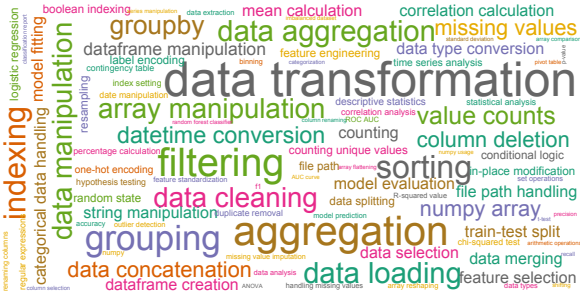


Figure 9: Knowledge points involved in the problems.

each problem set (see Figure 8b for example). On average, we observe an in-degree of 2.08 across all graphs. Regarding the maximum dependency chain length, the longest chain spans 10 dependencies, with an average chain length of 4.06.

Session contexts. A major challenge in our proposed benchmark lies in retrieving and representing contexts from runtime sessions. On average, we estimate that each problem involves 3.68 variables, with a maximum of up to 29. The total data size of these variables is 1.12 kilobytes at the median, and can reach up to 268 megabytes in extreme cases.

6 Evaluation

6.1 Setups

Error Categories. When an agent fails to successfully respond to a problem, the errors in an agent-generated code snippet can be classified into eight major categories, which can be further broken down into 32 subcategories. The complete catalog is presented in Figure 10 and Appendix D. Two common errors are highlighted below:

- *Presentation Error:* This occurs when the result is almost correct but problematic in terms of format or presentation approach. For example, the agent might fail to capitalize a column name as instructed or erroneously print results to the console instead of placing them in cell outputs.
- *Intact Violation:* Happens when the solution is almost correct except for violating the concept of intactness. This typically occurs when the computation requires some intermediate columns and the agent modifies the original data, which is unnecessary.

Metrics. The "*Pass Rate*", which is the number of problems passed divided by all problems in the benchmark, is the default metric used to assess the quality of an agent. By default, the runtime session is set to the ground-truth state before evaluating

	Framework	DSEval-Kaggle				DSEval-Exercise				DSEval-LeetCode			DSEval-SO		
		Pass Rate	Error Prop	w/o Intact	w/o PE	Pass Rate	Error Prop	w/o Intact	w/o PE	Pass Rate	w/o Intact	w/o PE	Pass Rate	w/o Intact	w/o PE
Single-Agent	Chapyter (chapyter, 2023)	34.1	26.0	35.6	55.3	39.6	28.3	42.2	70.6	45.0	45.0	60.0	46.5	48.5	59.9
	CoML (Zhang et al., 2023a)	59.8	56.8	61.1	63.6	78.6	78.6	79.1	81.3	42.5	42.5	62.5	78.2	79.7	79.7
	Code Interpreter API (shroominic, 2023)	42.4	41.7	43.9	47.0	67.4	67.9	68.4	71.7	45.0	45.0	55.0	58.4	68.3	65.8
	Jupyter-AI (jupyterlab, 2023)	51.8	38.4	52.8	58.1	78.6	55.1	79.1	81.8	57.5	57.5	70.0	50.0	50.0	56.4
Multi-	MetaGPT (Hong et al., 2023)	41.2	-	42.7	51.3	62.0	-	62.0	74.9	45.0	45.0	65.0	63.4	68.8	73.8
	ChatDev (Qian et al., 2023a)	-	-	-	-	-	-	-	-	32.5	32.5	50.0	35.1	35.1	37.6

Table 2: Performance of agent frameworks on DSEval benchmarks. We compare: pass rate, pass rate with error propagation, pass rate without the constraint of intact violation, and pass rate without considering presentation error. ChatDev is only evaluated on DSEval-LeetCode and DSEval-SO due to the difficulty of injecting complex context. MetaGPT is not evaluated under error propagation settings due to similar reasons.

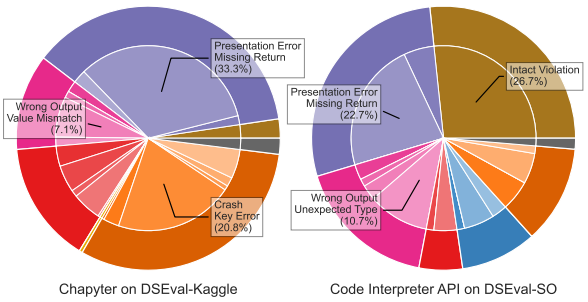


Figure 10: Two examples of error type breakdowns. More in Figure 14.

each problem. We refer to "*error propagation*" as a special setting where erroneous states accumulate to affect future problems within the same problem set. Additionally, we compute the pass rate while ignoring intact violations and presentation errors ("*w/o Intact*" and "*w/o PE*" respectively), as they can be considered correct in a looser setting.

6.2 Evaluating Data Science Agents

We evaluate 6 popular LLM-based agents that are currently applicable to data science scenarios: Chapyter, ChatDev, CoML, Code Interpreter API, Jupyter-AI, and MetaGPT (summarized in § C.1). These selected agents cover mainstream agent-building approaches, including function calls, expert knowledge, and multi-agent communications. For fair comparisons, we use GPT-3.5 (v1106) with a temperature of 0 as backend LLMs for all agents.

The key observations from Table 2 are as follows: (i) Chapyter is the worst-performing agent, but its pass rate significantly improves when presentation errors are ignored. (ii) CoML is the best for most benchmarks, except for LeetCode, where Jupyter-AI outperforms greatly. (iii) When errors propagate, Chapyter and Jupyter-AI suffer greatly, yet the other two frameworks remain stable. (iv) Intact violations sometimes occur, but not frequently. (v) While multi-agent frameworks incur significantly higher costs due to increased interaction rounds, they do not demonstrate clear advantages over single-agent frameworks.

To gain a better understanding of these error types, we did several case studies and visualized the percentages of error causes in Table 2 in Figure 10. We can see that the primary issue with Chapyter is missing returns (e.g., using "`print()`" instead of "`return`" to show the output) and key errors (e.g., referencing non-existing columns). Code Interpreter API on DSEval-SO often triggers intact violations, as the framework has a tendency to perform inplace modifications to existing variables.

6.3 Context Selection and Representations

Aiming to investigate the key factor that impacts the performance, we identify one fundamental difference among the agent frameworks, which is how they select and represent contexts from the sessions. Contexts are crucial for agents as they complement the missing parts of the query. Under the scope of our benchmarks, contexts are roughly categorized into variable descriptions and executed code history (see Figure 11 an illustrative example). The section delves into the selection and representation of contexts in prompts.

We conduct experiments with different combinations and orders of variable descriptions, code histories, and queries. We pick CoML as the baseline agent framework as it appears to be the best-performing one in previous experiments. The results are shown in Table 3. We observe that without any context, LLMs struggle to produce correct results. Providing code history and variable descriptions as context improve performance of agents. Code history seems to be more essential, especially for simpler tasks like DSEval-Exercise. The order of the context also has a slight impact: placing variable descriptions and queries at the end of the input tends to improve the results.

Context	DSEval-Kaggle		DSEval-Exercise	
	Pass Rate	w/ Error Prop	Pass Rate	w/ Error Prop
Q	13.9	13.9	13.9	13.9
C+Q	53.8	40.4	**81.3**	**80.7**
V+Q	52.3	51.5	73.3	71.1
C+V+Q	**61.4**	52.5	80.7	80.2
V+C+Q	59.8	**56.8**	78.6	78.6
Q+V+C	58.3	53.5	74.3	71.7

Table 3: Comparison of combinations in the context. "C" stands for "Code history", "V" stands for "Variable descriptions" and "Q" stands for "Query".

LIDA

```
pandas.DataFrame(shape=(5, 3),
columns=[{'column': 'name', 'properties':
{'dtype': 'string', 'samples': ['banana',
'elderberry', 'cherry', 'apple', 'durian'],
'num_unique_values': 5}}, {'column':
'price', 'properties': {'dtype': 'string',
'samples': ['$0.50', '$1.00', '$0.75',
'$1.20', '$2.50'], 'num_unique_values':
5}}, {'column': 'color', 'properties':
{'dtype': 'string', 'samples': ['yellow',
'purple', 'red', 'green'],
'num_unique_values': 4}}])
```

CoML

```
pandas.DataFrame(shape=(5, 3),
columns=["name", "price",
"color"])
          name  price   color
0        apple  $1.20     red
1       banana  $0.50  yellow
..         ...    ...     ...
3       durian  $2.50   green
4   elderberry  $1.00  purple
```

Figure 11: Illustration of data table formatter in LIDA and CoML.

Encoding the context into the prompt poses another challenge. Previous work (Sui et al., 2024) has explored this issue and proposed different methods to compress megabytes of data into dozens of tokens. We evaluate the approaches used in LIDA (Dibia, 2023) and CoML, with differences shown in Figure 11.

As shown in Table 4, LIDA and CoML have similar performance on DSEval-Kaggle, but LIDA outperforms CoML on DSEval-Exercise. This difference in performance could be due to LIDA encoding more information such as the data type and the unique-value count of each column. However, this also means that LIDA consumes more tokens than CoML to represent the same table.

6.4 Evaluating LLMs

We experimentally combine CoML with different LLMs and compare their performance. The results are shown in Figure 12. In addition to GPT-3.5, which we have already tried, we include four more models for comparison: GPT-4 (OpenAI, 2023), Gemini-Pro (Team et al., 2023), CodeLlama-7B (Roziere et al., 2023), and CodeLlama-34B. The rank of the models is approximately as follows: CodeLlama-7B $\approx$ CodeLlama-34B $<$ Gemini-Pro $<$ GPT-3.5 $<$ GPT-4. More details are in § C.4.

6.5 Self Repair

To evaluate the diagnostic and self-repair abilities of data science agents, we apply the self-debug (Chen et al., 2023) to the DSEval benchmarks. We use the CoML implementation, which

Format	DSEval-Kaggle		DSEval-Exercise	
	Pass Rate	# Tokens	Pass Rate	# Tokens
CoML	59.8	2963.7	78.6	2126.3
LIDA	59.8	4192.7	82.4	2547.6

Table 4: Comparison of pass rate and consumed prompt tokens for different code and data encodings in prompts.

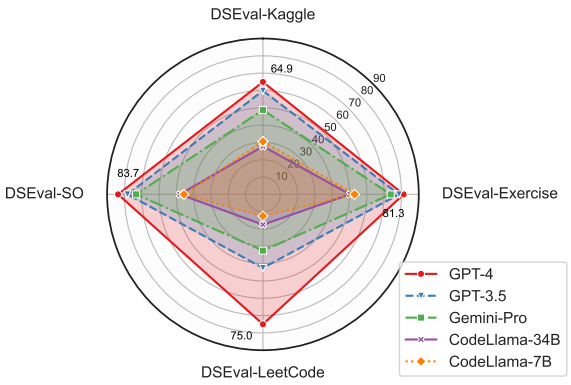


Figure 12: Performance of CoML combined with different LLMs on four benchmarks of DSEval.

sends the output and errors to LLMs for line-by-line analysis and feedback, before receiving a revised code. We do not use any hints from validators during this process. It repeats until we obtain a correct result or reach the maximum number of attempts. We also compare self-debug with a simple resampling baseline, which resamples a new code snippet if the previous one is incorrect.

Figure 13 shows two main findings. First, both self-debug and resampling enhance performance, but self-debug is generally more effective. Second, models with lower capabilities (e.g., GPT-3.5) can outperform models with higher capabilities (e.g., GPT-4) with enough self-repair attempts.

We also analyzed the error types that can be fixed via self-repair on DSEval-Kaggle and found that around half of them are "Crash" errors. Among all the "Crash" errors, 15% will still crash after the 4th attempt, and 41% will turn into other error types. Among all error types except "Crash", the type that is most likely to be fixed is "Presentation Error", with a fixed probability of 20% (4 / 20). This suggests there is room for improvement in current self-repairing techniques.

7 Conclusion

In this paper, we introduce DSEval, an evaluation paradigm for data science agents. Based on DSEval, we created 4 benchmarks that cover different aspects of data science tasks, and existing agents were evaluated and analyzed on the benchmarks.

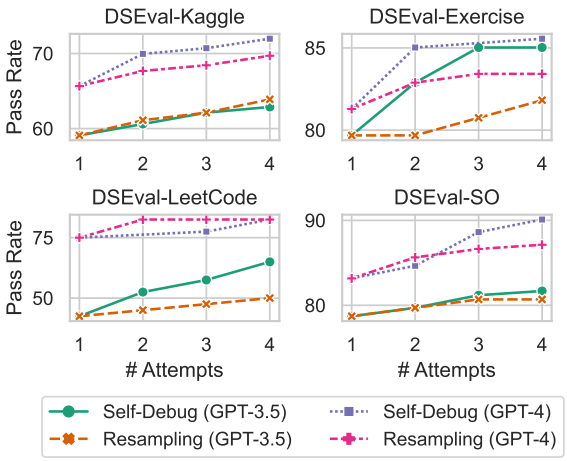


Figure 13: Self-debug versus vanilla resampling.

We aim to establish a standard for evaluating data science agents and we welcome more contributions of problemsets from the community.

8 Ethical Considerations

Modern data science agents have made it easier to analyze, visualize and process data. However, such agents can also pose serious risks if they are not used carefully. For example, a data science agent can alter the data without the user's awareness, or generate a misleading data analysis that appears to be correct but is actually erroneous.

Our work is the first to address these issues in a comprehensive way. For instance, we developed a validator that can track the full lifecycle of agent and assess whether the agent causes any unwanted changes (via "Intact" validator). We think future data science agents should follow our benchmarks as a reference, to ensure that they produce reliable and safe outcomes.

9 Limitations

Evaluating Planning Ability. The goal of planning is to break down a complex task into several simple, executable tasks, which is a key skill of LLM agents (Shen et al., 2023; Wu et al., 2023). In this paper, we focus on evaluating data science agents' performance on single tasks. Although some tasks (especially those in DSEval-Kaggle) are very complex and require careful planning to solve, we did not include high-level data science tasks that are vague and open-ended, such as "design a data pipeline that will win this Kaggle competition". However, we think that DSEval framework can also support those tasks, as long as the evaluation criteria (i.e., validator) are properly defined and configured.

Model	Temp	Repeat	Kaggle	Exercise	LeetCode	SO
CodeLlama-7B	0.0	0	30.6	52.9	12.5	45.5
CodeLlama-7B	0.0	1	30.3	52.9	12.5	45.5
CodeLlama-7B	0.5		30.8	46.0	15.0	47.0
CodeLlama-34B	0.0	0	27.8	50.3	17.5	48.0
CodeLlama-34B	0.0	1	27.8	49.7	10.0	48.0
CodeLlama-34B	0.5		25.5	48.1	30.0	45.5
Gemini-Pro	0.0	0	48.7	73.8	32.5	73.3
Gemini-Pro	0.0	1	47.2	73.3	32.5	73.3
Gemini-Pro	0.5		43.4	65.2	37.5	66.8
GPT-3.5	0.0	0	59.8	78.6	42.5	80.2
GPT-3.5	0.0	1	60.6	80.7	45.0	79.2
GPT-3.5	0.0	2	60.4	79.7	42.5	78.2
GPT-3.5	0.5		58.8	79.1	47.5	80.7
GPT-3.5 (v0613)	0.0		61.9	80.7	37.5	79.2
GPT-4	0.0	0	64.9	81.3	75.0	83.7
GPT-4	0.0	1	64.1	81.3	70.0	84.7
GPT-4	0.0	2	64.6	82.4	77.5	85.1
GPT-4	0.5		64.6	82.4	72.5	85.1
GPT-4-32k	0.0		65.4	80.2	67.5	80.7

Table 5: Reproducibility test by repeating the experiment, and possibly varying the temperature and model version. Default versions for GPT-3.5 and GPT-4 are v1106.

Reproducibility and Stableness. We conducted extensive evaluations and obtained some interesting insights, but unfortunately we could not repeat every experiment to check the reproducibility of each result due to the budget constraint. Instead, we focused more on evaluating different settings and benchmarks, which we believe are more informative. In Table 5, we verified some of the experiments by either repeating them, using a different model version, or changing a parameter (e.g., the temperature). We observed that the results are not very stable and can vary by up to $\pm 2\%$ even with a minimized temperature. On DSEval-LeetCode, the variation is even more significant, probably because the benchmark only has 40 problems. However, we remark that we have published 4 benchmarks based on DSEval, multiple results on different benchmarks can still have some significance. We encourage the community to adhere to the following guidelines to enhance the reproducibility:

1. **Repeat experiments.** Run the evaluation multiple times and report the average whenever the budget permits.

2. **Run all benchmarks.** The insights and findings should validated to be significant across all proposed benchmarks, to ensure that they have covered different aspects of the data science domain.

3. **Open source.** Make the logs and results public to ensure that they are reliable and trustworthy.

References

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

Shubham Chandel, Colin B Clement, Guillermo Serrato, and Neel Sundaresan. 2022. Training and evaluating a jupyter notebook data science assistant. *arXiv preprint arXiv:2201.12901*.

chapyter. 2023. chapyter. https://github.com/chapyter/chapyter.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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

Liying Cheng, Xingxuan Li, and Lidong Bing. 2023. Is gpt-4 a good data analyst? *arXiv preprint arXiv:2305.15038*.

Victor Dibia. 2023. Lida: A tool for automatic generation of grammar-agnostic visualizations and infographics using large language models. *arXiv preprint arXiv:2303.02927*.

Yann Dubois, Xuechen Li, Rohan Taori, Tianyi Zhang, Ishaan Gulrajani, Jimmy Ba, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Alpacafarm: A simulation framework for methods that learn from human feedback.

guipsamora. 2020. pandas_exercises. https://github.com/guipsamora/pandas_exercises.

Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Binhao Wu, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, Li Zhang, Lingyao Zhang, Min Yang, Mingchen Zhuge, Taicheng Guo, Tuo Zhou, Wei Tao, Wenyi Wang, Xiangru Tang, Xiangtao Lu, Xiawu Zheng, Xinbing Liang, Yaying Fei, Yuheng Cheng, Zongze Xu, and Chenglin Wu. 2024. Data interpreter: An llm agent for data science.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2023. Metagpt: Meta programming for a multi-agent collaborative framework.

jupyterlab. 2023. jupyter-ai. https://github.com/jupyterlab/jupyter-ai.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023a. Camel: Communicative agents for" mind" exploration of large scale language model society. *arXiv preprint arXiv:2303.17760*.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023b. Structured chain-of-thought prompting for code generation. *arXiv preprint arXiv:2305.06599*.

Jiachang Liu, Dinghan Shen, Yizhe Zhang, William B Dolan, Lawrence Carin, and Weizhu Chen. 2022. What makes good in-context examples for gpt-3? In *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, pages 100–114.

Harsha Nori, Yin Tat Lee, Sheng Zhang, Dean Carignan, Richard Edgar, Nicolo Fusi, Nicholas King, Jonathan Larson, Yuanzhi Li, Weishung Liu, Renqian Luo, Scott Mayer McKinney, Robert Osazuwa Ness, Hoifung Poon, Tao Qin, Naoto Usuyama, Chris White, and Eric Horvitz. 2023. Can generalist foundation models outcompete special-purpose tuning? case study in medicine.

OpenAI. 2023. Gpt-4 technical report.

Chen Qian, Xin Cong, Wei Liu, Cheng Yang, Weize Chen, Yusheng Su, Yufan Dang, Jiahao Li, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023a. Communicative agents for software development.

Chen Qian, Yufan Dang, Jiahao Li, Wei Liu, Weize Chen, Cheng Yang, Zhiyuan Liu, and Maosong Sun. 2023b. Experiential co-learning of software-developing agents.

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

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. HuggingGPT: Solving AI tasks with chatGPT and its friends in hugging face. In *Thirty-seventh Conference on Neural Information Processing Systems*.

shroominic. 2023. codeinterpreter-api. https://github.com/shroominic/codeinterpreter-api.

Significant-Gravitas. 2023. Autogpt. https://github.com/Significant-Gravitas/AutoGPT.

Yuan Sui, Mengyu Zhou, Mingjie Zhou, Shi Han, and Dongmei Zhang. 2024. Table meets llm: Can large language models understand structured table data? a benchmark and empirical study. In *The 17th ACM International Conference on Web Search and Data Mining (WSDM '24)*.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Xinming Tu, James Zou, Weijie J Su, and Linjun Zhang. 2023. What should data science education do with large language models? *arXiv preprint arXiv:2307.02792*.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. 2023. Mint: Evaluating llms in multi-turn interaction with tools and language feedback.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*.

yoheinakajima. 2023. babyagi. https://github.com/yoheinakajima/babyagi.

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

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. In *The 2022 International Joint Conference on Artificial Intelligence*.

Lei Zhang, Yuge Zhang, Kan Ren, Dongsheng Li, and Yuqing Yang. 2023a. Mlcopilot: Unleashing the power of large language models in solving machine learning tasks.

Wenqi Zhang, Yongliang Shen, Weiming Lu, and Yueting Zhuang. 2023b. Data-copilot: Bridging billions of data and humans with autonomous workflow. *arXiv preprint arXiv:2306.07209*.

A DSEval Compared to Other Evaluation Frameworks

We summarize the differences between DSEval and other recent frameworks in the table below.

Benchmark	Scale (# Problems)	Task Domain	Evaluation Target	Evaluation Methodology	Annotation
HumanEval (Chen et al., 2021)	164	General	LLMs' code completion abilities	Unit-tests for function correctness	Hand-written
PandasEval / HumanEval [4]	202	Pandas and NumPy	LLMs' code completion abilities	Unit-tests for result correctness	Rule-based filtering + manual cleaning
DSP (Chandel et al., 2022)	1119	General Data Science (e.g., Pandas, Scipy)	LLMs' code completion abilities	Unit-tests for result correctness	Rule-based filtering
DS-1000 (Lai et al., 2023)	1000	General Data Science (e.g., Pandas, Scipy)	LLMs' code completion / in-filling abilities	Unit-tests for results + Surface-form code checking	Human (1200 hours)
(Cheng et al., 2023)	1000	General Data Science (e.g., Pandas, Scipy)	Agents' ability in generating figures and analysis	3 aspects with human evaluation	Not needed
LIDA (Dibia, 2023)	57	Data Visualization	Agents' ability in visualizing data	6 aspects with GPT self-evaluation	Not needed
Ours	825	General Data Science (e.g., Pandas, Scipy)	Agents' full ability in handling data science problems	Full-lifecycle monitoring, 7 aspects + their combinations, configurable and automated	LLM-bootstrapping with human-in-the-loop (around 20 human hours)

Table 6: Comparison with other evaluation frameworks.

B Validator Implementations

In Table 7, we list the currently supported validator implementations and the purpose for each of them. We also show how many times each validator has appeared in the four benchmarks.

Name	Alias	Purpose	#
Crash	`error`	Fail if the generated code crashes.	825
Return-Val	`execute_result`	Fail if the executed result of generated code is not expected.	796
Variables	`namespace_check`	Fail if some variables are not correctly created or modified.	276
Unit-test	`table_test`	The defined function fail in at least one of the test cases.	136
ModelEval	`model`	Fail if the defined model does not satisfy the criteria.	26
Console	`stream_output`	Fail if the console output is not expected.	1
AnswerInCode	`answer_in_source`	Succeed if the answer to the query is shown within the generated code itself.	825
Intact	`namespace_intact`	Fail if some variables are unexpectedly modified, violating intactness constraints.	825
And	`and`	Fail if at least one of the sub-validators fails.	825
Or	`or`	Succeed if at least one of the sub-validators succeeds.	825

Table 7: Supported validators and their usage counts.

Every validator within our framework are designed to target specific aspects of an agent's lifecycle. When addressing a new problem, the problem writer is given the flexibility to select from existing validators, create new validators, or combine existing and new validators to meet all necessary criteria. However, in our benchmark construction process, we discovered that 99.6% of the problems could be constructed and thoroughly evaluated using the built-in validators alone, with only a minimal number requiring the creation of new validators. This indicates that our validators possess strong generability and completeness.

C Supplementary Evaluations

C.1 Introduction to Benchmarked Data Science Agents

We briefly introduce the benchmarked data science agents as below.

- Chapyter (chapyter, 2023): A JupyterLab extension translating natural language intentions into Python code with automatic execution. It generates codes based on some predefined examples as well as the conversation history.
- ChatDev (Qian et al., 2023a,b): A software development framework that operates through the communication between multiple agents, all powered by LLMs. It is non-trivial to adapt ChatDev into an interactive coding agent, thus we only tested it on DSEval-LeetCode.
- CoML (Zhang et al., 2023a): An interactive coding assistant specifically built for the assistance of data scientists and machine learning practitioners. It has incorporated few-shot examples (Brown et al.,

2020), session variable representations, and code history into the prompt, and also implemented an auto-fixer in case of errors.

- Code Interpreter API (shroominic, 2023): An open-sourced implementation of ChatGPT code interpreter. It uses a natural language chatbot as its primary interface. The code executor functions as an external tool.
- Jupyter-AI (jupyterlab, 2023): A helpful tool for calling LLMs within a notebook. The generation is purely based on history calls are does not rely on contextual information such as session variables.
- MetaGPT (Hong et al., 2023): A multi-agent framework that leverages role playing and communication techniques to realize the goal. Specifically, we used Data Interpreter (Hong et al., 2024) as our implementatiion since it is optimized for solving data-related problems.

C.2 Error Reason Analysis

From Figure 14, we can see that although Chapyter on DSEval-Kaggle and ChatDev on DSEval-Kaggle both suffer from presentation error, one is primarily due to missing return (e.g., using "`print()`" instead of "`return`" to show the output), the other is due to index match (e.g., naming the columns with a wrong name). The error cause of Jupyter-AI is rather diverse, with "wrong output" being the dominant cause. Code Interpreter API on DSEval-SO often triggers intact violation, as the framework has a tendency to perform inplace modifications to existing variables.

A detailed explanation of each error reason can be found in Appendix D.

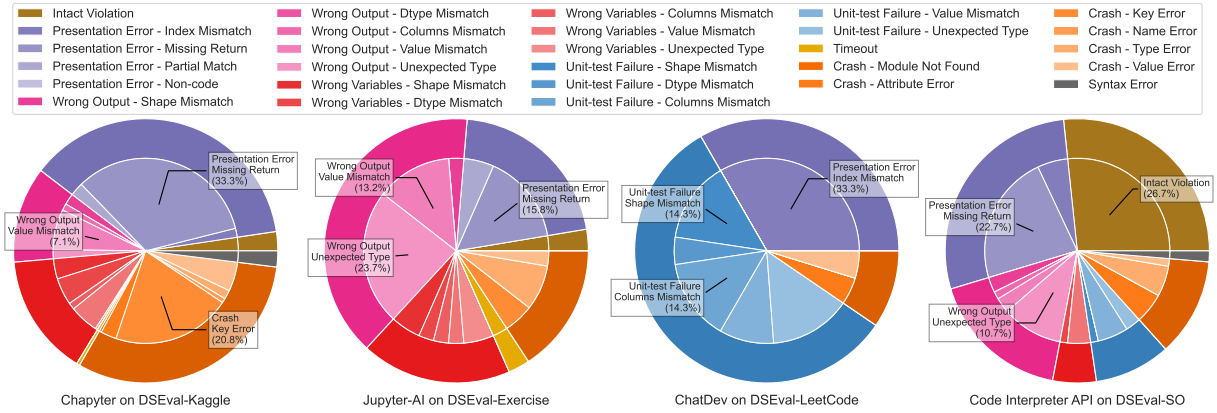


Figure 14: A catalog of all error reasons supported by DSEval (full explanations in Appendix D). The error causes of a selected subset of data science agents on the benchmarks are visualized in the pie charts.

C.3 Prompt Techniques

We incorporate various prompt techniques that are commonly used for different tasks into CoML for evaluation. Our goal is to identify the strengths and weaknesses of these techniques under data science scenarios.

Chain-of-thought. CoT (Wei et al., 2022) is a popular method for generating prompts that can handle various tasks. However, as shown in Table 8, CoT does not perform well on DSEval benchmarks as expected. A possible explanation is that the code itself already has a logical structure and does not require additional chain-of-thoughts. This result is consistent with recent works such as SCoT (Li et al., 2023b), which introduces CoT variants for code generation tasks. However, since most data science code lacks the "structure" of loops and conditions, adapting the method is challenging and we leave it as future work.

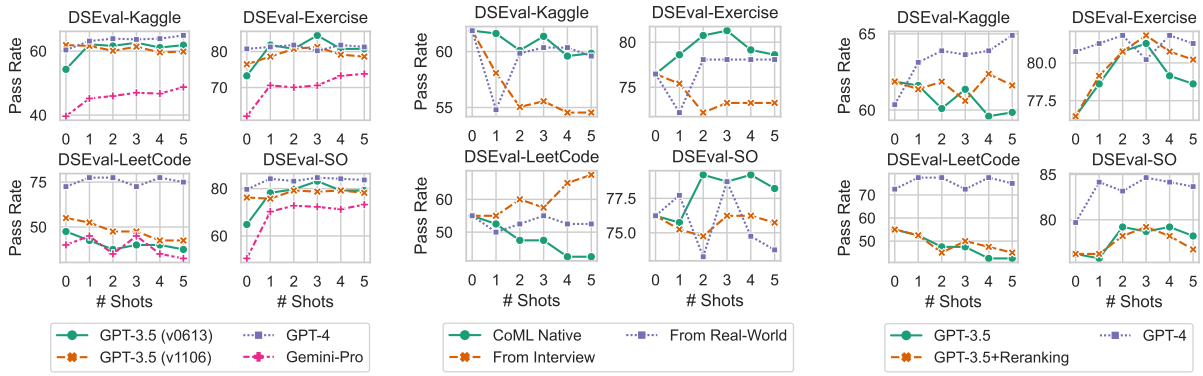
Prompt	Kaggle	Exercise	LeetCode	SO
CoML	**59.8**	78.6	42.5	**78.2**
CoML + COT	57.8	**80.2**	**45.0**	76.2

Table 8: Comparison of CoML w/ and w/o CoT.

Few-shot prompting. Few-shot prompting (Kaplan et al., 2020) is a method that uses demonstrations in prompts to help the model learn from the context. CoML uses a 5-shot prompt (5 demonstrations) by default to improve the quality of its generation. Few-shot prompting has the drawback of using more tokens (around 1k for 5 demonstrations). We want to see what happens when we use less demonstrations in the prompt.

In Figure 15a, we use a simple strategy, that is to keep the first k demonstrations in the order of appearance, where k is the number of demonstrations to keep. We ran the experiment with different backend LLMs, including two versions of GPT-3.5, GPT-4, and Gemini-Pro. The results show that the performance tends to get better with more shots (i.e., demonstrations). But there are also some exceptions. For instance, the pass rate of GPT-3.5 keeps going down on DSEval-LeetCode. On DSEval-Kaggle, the pass rate also fluctuates and the zero-shot performance is not worse than more shots.

We hypothesize that this phenomenon is because of a misalignment between the demonstrations and the benchmarks. The demonstrations in CoML are made with toy datasets and problems, which might not match the problems in each benchmark. In Figure 15b, we manually created two more sets of demonstrations. One is from real-world situations such as data processing and model training. The other is from interview questions, from platforms like LeetCode. We made sure that the demonstrations did not overlap with any problem in the benchmarks. As shown in Figure 15b, with demonstrations from interviews, DSEval-LeetCode benefits a lot from demonstrations. However, this set of demonstrations does not work well for other benchmarks. Demonstrations from real-world have an unstable performance and generally not satisfactory, implying that choosing the right demonstrations is a challenging issue in this scenario.



(a) With different LLMs. (b) Different sets of demonstrations. (c) Rerank the examples by relevance.

Figure 15: Impact of number of examples used for few-shot prompting.

Reranking. Another well-known technique that is often used in conjunction with few-shot prompting is reordering the demonstrations (Liu et al., 2022; Nori et al., 2023), which is also known as "similarity-based example selector" or "kNN-curated dynamic few-shot exemplar". In our setting, we reorder the demonstrations by their cosine similarity (on "text-embedding-ada-002") with the incoming query at inference time, and select the top k demonstrations as the k shots. The outcomes are presented in Figure 15c. Reordering the demonstrations is generally helpful for the performance, but the improvement is marginal. The performance is still much lower than a more powerful LLM (e.g., GPT-4).

C.4 Comparison of Different LLMs

As can be seen in Table 9, less capable models tend to suffer more from stricter evaluation settings (e.g., with error propagation). More capable models are also better at following instructions to preserve intactness or follow the desired format. For CodeLlama and Gemini, the pass rates can improve up to 7∼9% when loosing the presentation error, but for GPT-3.5 and GPT-4 the improvement is much smaller.

D Verdict Catalog

The following table summarize all the verdicts and subverdicts supported in DSEval. We refer to the code generated by the benchmarked agent as "*submission*" and the oracle code as "*reference*".

Model	DSEval-Kaggle				DSEval-Exercise				DSEval-LeetCode			DSEval-SO		
	Pass Rate	Error Prop	w/o Intact	w/o PE	Pass Rate	Error Prop	w/o Intact	w/o PE	Pass Rate	w/o Intact	w/o PE	Pass Rate	w/o Intact	w/o PE
CodeLlama-7B	30.6	24.5	31.6	37.9	52.9	44.4	53.5	56.7	12.5	12.5	22.5	45.5	47.0	53.0
CodeLlama-34B	27.8	17.9	28.8	39.4	50.3	43.3	50.3	59.9	17.5	17.5	25.0	48.0	48.5	55.9
Gemini-Pro	48.7	41.9	49.0	56.1	73.8	67.4	73.8	77.0	32.5	32.5	45.0	73.3	73.3	79.2
GPT-3.5	59.8	56.8	61.1	63.6	78.6	78.6	79.1	81.3	42.5	42.5	62.5	78.2	79.7	79.7
GPT-4	64.9	59.3	67.4	69.7	81.3	78.1	81.3	82.4	75.0	75.0	80.0	83.7	84.7	86.1

Table 9: Comparison of different LLMs. The metrics are: pass rate, pass rate with error propagation, pass rate without the constraint of intact violation, and pass rate without considering presentation error.

Verdict	Sub-verdict	Explanation	Example
Correct		Correct.	

Intact Violation		The submission violates the constraints of not modifying, updating or deleting existing variables unless necessary.	Q: What is the most dangerous decade to live in the US? Write it in the format of "19XXs" or "20XXs".
			`crimes['Total_Crimes'] = crimes.iloc[:, 1:].sum(axis=1)`
			`most_dangerous_decade = crimes['Total_Crimes'].idxmax()`
			`most_dangerous_decade.strftime("%Ys")`

Presentation Error	Index Mismatch	Only for DataFrame / Series outputs. The submission DataFrame / Series is correct, but has the wrong column names, incorrect index, or not properly sorted.	Count the number of fatalities for each year. Return a Series with "Year" as the index and "Number of Fatalities" as the values.
			`fatalities['date_of_event'] \`
			`  .dt.year.value_counts() \`
			`  .rename("Number of Fatalities")`

	Missing Return	The submission output is printed to the console output rather than put into the desired returning value.	Q: Show the first rows of the dataset. `print(df.head())`

	Partial Match	The desired output can be partially found within the submission output. For example, the reference output is a subset DataFrame of the submission, or the index of the submitted series, etc.	Q: List the names of the top 10 industries that have produced the most billionaires.
			`billionaires.groupby('industries') \`
			`  ['personName'].count() \`
			`  .sort_values(ascending=False). \`
			`  head(10)`

	Non-code	The submission generates plain texts (rather than code) to answer the query.	Q: What is the number of columns in the dataset? `The number of columns in the dataset is 5.`
Wrong Output	Shape Mismatch	Output is wrong in the shape of the DataFrame or array.	Q: Select all columns except the last 3. `euro12.iloc[: , 0:7]`
	Dtype Mismatch	Submission output is a DataFrame / Series and is wrong in the data type.	Q: Encode the categorical feature with label encoder and transform it into float. `LabelEncoder().fit_transform(x)`
	Columns Mismatch	Submission output is a DataFrame / Series and the column names are not expected.	Q: Remove excessive spaces from the column names. Save the cleaned dataset in-place. `netflix.columns = netflix.columns.str.strip()`
	Value Mismatch	Output is wrong in the data itself.	Q: Is there any duplicate dates? `apple.index.duplicated().any()`
	Unexpected Type		Q: Get a summary with the mean, min, max, std and quartiles of the dataset. `baby_names['Count'].describe()`
	Others	Uncategorized wrong output.	

<table>
<tr><td>Wrong Variables</td><td>Shape Mismatch
Dtype Mismatch
Columns Mismatch
Value Mismatch
Unexpected Type
Others</td><td>Variables are incorrect after execution of the submission code. Sub-verdicts are the same as “Wrong Output”.</td><td>Q: Add another column called place. The values of place are as follows: Bulbasaur is in park, Caterpie is in forest, Squirtle is in lake, Charmander is in street.
<pre>pokemon_col["place"] = [
  "park", "forest", "lake", "street"]</pre></td></tr>
<tr><td>Unit-test Failure</td><td>Shape Mismatch
Dtype Mismatch
Columns Mismatch
Value Mismatch
Unexpected Type
Others</td><td>The function in the submission did not pass the pre-defined unit-tests. Sub-verdicts are the same as “Wrong Output”.</td><td>Q: Write a sentiment prediction function called <code>predict_sentiment</code>. The function should take a review as input and return the predicted sentiment (“Positive”, “Negative”, or “Neutral”) as output.
<pre>def predict_sentiment(review):
  words = word_tokenize(review.lower())
  words = [word for word in words if
  ↪ word.isalpha() and word not in
  ↪ stopwords.words('english')]
  features =
  ↪ vectorizer.transform(words)
  return model.predict(features)</pre></td></tr>
<tr><td colspan="2">Timeout</td><td>The code fails to finish in the limited time. It could be due to endless loops or inefficiency.</td><td>Q: Use grid search to tune the hyperparameters of the random forest classifier. The time limit is 30 seconds.
<pre>param_grid = {
  'n_estimators': [100, 200, 300],
  'max_depth': [None, 10, 20],
  'min_samples_split': [2, 5, 10],
  'min_samples_leaf': [1, 5, 10],
  'bootstrap': [True, False],
  'criterion': ['gini', 'entropy']
}

grid_search = GridSearchCV(model,
↪ param_grid, cv=5, n_jobs=-1,
↪ verbose=1)
grid_search.fit(X_train, y_train)</pre></td></tr>
<tr><td rowspan="5">Crash</td><td>Module Not Found</td><td>Usually when the code fails to import a library.</td><td>Q: Conduct a chi-squared test to examine the relationship between “Gender” and “Discount Applied”. Show the chi-squared statistic.
<pre>import pandas as pd
import numpy as np
from stats import chi2_contingency
pd.crosstab(shopping['Gender'],
↪ shopping['Discount Applied'])
No module named 'stats'</pre></td></tr>
<tr><td>Attribute Error</td><td>Usually happens when referencing a non-existing method or attribute.</td><td>Q: Fill the missing values with NaN
<pre>salaries_growth_rate = \
  salaries_growth_rate \
  .fillna(value=pd.np.nan)</pre></td></tr>
<tr><td>Key Error</td><td>Usually happens when referencing a non-existing column.</td><td>Q: Select the third cell in the row named Arizona
<pre>army.loc["Arizona", 2]</pre></td></tr>
<tr><td>Name Error</td><td>Referencing an undefined variable or using an unimported API.</td><td>Q: Calculate the pearson correlation between the final worth and age of billionaires.
<pre>df['finalWorth'].corr(df['age'])
name 'df' is not undefined</pre></td></tr>
<tr><td>Type Error</td><td>Happens when a type is misused. For example, running numeric operations on string types.</td><td>Q: How many products have a unit cost more than $10.00?
<pre>chipo['item_price'] > 10
'>' not supported between instances
↪ of 'str' and 'int'</pre></td></tr>
</table>

	Value Error	Happens when operations can not process certain values.	Q: Compute the correlation of heart attack risk against other numeric features. Sort the factors by the absolute values of the correlation coefficients in descending order.
			`corr_matrix = heart.corr()`
			`corr_matrix.abs().sort_values(ascending=False)`
			`# could not convert string to float: 'BMW7812'`

	Others	Uncategorized Crash.	
Syntax Error		Code has syntax error.	

Table 10: Catalog of verdicts supported by DSEval. In case a solution is problematic from multiple perspectives, the verdicts from the bottom of the table have higher priorities to appear.

E Result Visualizer

We build a visualizer accompanying DSEval, to facilitate the examination and diagnosis of the results. A demonstration is shown in Figure 16.

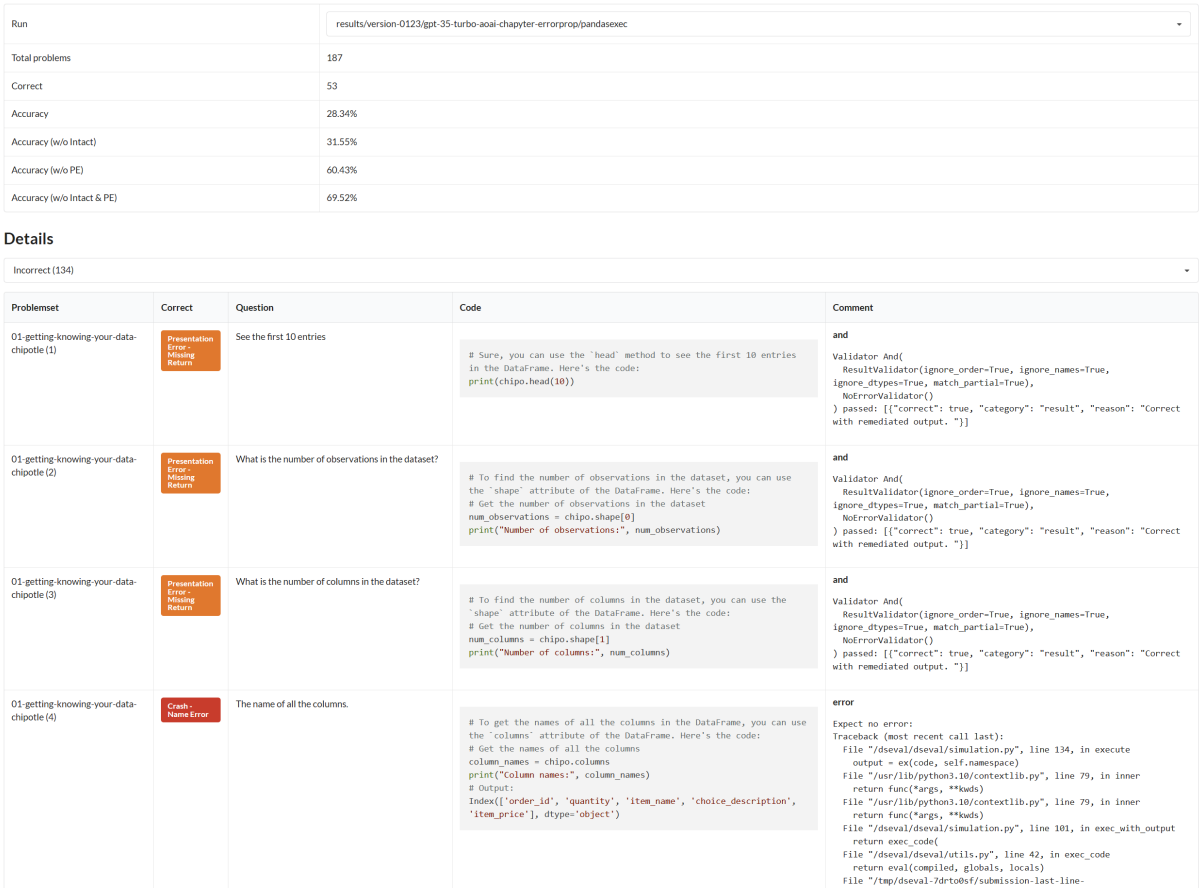


Figure 16: Result Visualizer.

F Benchmark Annotation Details

F.1 DSEval-Exercise

Pandas-exercises (guipsamora, 2020) contains 252 tutorial-purposed pandas questions organized in 27 notebooks. Each notebook contains multiple mutually correlated questions, featuring one specific theme, which could be filtering, sorting, time series, visualization and etc. The notebooks can be converted into DSEAL through a simple rule-based conversion script. We manually clarified some vague questions and corrected the validator configurations for each question to allow for proper error tolerances for some problems. Visualization problems are discarded due to validators of charts are not implemented,

which we left as future work. We end up selecting 187 problems from 21 problem sets, which we call "DSEval-Exercise".

F.2 DSEval-SO

CERT (Zan et al., 2022) presents PandasEval and NumpyEval, which altogether contain 202 pandas and numpy problems, collected from StackOverflow (SO). The queries are mostly related to usages of pandas or numpy APIs and tricks. Most answers are very short in terms of the number of operations involved. The original benchmarks are in the form of code completion. We manually clarified ambiguous queries and converted them into DSEAL with the help of GPT-4 (OpenAI, 2023).

F.3 DSEval-LeetCode

LeetCode has published several dozens of problems targetting data science areas[**]. By August 2023, 40 of them were available to free-tier accounts, which we crawled and converted into DSEAL with GPT-4.

We prompt GPT-4 to start the converted query by "Write a function `` `def ...:` ``", followed by the explanation of its inputs and outputs. We also instruct GPT-4 to include both the problem statement and sample inputs/outputs in our query part as we found that the query could often be ambiguous without the samples. "`table_test`" validators are used to validate the agents' output as the task of the agents is to write a function that can handle specific kinds of inputs. Since test cases and standard solutions are not obtainable directly from LeetCode, those parts are also written by GPT-4. The test cases are meant to cover the samples, scaled inputs, and some corner cases. To ensure the generated test cases were legal, we also asked the GPT-4 to generate a function to prevent illegal cases from coming into the benchmark.

After the conversion, we first submitted the generated solutions to the LeetCode platform for verification. 38 solutions were successfully submitted (as the other 2 turned out to require premium access), out of which 27 were correct. We manually fixed 10 out of 11 wrong solutions, while we found the other one mistakenly judged to be incorrect due to an invalid test case on LeetCode. We then tested the solutions on the auto-generated test cases, and manually corrected the tests, validators, or solutions where the results turned out to be incorrect. Finally, we used a data science agent (GPT-3.5 with CoML) for a trial run on this benchmark, and cross-checked our verdicts with the verdict on LeetCode online. We strengthened several test cases where our system returned correct and LeetCode returned otherwise. We show an example problem in DSEval-LeetCode in Appendix G.

F.4 DSEval-Kaggle

Here, we detail the annotation process and costs associated with the DSEval-Kaggle problemsets, building upon the process outlined in the main text.

The dataset comprises 31 problemsets. During the initial iteration, all 31 were generated without an exemplary problemset for reference. Human annotators then reviewed and selected one for revision, leaving the remaining 30 for further generation. This iterative process continued, with two problemsets revised at the second stage, two problemsets revised at the third stage, four problemsets revised at the third stage, until all 31 had undergone revision at the fifth stage. When the number of few-shot examples exceeded five, random selection was employed.

The total prompt token consumption for sketch generation amounted to 877 thousand tokens, while generating the full problemset consumed 1.44 million tokens.

We employed 2 human annotators, each with 1 year and 3 years of experience in the data science field, respectively. To ensure the high quality of our benchmark, we ensure all problemsets are examined and revised at least once (revision rate 100%). In terms of time required for revision, initially, annotators reported requiring 0.5 to 2 hours to revise each problemset sketch. However, this time decreased to "20 to 30 minutes" after adding more high-quality problemsets as examples for problem generation. To address controversial scenarios, DSEval was dry run on a vanilla CoML (Zhang et al., 2023a) agent with GPT-3.5 in our early experiments, and all controversial cases were discussed in a roundtable meeting among annotators and paper authors.

[**] https://leetcode.com/problemset/pandas/

The criterion for a "high-quality" problemset is shown below, which is meanwhile used as the prompt for LLMs to auto-generate or auto-revise the problemset.

Problem Sketch Instruction

Your task is to help a teacher design a problemset for an examination. The problemset is to test the students' ability to write Python code to solve data science problems (using numpy and pandas). The dataset that will be used for the problemset is pre-determined and shall be given by the user. You will also be provided a reference that might give you some ideas on what can be done with this dataset, but do not rely on it or copy it. You should write a sketch of the new problemset using the provided dataset. The sketch should include the following information:

- The knowledge points of the problemset: what knowledge points or programming skills are tested in the problemset?
- A sketch of the problemset: How many subproblems roughly are there in the problemset? what is each subproblem in the problemset about? How are the subproblems related to each other?

<DATASET DESCRIPTION>
For the new dataset mentioned above, please design a new problemset that is more difficult and more challenging than all the problemsets above, and write its desired knowledge points and sketch. Please follow the instructions below:

- The new problemset should also be different from the existing problemsets, i.e., it should not be a combination of existing problemsets.
- The new problemset should cover some new knowledge points or programming skills that are not covered by the existing problemsets.
- The problemset should contain roughly 10 - 15 problems.
- But try to follow the format of the existing problemsets.
- Problems with more logical thinking and reasoning challenges are preferred.
- Do not include visualization problems, system design problems, model training problems or open questions as I won't be able to automatically evaluate their results.
- Please do not be constrained by the ideas from existing problemsets. You can design a problemset that is novel, creative and interesting.

Full Problemset Writing Instruction

Your task is to help a teacher design a problemset for an examination. The problemset is to test the students' ability to write Python code to solve data science problems (using numpy and pandas). In particular, you are to write a full problemset based on a scratch.
The desired format is a Python file with multiple cells separated with "`# %%`". The first cell is some preparation code (e.g., import libraries like pandas), and the rest are the tasks. Each task consists of a docstring (containing question and validator) and a code block (containing the reference solution). The docstring is written in YAML, and the code block is written in Python.
Some extra instructions:

- Data files used in the problems are located under `inputs/`. You can use them in your problemset.
- If the sketch contains problems that are ambiguous or do not make sense, you can refine them. You can also add more problems to the problemset.
- When the sketch gives problem examples like "such as", "e.g.", "for example", etc., you can think of your own problem based on the given data. You don't need to follow the exact concrete problems given in the sketch.
- When using external data, you should use your knowledge to find the right URL on the Internet. You should write a separate question to read the data from online, and then use the data in the following questions.
- The result of each subproblem's reference code should ideally be a single value (e.g., a number, a string, a list, a dictionary, a dataframe, etc.). When students submit their code, the result of their code will be compared with the result of the reference code. If the results are the same, the student's code is considered correct. Otherwise, the student's code is considered incorrect.
- When manipulating the data and creating the features, try to adhere to the style and content of original data. For example, if the data columns are named in camel case, you should also name new columns in camel case. If the data only contains values between 0 and 1, you should not create a new feature that categorizes the data into 0-10, 10-20, etc.
- To make the comparison above possible, the result of the reference code should be the one and only possible answer to the question. Therefore, the question should be specific enough to have only one possible answer. For example, instead of asking "Provide a summary of the dataset", you should ask "What is the mean, std of the temperature anomalies of `dataset_a`? Put them in a tuple", or "return the results in a dataframe with columns `mean` and `std`", or "show the first 5 rows of the dataframe", etc. If the question is clear enough, please omit this.
- Use the validator only when necessary. For when and how to use the vaildators, please refer to the examples.
- Some problemset references are provided below. They are real-world problemsets that are used in data science courses. However, they are not the best examples of problemsets. You are encouraged to write better problemsets than them.

G Problem Examples

We provide a few examples for each benchmark here. The full benchmarks will also become publicly available.

G.1 DSEval-Kaggle

We show the first few problems from "*disease-symptoms-and-patient-profile-dataset*" in DSEval-Kaggle.

```
# %%
import pandas as pd
import numpy as np

# %%
"""
query: |
  Import the dataset from `inputs/Disease_symptom_and_patient_profile_dataset.csv`. Assign it to a
  ↪  variable called `disease`.

validator:
  namespace_check:
    disease:
"""

disease = pd.read_csv('inputs/Disease_symptom_and_patient_profile_dataset.csv')

# %%
"""
query: |
```

```
  Check the balance of the dataset. Count the number of positive and negative outcomes. Put them in a
  ↪  Series with "Positive" and "Negative" as the index.
"""

disease['Outcome Variable'].value_counts()

# %%
"""
query: |
  Handle the imbalance in the dataset using oversampling. Randomly duplicate some rows from the
  ↪  minority class to make it have the same number of rows as the majority class (use `resample` in
  ↪  sklearn with `random_state` 123 please). Save the balanced dataset in `disease_balanced`.

validator:
  namespace_check:
    disease_balanced:
      ignore_order: true
"""

from sklearn.utils import resample

# Separate majority and minority classes
df_majority = disease[disease['Outcome Variable']=='Positive']
df_minority = disease[disease['Outcome Variable']=='Negative']

# Upsample minority class
df_minority_upsampled = resample(df_minority,
                                 replace=True,     # sample with replacement
                                 n_samples=df_majority.shape[0],    # to match majority class
                                 random_state=123) # reproducible results

# Combine majority class with upsampled minority class
disease_balanced = pd.concat([df_majority, df_minority_upsampled])

# %%
"""
query: |
  Convert binary features into indicator (0/1) variables, and other categorical features (except
  ↪  "Disease" column) into numerical features using one-hot encoding. Save the encoded dataset
  ↪  in-place.

validator:
  namespace_check:
    disease_balanced:
"""

for column in ['Fever', 'Cough', 'Fatigue', 'Difficulty Breathing']:
    disease_balanced[column] = disease_balanced[column].map({'Yes': 1, 'No': 0})
disease_balanced['Outcome Variable'] = disease_balanced['Outcome Variable'].map({'Positive': 1,
↪  'Negative': 0})

categorical_columns = [column for column in disease_balanced.columns if
↪  disease_balanced[column].dtype == 'object' and column != "Disease"]
disease_balanced = pd.get_dummies(disease_balanced, columns=categorical_columns)

# %%
"""
query: |
  Let's assume the name of disease irrelevant for the following case study.
  Split the dataset into training and test sets. The test size should be 20% of the whole dataset.
  ↪  Random state should be set to 42. Use `X_train`, `y_train` to store the training set and
  ↪  `X_test`, `y_test` for test set.

validator:
  namespace_check:
    X_train:
    y_train:
    X_test:
    y_test:
"""
```

```
from sklearn.model_selection import train_test_split

X = disease_balanced.drop(['Outcome Variable', 'Disease'], axis=1)
y = disease_balanced['Outcome Variable']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# ... more problems omitted
```

G.2 DSEval-Exercise

Part of problem set "*02-filtering-sorting-euro12*" from DSEval-Exercise.

```
# %%
import pandas as pd

# %%
"""
query: |
  Import the dataset from `inputs/euro12.csv`.
  Assign it to a variable called euro12.

validator:
  namespace_check:
    euro12:

data:
  euro12.csv:
  ↪  https://raw.githubusercontent.com/guipsamora/pandas_exercises/master/02_Filtering_%26_Sorting/Euro12/Euro_2012_
"""

euro12 = pd.read_csv('inputs/euro12.csv', sep=',')

# %%
"""
query: Select only the Goal column.
"""

euro12.Goals

# %%
"""
query: How many team participated in the Euro2012?
"""

euro12.shape[0]

# %%
"""
query: What is the number of columns in the dataset?
"""

euro12.info()

# %%
"""
query: View only the columns Team, Yellow Cards and Red Cards and assign them to a dataframe called
↪  discipline

validator:
  namespace_check:
    discipline:
"""

discipline = euro12[['Team', 'Yellow Cards', 'Red Cards']]

# ... more problems omitted
```

G.3 DSEval-LeetCode

Problem "*duplicate-emails*" from DSEval-LeetCode.

```
# %%

import pandas as pd

# %%

"""
query: |
  Write a function `def duplicate_emails(person: pd.DataFrame) -> pd.DataFrame`.

  `person` is a DataFrame with the following columns:
  - id: int
  - email: str
  `person` contains an email for each record. The emails will not contain uppercase letters.

  The function should return all the duplicate emails. Note that it's guaranteed that the email field
  ↪  is not NULL. Return the result table in **any order**.

  The result format is in the following example.

  Example input:
  ```
  person:
  +----+---------+
  | id | email |
  +----+---------+
1	a@b.com
2	c@d.com
3	a@b.com
  +----+---------+
  ```

  Example output:
  ```
  +---------+
  | email |
  +---------+
  | a@b.com |
  +---------+
  ```

  Example explanation: a@b.com is repeated two times.

validator:
  table_test:
    function_name: duplicate_emails
    input_validator: |
      def _validate(person):
        assert person.shape[0] > 0
        assert person.dtypes.equals(pd.Series({'id': 'int64', 'email': 'object'}))
        assert person.id.is_unique
        assert person.email.str.match(r'^[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,}$').all()

    output_checker:
      ignore_order: true

    test_cases:
    - # example
      - "`pd.DataFrame({'id': [1, 2, 3], 'email': ['a@b.com', 'c@d.com', 'a@b.com']})`"
    - # corner case: only one email
      - "`pd.DataFrame({'id': [1], 'email': ['a@b.com']})`"
    - # corner case: all emails are the same
      - "`pd.DataFrame({'id': [1, 2, 3], 'email': ['a@b.com', 'a@b.com', 'a@b.com']})`"
    - # corner case: all emails are different
      - "`pd.DataFrame({'id': [1, 2, 3], 'email': ['a@b.com', 'c@d.com', 'e@f.com']})`"
    - # corner case: some emails are the same
      - "`pd.DataFrame({'id': [1, 2, 3, 4], 'email': ['a@b.com', 'c@d.com', 'a@b.com', 'c@d.com']})`"
```

```
    - # corner case: some emails are the same, but not all
      - "`pd.DataFrame({'id': [1, 2, 3, 4, 5], 'email': ['a@b.com', 'c@d.com', 'a@b.com', 'c@d.com',
      ↪  'e@f.com']})`"
"""

def duplicate_emails(person: pd.DataFrame) -> pd.DataFrame:
    # Group by email and count the occurrences
    email_counts = person.groupby("email").size().reset_index(name="count")

    # Filter the emails with count greater than 1 (duplicates)
    duplicates = email_counts[email_counts["count"] > 1]

    # Return the duplicate emails as a DataFrame
    return duplicates[["email"]]
```

G.4 DSEval-SO

Problem "*numpyeval-001*" from DSEval-SO.

```
# %%
import numpy as np
from numpy import newaxis

a = np.array([[1, 2, 3], [3, 4, 5], [5, 6, 7]])

# %%
"""
query: |
  I have a 2d array with shape (x, y) which I want to convert to a 3d array with shape (x, y, 1).
  Is there a nice Pythonic way to do this?
"""

a[:, :, newaxis]
```