DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation

Chengxi Li[* 1] **Yuhang Lai**[* 1] **Yiming Wang**[* 2] **Tianyi Zhang**[* 3] **Ruiqi Zhong**[* 4]
Luke Zettlemoyer[5 6] **Scott Wen-tau Yih**[7] **Daniel Fried**[8] **Sida Wang**[7] **Tao Yu**[1 5]

Abstract

We introduce DS-1000, a code generation benchmark with a thousand data science problems spanning seven Python libraries, such as `NumPy` and `Pandas`. Compared to prior works, DS-1000 incorporates three core features. First, our problems reflect diverse, realistic, and practical use cases since we collected them from StackOverflow. Second, our automatic evaluation is reliable – only 1.8% of the Codex-002-predicted solutions accepted by our evaluation are incorrect; we achieve this with multi-criteria metrics, checking both functional correctness by running test cases and surface form constraints by restricting API usages or keywords. Finally, we proactively defend against memorization by perturbing our problems; consequently, models pretrained on StackOverflow cannot rely on solutions memorized during pretraining. The current best public system (Codex-002) achieves 34.6% accuracy, leaving ample room for improvement. We release our benchmark at `https://ds1000-code-gen.github.io`.

1. Introduction

Data science is important in many areas (Romero & Ventura, 2013; Bolyen et al., 2019; Faghmous & Kumar, 2014), but requires programming proficiency in specialized libraries, thus posing substantial barriers to lay users. Fortunately, these barriers could potentially be reduced by pretrained code generation models: for example, Codex (Chen et al., 2021a) can complete small Python snippets with non-trivial accuracy and AlphaCode (Li et al., 2022) can tackle difficult competitive programming problems. We anticipate these barriers will diminish if the community can make solid progress in applying these models to data science problems.

However, we currently lack a benchmark that 1) focuses on everyday data science applications, 2) includes naturalistic intents and contexts, and 3) has a reliable execution-based evaluation metric. Most of the existing datasets with reliable test cases (Hendrycks et al., 2021; Chen et al., 2021a) focus on competition or interview-style programming problems; they measure algorithmic understanding but do not target real-world usage. Also, as represented by e.g., user problems on StackOverflow, users' data science coding problems usually have diverse contexts including their incorrect code, error messages, and input-output examples, which cannot be found in most prior data science relevant code generation benchmarks (Yin et al., 2018; Hendrycks et al., 2021; Chandel et al., 2022b; Chen et al., 2021a). Moreover, most of these benchmarks solely rely on surface form metrics such as BLEU or CodeBLEU (Yin et al., 2018; Agashe et al., 2019; Chen et al., 2021b). These metrics diverge from the programmer's intent, increasingly so as model capability improves (Zhong et al., 2020). To our knowledge, no existing benchmarks contain both naturally occurring problems with diverse contexts and reliable evaluation metrics.

To fill this gap, we introduce DS-1000, a benchmark with a thousand problems covering seven widely-used Python data science libraries: `NumPy`, `Pandas`, `TensorFlow`, `PyTorch`, `SciPy`, `Scikit-learn`, and `Matplotlib`. We highlight three core features of DS-1000: 1) it contains realistic problems with diverse contexts, 2) it implements reliable multi-criteria execution-based evaluation metrics, and 3) it proactively defends against memorization. We outline how we achieved each of them below.

First, we collected naturally occurring problems from StackOverflow, manually scored their representativeness and usefulness, and curated a subset of them to create our benchmark. While inputs in existing code generation datasets are either highly structured (problems or code context) or restricted in scope, our natural problems are diverse in content and format. For example, users might inquire about specific API usage (Figure 14), search for more efficient code implementations (Figure 1), provide incorrect code with an error message and ask for bug fixes (Figure 13), or ask for code that implements functionality they specify with input-output examples (Figure 1). These problems better reflect real-world applications and open up new modeling

[*]Equal contribution [1]The University of Hong Kong [2]Peking University [3]Stanford University [4]UC Berkeley [5]University of Washington [6]Meta AI [7]Facebook AI Research [8]Carnegie Mellon University. Correspondence to: Tao Yu <tyu@cs.hku.hk>.

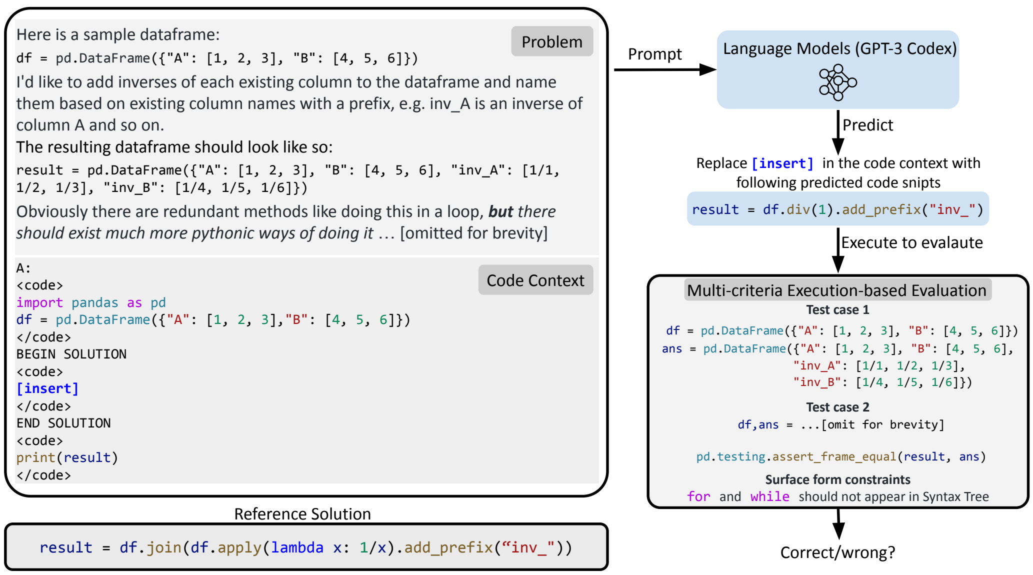


Figure 1: An example problem in DS-1000. The model needs to fill in the code into "[insert]" in the prompt on the left; the code will then be executed to pass the multi-criteria automatic evaluation, which includes the test cases and the surface form constraints; a reference solution is provided at the bottom left.

challenges, which have been understudied in existing code generation benchmarks.

Second, it is challenging to evaluate program solutions to natural and diverse problems reliably. Unlike competition-style problems, natural problems might lack executable contexts and test cases, allow multiple solutions, depend on external libraries, etc. To address these challenges, five of the authors of this paper, all proficient in data science and experts in Python, hand-adapted the original problems by writing executable code contexts, rewriting problems to be specific enough to be testable, and implementing automatic multi-criteria execution-based evaluation using carefully written and reviewed test cases and constraints that check functional correctness and surface form constraints. On program solutions predicted by Codex-002, we find that only 1.8% of the predicted programs passing our evaluation are incorrect (false discovery rate), indicating that our evaluation is reliable.

Third, one potential concern for adapting public problems is that the models might simply memorize the corresponding solution during pre-training time (Carlini et al., 2021a). We show in Section 2.4 that this can indeed happen: while Codex achieves 72.5% accuracy on the popular `numpy-100` dataset, the accuracy drastically drops to 23.5% after perturbing them without increasing their difficulty. Therefore, while building DS-1000, we proactively took measures against memorization by perturbing each problem.

Figure 1 shows an example DS-1000 problem, its reference solution, and an expert-written automatic multi-criteria evaluation. To answer the problem, the model needs to fill in the solution; to pass our automatic evaluation, it needs to 1) return the correct output and 2) avoid inefficient implementations that use for-loops.

We use DS-1000 to evaluate several popular code generation models, including Codex (Chen et al., 2021a), CodeGen (Nijkamp et al., 2022), and InCoder (Fried et al., 2022). We found model performance ranges from 9.0% to 34.6%, with Codex-002 model being the best. This implies that these models have the potential to reduce the barrier for data analysis, yet still have a large room for improvement.

2. Benchmark Construction

Our pipeline for building DS-1000 contains five stages, illustrated in Figure 2 and described below. 1) We scraped and selected high-quality problems from StackOverflow (Section 2.1). 2) We rewrote the problem and the reference solution so that the problem is unambiguous and the reference solution is executable.[$^{Y}_{M}$ Section 2.2] 3) We implemented a multi-criteria automatic evaluation for each problem, which includes test cases and surface form constraints (Section 2.3). 4) We performed a pilot study which shows that Codex can answer problems by memorizing the pre-training corpus, and proactively took measures to prevent this by perturbing the problems and their reference solutions in DS-1000 (Section 2.4). 5) We improved our multi-criteria evaluation by requiring it to reject a small set of sample predictions that we considered incorrect via manual review (Section 2.5), and then calculated the false discovery rate of our metric on a larger set of sample predictions. To reliably carry out this data collection procedure,

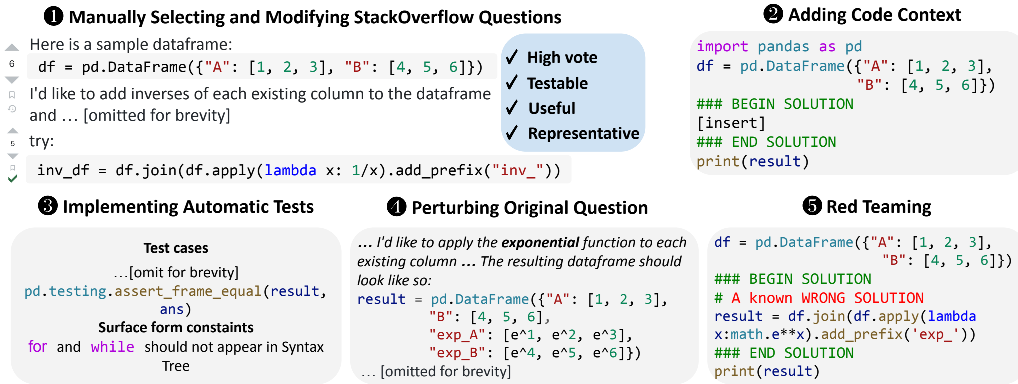


Figure 2: The pipeline for building DS-1000. See the start of Section 2 for a detailed description.

five authors who are computer science students and familiar with data science spent a total of about 1200 hours constructing DS-1000 (including steps from problem selection to quality review).

2.1. Problem Selection

Sourcing Popular StackOverflow Problems. To obtain natural and high-quality problems, we scraped data from StackOverflow under each library tag (e.g., "NumPy"). To select popular problems, we first removed duplicates and selected problems with at least 1 vote, 1000 views, that had an accepted answer. Next, we ranked problems based on votes and views and calibrated these statistics based on the time a problem was created since older problems naturally have more views and votes. We refer readers to Appendix A.1 for more details. Among the filtered problems, we randomly sampled an initial pool containing 4500 problems (1000 for NumPy, Pandas, and Matplotlib, 500 for Scikit-learn and SciPy, 250 for TensorFlow, and 250 for PyTorch).

Filtering Suitable Problems. To select problems from the above pool for our benchmark, our annotators scored each problem according to the following rubric: whether a problem a) contains input-output examples in the problem, b) is difficult to predict the solution for models according to the annotators' judgment, c) is practically useful, d) has a clear description, and e) is feasible to evaluate the solution. We aggregated these scores, reranked the candidate problems, and incorporated the top-ranked ones to create DS-1000. We ended up with 451 unique StackOverflow problems. More than half of the original StackOverflow problems were filtered out because they ask for an explanation for an algorithm or general content (see Appendix A.1).

Controlling Library Version. Data science libraries are continuously evolving. As a result, the semantics of the problem is determined not only by the language description but also by the software environment (e.g., library version). For example, the same code snippet, `tf.math.reciprocal(A)`, is only valid in the newer version of TensorFlow. We fixed the evaluation environment to include the latest libraries version that can be installed with python 3.7.10 and present the detailed documentation in Appendix A.1.

2.2. Rewriting problems and Their Reference Solutions

Creating Executable Context. To implement an execution-based evaluation for each natural language problem, we needed to write an executable context. We first added package imports and defined the variables described in the problem. For example, in Figure 2, we imported the Pandas package and created the data frame described in the problem as part of the context. Second, we needed to specify the desired behavior of the target program to be predicted. For example, in Figure 2, a code generation model can infer from the context that the resulting data frame should be named as `result`, rather than `output`.

Rewriting Matplotlib problems. Many Matplotlib problems on StackOverflow clarify their problems with example figures, which, however, cannot be encoded by current pre-trained code models. Therefore, we rewrote the StackOverflow problems in symbols (i.e., code and text) and adopted a different format from other libraries (see Figure 15).

Collecting Reference Solutions. Finally, we obtained the reference solution for each problem from multiple high-vote replies, edited all reference solutions to be executable given the context we provided, and fixed errors whenever we noticed them (e.g., Figure 11). Even though we did not use the reference solution in DS-1000 for evaluation, we provide them in DS-1000 to facilitate future research.

2.3. Implementing Multi-Criteria Evaluations

Our automatic evaluation is multi-criteria, checking both functional correctness and surface form constraints.

Perturbation	Categories	Example
Surface	Convert to completing function	Figure 16, change format of code context
	Paraphrase the description of the problem	Figure 17, express the same problem in different words
	Change the example input and output	Figure 18, replace this example with a longer one
Semantic	Replace keywords with analogy words	Figure 19, replace 'inv' with 'exp'
	Change the required index	Figure 20, need the specified rows and columns
	Reverse the order of the list, string or dataframe	Figure 21, reverse the needed string
	Change the type of the required result	Figure 22, change the DataFrame to a Series
Difficult Rewrite	Combining several surface and semantic perturbations	Figure 23, change examples and replace 'highest' with 'lowest'
	Digging more perturbations that increase the difficulty	Figure 24, hypothesis testing

Table 1: The perturbation categories along with examples. "Surface" perturbations do not change the reference solution, while "Semantic" perturbations do.

Functional Correctness. To evaluate functional correctness, we construct test cases by converting the input-output examples provided in the StackOverflow problem; then the expert annotators manually wrote additional test cases to improve the evaluation. To evaluate a predicted program, we execute it on the test inputs and compare the outputs with the ground truth.

However, checking the exact equivalence of outputs can inadvertently reject correct programs. Many problems involve floating point arithmetic, and many return values are acceptable since they are close to the ground truth answer, but they are not exactly equal. Some problems require random outputs, e.g., generating 100 samples from a distribution, and even executing the reference solution twice can lead to different results. Many problems do not fully specify all the parameters, e.g., the color scheme for the output figure in the `Matplotlib` library, or the hyper-parameters of a learning algorithm in `Scikit-learn`; therefore, programs with different parameters can satisfy the requirement, returning values that are different. In all these cases, we relied on the best judgment of our expert annotators to implement the metric for each problem, which sometimes involves complicated techniques, such as using statistical tests to handle randomness. See more examples in Appendix A.2.

Surface Form Constraints. Functional correctness alone is insufficient. For example, vectorized operations can be expanded using for-loops, which, however, are inefficient and do not meet the requirement of the problem. Therefore, we introduced additional surface form constraints that require the presence/absence of specific APIs for keywords. Notably, such a check is different from the standard surface form metrics such as CodeBLEU (Ren et al., 2020), which requires the whole model prediction to be uniformly similar to a reference solution; instead, DS-1000 precisely targets small but important parts of surface form.

2.4. Perturbation to Defend Against Memorization

Many models are pre-trained on web text and hence memorize its content (Elangovan et al., 2021; Carlini et al., 2021b); therefore, they might answer our problems correctly by simply recalling the solutions seen during pretraining if they were trained on StackOverflow or derivative sites. We demonstrate this effect on `numpy-100`, [1] a problem set of 100 `NumPy` problems with solutions that are copied several thousand times on GitHub. When prompted to answer a selected subset of 20 problems, Codex-002 achieves 72.5% pass@1 accuracy.[2]

However, if the model truly knows how to solve those problems, it should be able to solve similar problems at the same level of difficulty. This motivates us to perturb the problems in two ways: surface perturbations and semantic perturbations. For surface form perturbations, we paraphrased the problem or modified the code context in the problem, but the reference solution should stay the same after the perturbation; for example, changing from "Create a 5x5 matrix . . . " to "I need a matrix sized 5x5 . . . ". For semantic perturbations, we changed the semantics of the reference solution without changing its difficulty ; for example, asking for "*min*" instead of "*max*" in the problem. We provide more detailed categories in Table 1. In all of these cases, the difficulty of the problem does not change for humans.

Origin	Surface	Semantic	Avg. Perturbation
72.5	50.8	23.6	40.6

Table 2: The performance of Codex-002 on `numpy-100`.

We manually applied these perturbations to `numpy-100` and show the result on Table 2. Although the difficulty level remains the same for people, the performance of Codex-002 drops to 40.6% after perturbation (50.8% on surface form perturbations and 23.5% on semantic perturbations). Fur-

[1] `https://github.com/rougier/numpy-100`

[2] The fraction of Codex-002 samples that are correct.

	Pandas	NumPy	Plotting	Scikit-learn	SciPy	TensorFlow	PyTorch	Total/Avg.
Problem	291	220	155	115	106	45	68	1000
Origin	100	97	111	46	58	17	22	451
Surface Perturbation	24	22	0	57	11	11	27	152
Semantic Perturbation	88	51	44	9	20	12	11	235
Difficult Rewrite	79	50	0	3	17	5	8	162
% Surface Form Constraints	12.0	36.4	0	27.8	17.9	20.0	27.9	19.4
Avg. Test Cases	1.7	2.0	1.0	1.5	1.6	1.6	1.7	1.6
Avg. Problem Words	184.8	137.5	21.1	147.3	192.4	133.3	133.4	140.0
Avg. Lines of Code Context	9.0	8.3	6.9	11.0	10.2	9.2	9.0	8.9
Avg. Lines of Code Solution	5.4	2.5	3.0	3.3	3.1	4.1	2.1	3.6

Table 3: Detailed statistics of DS-1000.

thermore, in 36% of the cases, the model still predicted the original answer of the problem before the semantic perturbation, implying that the model is solving the original problems by memorizing their corresponding solutions. Therefore, we could significantly overestimate model performance if we test them on problems directly taken from the web. (See Appendix B for more details)

Therefore, to proactively prevent memorization, we applied the above two perturbations to DS-1000 problems. Perturbation is a labor-intensive process. Even for a simple perturbation from *min* to *max*, our annotators needed to edit all mentions of *min*, *smallest*, *minimum* to make the problem coherent, and updated the code context, reference solution, and our evaluation metric accordingly.

Finally, to make DS-1000 more challenging, we additionally introduced several semantic perturbations that increase the difficulty on purpose ("Difficult Rewrite" in Table 1).

2.5. Quality Assurance

To ensure the quality of our benchmark, each problem, reference solution, and automatic multi-criteria evaluation were reviewed by at least three expert annotators familiar with the library. Additionally, we "red teamed" our automatic evaluation by requiring it to reject all programs known to be incorrect, e.g., solutions to semantically perturbed problems (see Figure 2). After the quality review, we also quantitatively measured the evaluation quality by examing whether our multi-criteria automatic metric can reject incorrect Codex-002 predictions (more details in Section 3).

3. Dataset Statistics

We provide detailed dataset statistics in Table 3. DS-1000 contains 1000 problems originating from 451 unique StackOverflow problems. To defend against potential memorization, more than half of the DS-1000 problems are modified from the original StackOverflow problems (Section 2.4); they include 152 surface perturbations, 235 semantic perturbations, and 162 difficult rewrites.

DS-1000 has carefully designed testing methods, checking both execution semantics and surface form constraints. For each problem, there are 1.6 test cases (manually annotated corner test cases) on average, and 19.4% of them are accompanied by a surface form constraint. The average of problem words in DS-1000 is 140. On average, the reference solution contains 3.6 lines of code. Table 3 shows the library breakdown statistics: Most packages have a similar distribution except Plotting because we adopted a different problem format due to its multimodal nature.

Table 4 compares DS-1000 to other datasets. Notably, the average number of words per problem in DS-1000 is much larger than other data science related datasets (e.g., DSP, Chandel et al. 2022a and CoNaLa, Yin et al. 2018). More importantly, the problems in DS-1000 represent more diverse and naturalistic intent and context formats that cannot be seen in any other datasets. Unlike generic Python code generation benchmarks (MBPP, Austin et al. 2021 and HumanEval, Chen et al. 2021a), we note that data science code generation benchmarks have fewer test cases since the annotators need to define program inputs with complex objects such as square matrices, classifiers, or dataframes than simple primitives, such as floats or lists. Nevertheless, as we will show next, even a few test cases suffice for DS-1000.

We evaluate our multi-criteria automatic metric by checking whether it can reject incorrect solutions. We randomly sampled 10 problems from each library and sampled 40 predictions from Codex-002 for each problem (2800 problem-code examples in total). We run our automatic metric on all the sample predictions, review the predictions manually, calculate how often they disagree, and report the following

Dataset	Problems	Evaluation	Avg. Test Cases	Avg. P Words	Avg. Lines of Code Solution	Data Source
HumanEval	164	Test Cases	7.7	23.0	6.3	Hand-Written
MBPP	974	Test Cases	3.0	15.7	6.7	Hand-Written
APPS	10000	Test Cases	13.2	293.2	18.0	Competitions
JuICe	1981	Exact Match + BLEU	-	57.2	3.3	Notebooks
DSP	1119	Test Cases	2.1	71.9	4.5	Notebooks
CoNaLa	2879	BLEU	-	13.8	1.1	StackOverflow
DS-1000	1000	Test Cases + Surface Form Constraints	1.6	140.0	3.6	StackOverflow

Table 4: Comparison of DS-1000 to other benchmarks. The first three benchmarks target general Python usage and the next three involve data science code generation. DS-1000 adapts realistic problems from StackOverflow and checks both execution semantics and surface form constraints.

four quantities:

- Sample Level False Discovery Rate: among all predicted samples that **pass** our automatic evaluation, 1.8% of them are **incorrect** according to our annotator.
- Sample Level False Omission Rate: among all predicted samples that **do not pass** our automatic evaluation, 0.5% of them are **correct** according to our annotator.
- Problem Level False Positive Percentage: among all problems, 5.7% of the problems contain at least one incorrect sample prediction that pass our automatic metric.
- Problem Level False Negative Percentage: among all problems, 5.7% (it happens to be the same as the above) problems contain at least one correct sample prediction that fails to pass our automatic metric.

Generally, problem-level measures are especially stringent since they require correctly judging all predictions among the 40 sample predictions. While an apple-to-apple comparison with other datasets is not possible due to the difference in the underlying model and benchmark construction method (as a point of reference, Li et al. (2022) find the problem Level False Positive Percentage to be 60% on APPS (Hendrycks et al., 2021)), these measures reflect that DS-1000 is reliable.[3]

4. Benchmarking State of the Art Models

We used DS-1000 to benchmark five pre-trained code models from three different families. The best model Codex-002 Insertion achieves 34.6% accuracy, indicating room for improvement. We also show the results on the perturbed and unperturbed examples in Section 5.1.

4.1. Prompt Format

We provide an official prompt format in DS-1000 because it significantly impacts the performance of pretrained language models (Zhao et al., 2021). Figure 1 shows an example: each prompt starts with a natural language description and then provides a code context; the code context uses HTML-like markers to indicate the location of missing code that a model needs to fill in and provides both left and the right context to the missing code pieces.

We decide to use infilling as our official format because the right context is important to specify the behavior of the program predictions (e.g. the variable name for the result). More broadly, given that 1) infilling is an important functionality for real-world programming and 2) there is a growing trend in pre-training with the right context (Aghajanyan et al., 2022; Fried et al., 2022; Bavarian et al., 2022; Tay et al., 2022), we expect more future pre-trained models to perform infilling.

On the other hand, given that many current language models trained on code are not yet capable of infilling, we also provide an official prompt that transfers the right context information into the left context (Figure 25 and 26). Nevertheless, despite our best effort to design the prompts for left-to-right models, they still lag behind models with infilling capabilities (Section 4.3). We conjecture that infilling models are inherently more effective at utilizing the right context information. Finally, we only have the Completion format for Matplotlib problems because Matplotlib provides global access to the current figure so the right context is not necessary.

From now on, we refer to the infilling prompt format as Insertion format and the left-context-only format as Com-

[3] APPS often requires comprehensive analysis of problems, demanding solutions dealing with all potential scenarios. Thus, insufficient test coverage probably happens though there are more test cases (Li et al., 2022). We "red teamed" 10 problems from DSP and found 3 of them can accept incorrect programs.

Format	Model	Pandas	NumPy	Plotting	Scikit-learn	SciPy	TensorFlow	PyTorch	Overall
Left-to-right Completion	Codex-002	20.8	35.1	47.0	31.6	26.2	29.2	33.0	31.0
	Codex-001	6.3	20.6	30.5	11.2	14.0	13.2	10.1	15.0
	Codex-Cushman	4.6	13.7	33.4	10.4	7.4	9.6	11.5	12.7
	CodeGen-6B	1.8	7.4	17.9	3.4	4.3	7.3	3.5	6.3
	InCoder-6B	2.0	3.1	13.5	2.7	2.8	3.1	2.9	4.3
Insertion	Codex-002	22.7	40.4	47.0*	39.6	28.8	35.9	38.5	34.6
	InCoder-6B	1.6	0.9	13.5*	3.1	3.1	3.8	1.9	3.8

Table 5: pass@1 accuracy with 40 samples generated for each problem. The upper part shows accuracy on the left-to-right completion format, while the lower part shows the results of the insertion format. The rightmost "Overall" columns show the average accuracy on 1000 problems from all libraries. DS-1000 is able to differentiate the capabilities of different models and there is substantial room for improvement even for the best Codex-002 model. *: Plotting problems do not have the right context so completion and insertion formats are the same.

pletion format.

4.2. Experimental Setup

Models. We experiment with three families of pretrained models: Codex, InCoder (Fried et al., 2022), and CodeGen (Nijkamp et al., 2022). For the Codex models, we experiment with codex-davinci-002 (Codex-002), codex-davinci-001 (Codex-001), and codex-cushman-001 (Codex-Cushman). For InCoder and CodeGen, we experiment with the 6B parameters models. Among these models, Codex and CodeGen models are trained to predict the right context while InCoder models are trained for both left-to-right generation and infilling. In addition, Codex-002 also supports infilling, although the exact model training details are not disclosed.

Implementation Details. In all of our experiments, we generate 40 samples for each DS-1000 problem with temperature set to 0.7, top-p cutoff set to 0.95, and max generation length set to 1024. We set the stop sequence tokens to "</code>" as shown in Figure 1. These samples are used in the unbiased estimator of pass@1. [$^{\text{Y}}_{\text{L}}$ need InCoder and CodeGen details.] For DS-1000, evaluating generated codes does not require special computational resources like GPUs.

4.3. Main Results

Table 5 displays the pass@1 accuracy on DS-1000. We find that DS-1000 can differentiate models with different capabilities. The best model Codex-002 achieves a nontrivial but far-from-perfect average accuracy of 34.6%, indicating substantial room for improvement. In contrast, other models like CodeGen-6B or InCoder-6B have much worse performance, with accuracy lower than 5% on many libraries. Qualitatively, these smaller models often cannot correctly follow the prompt instruction, generating additional comments instead of the required code. Future ablation is needed to understand the underlying cause for this performance gap, which could be the difference in model size, lack of instruction tuning, or the difference in pre-training data.

In addition, we observe that model accuracy varies across different libraries. This speaks to the importance of a holistic evaluation of multiple data science packages because performance in a specific library may not directly generalize to other libraries.

Moreover, we find that the insertion format often leads to better performance. The same Codex-002 model has a 3.6% average accuracy improvement when used with the insertion format than used with the completion format. This shows the importance of the infilling capability for data science code completion.

5. Analysis

5.1. Results by Perturbation

In Section 2.4, we demonstrated the risk of memorizing the solutions on the `numpy-100` problem set; do we observe the same effect on DS-1000? To investigate this, we apply surface perturbations (i.e., the problem changes but the reference solution does not change) and semantic perturbations (the reference will change) to the problems in DS-1000.

Table 6 shows the results.[4] The performance of Codex-002 drops after perturbation (3.5% on surface form perturbations

[4]Note that the results are not comparable to Table 5 since for each kind of perturbation, we only selected a subset of problems to perturb.

	Pandas	NumPy	Scikit-learn	SciPy	TensorFlow	PyTorch	Overall
Origin$_{\text{surface}}$	33.4	54.8	40.4	33.2	47.7	52.3	43.6
Surface	27.3 −6.2	50.5 −4.3	41.1 +0.6	28.9 −4.3	42.3 −5.5	44.4 −7.9	40.1 −3.5
Origin$_{\text{semantic}}$	28.3	47.9	42.5*	32.6	52.5	49.8	37.5
Semantic	24.6 −3.7	41.0 −6.9	27.5* −15.0	25.3 −7.4	27.5 −25.0	22.3 −27.5	29.3 −8.2
Origin$_{\text{difficult}}$	31.0	49.4	10.0*	48.4	66.5*	60.3*	40.7
Difficult Rewrite	12.0 −19.1	21.4 −28.0	0.8* −9.2	13.7 −34.7	14.5* −52.0	25.9* −34.4	15.6 −25.1

Table 6: Effect of three different types of problem perturbation. In each subsection, we compare the accuracy of the perturbed problems to that of the original problems. We observe that although Surface and Semantic perturbations also cause a performance drop on DS-1000 the performance drop is much smaller compared to that on `numpy-100`. *: Numbers are averaged from less than 10 problems.

and 8.2% on semantic perturbations) but the drop is much less severe than what we observed on `numpy-100`. This indirectly suggests that Codex-002 might have memorized the solution for some StackOverflow problems, but the effect is less severe because they have not been repeated as often as `numpy-100` on the internet. Still, we believe problem perturbation to be a useful strategy to defend against memorization by future models proactively.

Additionally, we rewrite some problems to create more DS-1000 problems by intentionally making them more difficult even for human programmers. As expected, Codex-002 performs much worse after the rewrite, and we plan to use these problems as a challenge for future models.

5.2. Error Analysis

We provide a preliminary error analysis by showing an example model error in Figure 3 and provide additional examples in Figure 27 and 28. In this example, the problem asks for removing adjacent duplicated non-zero values in a given array, which cannot be satisfied by a single NumPy operation. The reference implements this problem by creating a binary array representing the selection and performing two operations to meet the problem requirement. However, we see Codex-002 fails on the composite request and attempts to answer the problem with a single method, `np.unique`, which the problem itself points out is incorrect. This example error demonstrates the challenges in DS-1000 problems, which require both natural language understanding and code generation abilities.

6. Related Work

Natural Language to Code. Research on translating natural language to executable forms dates back several decades. The models have become increasingly capable of producing complex and general programs while requiring fewer human annotations. Zelle & Mooney (1996) and Zettlemoyer & Collins (2007) translate natural language queries to domain-specific database queries. Liang et al. (2013) and Berant et al. (2013) parse natural language into first-order logic to answer generic knowledge-based questions. Yu et al. (2018); Scholak et al. (2021) translate natural language questions to general SQL programs and develop models that can generalize across domains. While all the works above still need to train their models on the task they evaluate, recently Li et al. (2022); Chen et al. (2021a) show that generative models pre-trained on code can produce Python snippets to tackle competitive programming challenges, without any additional human annotations. Many other recent works corroborated this finding (Nijkamp et al., 2022; Fried et al., 2022; Xu et al., 2022; Black et al., 2022), and additional techniques at inference time further improve the performance (Poesia et al., 2022; Shi et al., 2022).

Code Generation Benchmarks. As models become increasingly capable, researchers start to build increasingly difficult and general code generation benchmarks. While Zelle & Mooney (1996) focused only on domain-specific languages, Yu et al. (2018) builds a Text-to-SQL benchmark that evaluates the capability to write broad-domain SQL programs. Yin et al. (2018) evaluates the capability to write short but general Python snippets, while more recent papers Hendrycks et al. (2021); Li et al. (2022) evaluate models' capability to solve competitive programming problems in Python. If code generation models continue to improve, we expect future researchers to focus on more complex tasks.

At the same time, however, it becomes more difficult to build reliable benchmarks aligned with real-world applications. Programs are most useful when they are executed; therefore, we need to evaluate their denotational semantics, and the best general method so far is still to ask experts to manually write test cases. Consequently, most benchmarks with test cases focus on competition/interview/ programming chal-

Problem:
Given a numpy array, I wish to remove the adjacent (before removing) duplicate non-zero value and all the zero value.

For instance, for an array like that: [0,0,1,1,1,2,2,0,1,3,3,3], *I'd like to transform it to:* [1,2,1,3]. *Do you know how to do it?*

I just know `np.unique(arr)` *but it would remove all the duplicate value and keep the zero value. Thank you in advance!*

Reference Solution

```
# a: 1-d np.array as input
selection = np.ones(len(a), dtype = bool)
selection[1:] = a[1:] != a[:-1]
selection &= a != 0
result = a[selection]
```

Wrong Solution

```
# Just mimic mentioned wrong solution
result = np.unique(a)
```

Figure 3: An example model mistake. The problem specifies a composite requirement, removing adjacent non-zero duplicates, which cannot be solved by a single operation. The model mistakenly generates a single operation that removes all duplicates.

lenges (Hendrycks et al., 2021; Li et al., 2022), because these are the only applications where a lot of test cases are already available. Therefore, most recent papers that evaluate on real-world programs have to rely on unreliable surface form metrics (Ren et al., 2020; Chen et al., 2021b; Xu et al., 2022). This streetlight effect might incentivize the community to work on problems that are easy to evaluate but not useful in practice. In response to this challenge, our paper manually implements a reliable metric for naturally occurring questions. Future works can consider using models to help humans write useful tests (Tufano et al., 2020), or formally verify the correctness of a predicted solution (Chu et al., 2017).

7. Conclusion

We propose DS-1000, a benchmark for generating code for data analysis. Our benchmark 1) contains realistic questions, 2) implements reliable automatic metrics, and 3) proactively defends against memorization strategies. We hope DS-1000 can track the progress of this research area and facilitate fair comparisons between models, and our methods to construct it can inspire other areas where the task is complicated and the ground truth is challenging to evaluate.

Acknowledgements

We thank Noah A. Smith, Tianbao Xie, Shuyang Jiang for their helpful feedback on this work.

References

Agashe, R., Iyer, S., and Zettlemoyer, L. JuICe: A large scale distantly supervised dataset for open domain context-based code generation. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2019.

Aghajanyan, A., Huang, B., Ross, C., Karpukhin, V., Xu, H., Goyal, N., Okhonko, D., Joshi, M., Ghosh, G., Lewis, M., et al. Cm3: A causal masked multimodal model of the internet. *arXiv preprint arXiv:2201.07520*, 2022.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Bavarian, M., Jun, H., Tezak, N., Schulman, J., McLeavey, C., Tworek, J., and Chen, M. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*, 2022.

Berant, J., Chou, A., Frostig, R., and Liang, P. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pp. 1533–1544, 2013.

Black, S., Biderman, S., Hallahan, E., Anthony, Q., Gao, L., Golding, L., He, H., Leahy, C., McDonell, K., Phang, J., Pieler, M., Prashanth, U. S., Purohit, S., Reynolds, L., Tow, J., Wang, B., and Weinbach, S. GPT-NeoX-20B: An open-source autoregressive language model. In *Proceedings of BigScience Episode #5 – Workshop on Challenges & Perspectives in Creating Large Language Models*, virtual+Dublin, May 2022. Association for Computational Linguistics.

Bolyen, E., Rideout, J. R., Dillon, M. R., Bokulich, N. A., Abnet, C. C., Al-Ghalith, G. A., Alexander, H., Alm, E. J., Arumugam, M., et al. Reproducible, interactive, scalable and extensible microbiome data science using qiime 2 (vol 37, pg 852, 2019). *Nature biotechnology*, 2019.

Carlini, N., Tramèr, F., Wallace, E., Jagielski, M., Herbert-Voss, A., Lee, K., Roberts, A., Brown, T., Song, D., Erlingsson, Ú., Oprea, A., and Raffel, C. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*, pp. 2633–2650. USENIX Association, August 2021a. ISBN 978-1-939133-24-3. URL `https://www.usenix.org/conference/usenixsecurity21/presentation/carlini-extracting`.

Carlini, N., Tramèr, F., Wallace, E., Jagielski, M., Herbert-Voss, A., Lee, K., Roberts, A., Brown, T. B., Song, D., Erlingsson, Ú., Oprea, A., and Raffel, C. Extracting training data from large language models. In Bailey, M. and Greenstadt, R. (eds.), *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pp. 2633–2650. USENIX Association, 2021b. URL `https://www.usenix.org/conference/usenixsecurity21/presentation/carlini-extracting`.

Chandel, S., Clement, C. B., Serrato, G., and Sundaresan, N. Training and evaluating a jupyter notebook data science assistant. *CoRR*, abs/2201.12901, 2022a. URL `https://arxiv.org/abs/2201.12901`.

Chandel, S., Clement, C. B., Serrato, G., and Sundaresan, N. Training and evaluating a jupyter notebook data science assistant. *arXiv preprint arXiv:2201.12901*, 2022b.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021a.

Chen, X., Gong, L., Cheung, A., and Song, D. Plotcoder: Hierarchical decoding for synthesizing visualization code in programmatic context. In *Association for Computational Linguistics (ACL)*, 2021b.

Chu, S., Wang, C., Weitz, K., and Cheung, A. Cosette: An automated prover for sql. In *CIDR*, 2017.

Elangovan, A., He, J., and Verspoor, K. Memorization vs. generalization : Quantifying data leakage in NLP performance evaluation. In Merlo, P., Tiedemann, J., and Tsarfaty, R. (eds.), *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume, EACL 2021, Online, April 19 - 23, 2021*, pp. 1325–1335. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.eacl-main.113. URL `https://doi.org/10.18653/v1/2021.eacl-main.113`.

Faghmous, J. H. and Kumar, V. A big data guide to understanding climate change: The case for theory-guided data science. *Big data*, 2(3):155–163, 2014.

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W., Zettlemoyer, L., and Lewis, M. Incoder: A generative model for code infilling and synthesis. *CoRR*, abs/2204.05999, 2022.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and Steinhardt, J. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

Li, Y., Choi, D. H., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D. J., Robson, E. S., Kohli, P., de Freitas, N., Kavukcuoglu, K., and Vinyals, O. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814, 2022. doi: 10.48550/arXiv.2203.07814. URL https://doi.org/10.48550/arXiv.2203.07814.

Liang, P., Jordan, M. I., and Klein, D. Learning Dependency-Based Compositional Semantics. *Computational Linguistics*, 39(2):389–446, 06 2013. ISSN 0891-2017. doi: 10.1162/COLI_a_00127. URL https://doi.org/10.1162/COLI_a_00127.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. A conversational paradigm for program synthesis. *CoRR*, abs/2203.13474, 2022.

Poesia, G., Polozov, A., Le, V., Tiwari, A., Soares, G., Meek, C., and Gulwani, S. Synchromesh: Reliable code generation from pre-trained language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=KmtVD97J43e.

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., and Ma, S. Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297, 2020. URL https://arxiv.org/abs/2009.10297.

Romero, C. and Ventura, S. Data mining in education. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 3(1):12–27, 2013.

Scholak, T., Schucher, N., and Bahdanau, D. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 9895–9901, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

Shi, F., Fried, D., Ghazvininejad, M., Zettlemoyer, L., and Wang, S. I. Natural language to code translation with execution. *arXiv preprint arXiv:2204.11454*, 2022.

Tay, Y., Dehghani, M., Tran, V. Q., Garcia, X., Bahri, D., Schuster, T., Zheng, H. S., Houlsby, N., and Metzler, D. Unifying language learning paradigms. *arXiv preprint arXiv:2205.05131*, 2022.

Tufano, M., Drain, D., Svyatkovskiy, A., Deng, S. K., and Sundaresan, N. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617*, 2020.

Xu, F. F., Alon, U., Neubig, G., and Hellendoorn, V. J. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pp. 1–10, 2022.

Yin, P., Deng, B., Chen, E., Vasilescu, B., and Neubig, G. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pp. 476–486. ACM, 2018. doi: https://doi.org/10.1145/3196398.3196408.

Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., Zhang, Z., and Radev, D. R. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

Zelle, M. and Mooney, R. J. Learning to parse database queries using inductive logic programming. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pp. 1050–1055, 1996.

Zettlemoyer, L. and Collins, M. Online learning of relaxed ccg grammars for parsing to logical form. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pp. 678–687, 2007.

Zhao, Z., Wallace, E., Feng, S., Klein, D., and Singh, S. Calibrate before use: Improving few-shot performance of language models. In *International Conference on Machine Learning*, pp. 12697–12706. PMLR, 2021.

Zhong, R., Yu, T., and Klein, D. Semantic evaluation for text-to-SQL with distilled test suites. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 396–411, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.29. URL https://aclanthology.org/2020.emnlp-main.29.

Appendices

A. Details on Data collection

A.1. Problem Selection

Sourcing Popular StackOverflow Problems. We leverage StackOverflow to collect representative data science code generation problems on each package. To select popular problems, we first removed duplicates and selected problems with at least 1 vote, 1000 views, and accepted answers. After this initial filtering, we obtain 15881 NumPy problems, 26248 Pandas problems, 1965 PyTorch problems, 8258 TensorFlow problems, 4141 SciPy problems, and 4499 Scikit-learn problems. Next, we perform a stratified sampling on problems from each year to further subsample the problems from Pandas and TensorFlow. We design a threshold for each year's problems differently because older problems naturally have higher votes. Table 8 displays the criteria we use to filter each year's problem on Pandas and TensorFlow.

Filtering Suitable Problems From the initial pool of popular problems, our annotators select problems that are suitable for building DS-1000. Besides the considerations mentioned in Section 2, we discuss those problems that are not selected here. In general, we consider a problem to be unsuitable if our multi-criteria evaluation is not applicable (untestable problems). For example, we leave StackOverflow problems involving hardware problems (See Figure 29), software errors (See Figure 30), concrete execution time analysis, etc. out of DS-1000. See Figure 31 for a concrete example where the problem asks for a natural language explanation of a method in TensorFlow. We leave incorporating more unsuitable StackOverflow problems for future work.

Controlling Library Version. Table 7 details the software versions that we build DS-1000 with.

Package	Version
seaborn	0.11.2
matplotlib	3.5.2
numpy	1.21.6
pandas	1.3.5
scikit-learn	1.0.2
scipy	1.7.3
tensorflow	2.10.0
torch	1.12.1

Table 7: The versions of software in DS-1000

A.2. Example Problems

Here we present an example problem from each of the seven libraries in DS-1000 to illustrate the challenges we encountered in creating DS-1000.

Figure 9 shows a NumPy problem asking how to generate samples that suit log-uniform distribution. Since the result varied with different solutions and different settings, it's unreasonable to test the equivalence. Instead, we apply the Kolmogorov-Smirnov test that judges whether two groups of samples suit the identical or rather similar population.

Figure 10 gives a SciPy problem that describes some trouble with the number of stored elements in a sparse matrix and asks for a solution without repetitive type conversion. Since our self-made assertion that checks the equivalence of two matrices cannot distinguish the difference between stored numbers, we need a special design for this problem. For functional correctness, we check the type of `b`, match the elements, and check the number of non-zero elements(`nnz`), which is the core of the problem. For surface form constraint, we reject the use of `.toarray()`, `.A`, `.todense()`, and `.array()`, which might attempt to transform a sparse matrix into a dense one.

Figure 11 shows a problem of Pandas. We found that the solution with the highest votes ignores the requirement "but does not exactly match it" in the description of the problem, and thus we had to fix the bug in our reference solution. Besides, we enhanced the test case to check the point.

The example shown in Figure 12 is a problem of TensorFlow. Since there is no built-in testing function defined in TensorFlow 2.10.0, we had to write it ourselves. Additionally, we require generated program to use `tf.reduce_sum` in accordance with "the equivalent of ... in Tensorflow" in the description.

Figure 13 demonstrates a PyTorch problem. Here we use `load_data()` to hide the input and let the model learn from the description. The correct solution is not a regular type conversion, as indicated in the error message.

Figure 14 shows a Scikit-learn problem. It requires applying the preprocessing method defined in Scikit-learn to a Pandas data frame, and it tests whether models learn Scikit-learn, Pandas, and their interaction well. Actually, these data science packages are not independent of others, and this problem exemplifies the interactions.

Figure 15 shows a Plotting problem. Here the original question on StackOverflow contains an example figure, which cannot be processed by current code models. We rewrite the original question into a standalone question, that is, "Plot y over x and show blue dashed grid lines". The automatic evaluation comes in two parts. First, it compares the image produced by the generated program with the image produced by the reference program. If two images match exactly, then the generated program is considered correct. Otherwise, the automatic evaluation examines the Plotting axis object and asserts the conditions relevant to the question specification. In this example, the assertions are testing the existence of grid lines and the color of the grid lines.

A.3. Problem Perturbation.

Here, we give an example for each type of perturbation, as shown in Table 1. We highlight the changes we made through perturbation.

Figure 16, Figure 17 and Figure 18 give examples of surface

	Year	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021	2022
Pandas	vote	50	50	14	14	14	4	4	4	2	2	1	1
	view	5k	5k	5k	5k	5k	1k	1k	1k	1.1k	1.1k	1k	1k
	problems	2	8	467	494	554	2139	2483	1894	1985	809	225	8
TensorFlow	vote	-	-	-	-	10	5	4	2	2	1	1	1
	view	-	-	-	-	3k	2k	1k	1.6k	1.2k	1.3k	1k	1k
	problems	-	-	-	-	100	632	1136	1167	1004	776	185	6

Table 8: The problem selection parameters and the number of result problems of Pandas and TensorFlow.

perturbation, showing code context perturbation, paraphrasing, and changes in example respectively. The original task hasn't changed.

Figure 19 shows how we replace keywords with analogy words in a Pandas problem. The perturbed problem asks for applying an exponential function to column `A` and `B`. The problem in Figure 20 concentrates on changing the required index. Here we specify the target index on which to operate using ordinal numbers. Figure 21 gives an example of reversing the order. The desired output string is reversed(from "abc,def,ghi,jkl" to "jkl,ghi,def,abc"). We expect models to capture the information and handle the perturbation. Figure 22 shows an example of changing the type of the required result. Here we change the type from `pd.DataFrame` to `pd.Series`.

Figure 23 and Figure 24 demonstrate how we difficult rewrite problems. The example in Figure 23 replaces "highest" with "lowest" and changes the shape of the desired output (from n $\times$ 1 to 1 $\times$ n). Example in Figure 24, on the other hand, focuses on digging more perturbations that increase the difficulty. The model should not only learn how to use a two-sample KS test but also learn how to interpret the result of the KS test.

A.4. Prompt Format

Like what we've mentioned in Section 4.1, we also provide a prompt of Completion format. Here are two examples (Figure 25 and Figure 26) showing that we have to translate the code in the right context into natural language instruction as complementary information.

B. Details of experiments on `numpy-100`

`numpy-100` is a collection of 100 NumPy exercises from NumPy mailing list, StackOverflow, and NumPy documentation, which has been forked over 4.7k times on the GitHub.

As shown in Figure 4, in the `numpy-100`, each problem is given a short, one-sentence description, with no code context, followed by a reference answer.

````
#### 28. Consider a (6,7,8) shape array, what is the index (x,y,z)
of the 100th element?

```python
print(np.unravel_index(99, (6,7,8)))
```
````

Figure 4: A `numpy-100` example.

Firstly, we simply wrote a code context for each problem and applied Insertion prompt, as shown in Figure 5.

```
Problem:
Consider a (6,7,8) shape array, what is the index (x,y,z) of the 100th element?

<code>
import numpy as np
[insert]
print(result)
</code>
```

Figure 5: A `numpy-100` example prompt.

Then we paraphrased the problem and modified the code context in the problem for surface perturbations, as shown in Figure 6 and Figure 7. First, we changed the description from "Consider a (6,7,8) shape array, what is the index (x,y,z) of the 100th element?" to "I have an array with shape (6,7,8). I need to find the index of the 100th element.". In a second way, we changed the code context to require models to complete a function.

```
Problem:
I have a array with shape (6,7,8). I need to find the index of the 100th element.

<code>
import numpy as np
[insert]
print(result)
</code>
```

Figure 6: A `numpy-100` example of surface perturbation. We expressed the same description in different words.

For the semantic perturbation, we changed the requirements of the problem and the semantics of the reference solution without changing its difficulty. As shown in the Figure 8, we just changed the "100" to "99".

```
Problem:
Consider a (6,7,8) shape array, what is the index (x,y,z) of the 100th element?

<code>
import numpy as np

def f():
    [insert]
    return result
</code>
```

Figure 7: A `numpy-100` example of surface perturbation. We changed the code context.

```
Problem:
Consider a (6,7,8) shape array, what is the index (x,y,z) of the 99th element?

<code>
import numpy as np
[insert]
print(result)
</code>
```

Figure 8: A `numpy-100` example of semantic perturbation. We just changed the required index.

At last, we equipped each problem and perturbation with a test case and an automatic evaluation program and tested the performance of Codex-002 on them. We sampled 20 problems from `numpy-100` and generated 10 samples for each problem with the temperature set to 0.7, and top-p cutoff set to 0.95.

Problem:
I could not find a built-in function in Python to generate a log uniform distribution given a min and max value (the R equivalent is here), something like: loguni[n, min, max, base] that returns n log uniformly distributed in the range min and max.
The closest I found though was `numpy.random.uniform`.
That is, given range of x, I want to get samples of given size (n) that suit log-uniform distribution.
Any help would be appreciated!

```
A:
<code>
import numpy as np
min = 1
max = np.e
n = 10000
</code>
BEGIN SOLUTION
<code>
[insert]
</code>
END SOLUTION
<code>
print(result)
</code>
```

Reference Solution

```
import spicy.stats
result = scipy.stats.loguniform.rvs(a = min, b = max, size = n)
```

Automatic Evaluation

Test case 1

```
min = 1
max = np.e
n = 10000
ans = ... # generated by Reference solution
```

Test code

```
np.testing.assert_array_equal(result.shape, ans.shape)
from scipy.stats import ks_2samp
# Kolmogorov-Smirnov Test judges whether the two sampled
# from similar distribution
assert ks_2samp(result, ans)[0] <= 0.1
```

Surface form check

`for` and `while` should not appear in Syntax Tree

Figure 9: NumPy example problem involving randomness, requiring the use of a specialist knowledge test.

Problem:
I want to remove diagonal elements from a sparse matrix. Since the matrix is sparse, these elements shouldn't be stored once removed.
Scipy provides a method to set diagonal elements values: `setdiag`
...[omit for brevity]
However with csr_matrix, it seems diagonal elements are not removed from storage:
...[omit for brevity]

```
>>> b.setdiag(0)
>>> b
<2x2 sparse matrix of type '<type 'numpy.float64'>'
    with 4 stored elements in Compressed Sparse Row format>
>>> b.toarray()
array([[ 0.,  1.],
      [ 1.,  0.]])
```

Through a dense array, we have of course:

```
>>> csr_matrix(b.toarray())
<2x2 sparse matrix of type '<type 'numpy.float64'>'
    with 2 stored elements in Compressed Sparse Row format>
```

Is that intended? If so, is it due to the compressed format of csr matrices? Is there any workaround ***else than going from sparse to dense to sparse again****?*

```
A:
<code>
from scipy import sparse
import numpy as np
a = np.ones((2, 2))
b = sparse.csr_matrix(a)
</code>
BEGIN SOLUTION
<code>
[insert]
</code>
END SOLUTION
<code>
print(b)
</code>
```

Reference Solution

```
b.setdiag(0)
b.eliminate_zeros()
```

Automatic Evaluation

Test case 1

```
a = np.ones((2, 2))
```

Test case 2

```
a = []

ans = sparse.csr_matrix(a)
ans.setdiag(0)
ans.eliminate_zeros()
```

Test code

```
assert type(b) == type(ans)
# Matching elements
assert len(sparse.find(b != ans)[0]) == 0
# Checking number of nonzero elements
assert b.nnz == ans.nnz
```

Surface form check

`.toarray()`, `.A`, `.todense()`, `.array()` should not appear in Syntax Tree

Figure 10: An example problem of SciPy. Specific checking on conversion between dense matrix and sparse matrix.

Problem:
I have a dataframe with column names, and I want to find the one that contains a certain string, but does not exactly match it. I'm searching for 'spike' in column names like 'spike-2', 'hey spike', 'spiked-in' (the 'spike' part is always continuous).
I want the column name to be returned as a string or a variable, so I access the column later with `df['name']` *or* `df[name]` *as normal. I want to get a list like* `['spike-2', 'spiked-in']`.
I've tried to find ways to do this, to no avail. Any tips?

```
A:
<code>
import pandas as pd
data = {'spike-2': [1,2,3], 'hey spke': [4,5,6],
      'spiked-in': [7,8,9], 'no': [10,11,12]}
df = pd.DataFrame(data)
s = 'spike'
</code>
BEGIN SOLUTION
<code>
[insert]
</code>
END SOLUTION
<code>
print(result)
</code>
```

Highest-vote Solution

367

Just iterate over `DataFrame.columns` , now this is an example in which you will end up with a list of column names that match:

```
spike_cols = [col for col in df.columns if 'spike' in col]
```

Reference Solution

```
result = [col for col in df.columns
                if s in col and col != s]
```

Automatic Evaluation

Test case 1

```
data = {'spike-2': [1,2,3], 'hey spke': [4,5,6],
      'spiked-in': [7,8,9], 'no': [10,11,12],
      'spike': [13,14,15]}
df = pd.DataFrame(data)
s = 'spike'

ans = [col for col in df.columns
            if s in col and col != s]
```

Test code

```
assert result == ans
```

Figure 11: An example problem of Pandas. We need to write reference solutions by ourselves because high-vote replies from StackOverflow ignore the requirement "but does not exactly match it".

Problem:
I'm using tensorflow 2.10.0.
What is the equivalent of the following in Tensorflow?
`np.sum(A, axis=1)`
I want to get a tensor.

```
A:
<code>
import tensorflow as tf
import numpy as np

np.random.seed(10)
A = tf.constant(np.random.randint(100,size=(5,
3)))
</code>
BEGIN SOLUTION
<code>
[insert]
</code>
END SOLUTION
<code>
### output your answer to the variable 'result'
print(result)
</code>
```

Reference Solution

```
result = tf.reduce_sum(A, 1)
```

Test case 1

```
np.random.seed(10)
A = tf.constant(np.random.randint(100,size=(5, 3)))
ans =... # generated by Reference Solution
```

Test code

```
def tensor_equal(a, b): # self-made test function
    if type(a) != type(b):
        return False
    if isinstance(a, type(tf.constant([]))) is not True:
        if isinstance(a, type(tf.Variable([]))) is not True:
            return False
    if a.shape != b.shape:
        return False
    if a.dtype != tf.float32:
        a = tf.cast(a, tf.float32)
    if b.dtype != tf.float32:
        b = tf.cast(b, tf.float32)
    if not tf.reduce_min(tf.cast(a == b, dtype=tf.int32)):
        return False
    return True
assert tensor_equal(result, ans)
```

Surface form check

`tf.reduce_sum` should be called

Figure 12: An example problem of TensorFlow that calls for specific API usage.

Problem:

I have this code:

```
import torch
list_of_tensors = [ torch.randn(3), torch.randn(3),
torch.randn(3)]
tensor_of_tensors = torch.tensor(list_of_tensors)
```

I am getting the error:

```
ValueError: only one element tensors can be converted
to Python scalars
```

How can I convert the list of tensors to a tensor of tensors in pytorch?

```
A:

<code>
import numpy as np
import pandas as pd
import torch
list_of_tensors = load_data()
</code>
BEGIN SOLUTION
<code>
[insert]
</code>
END SOLUTION
<code>
print(tensor_of_tensors)
</code>
```

Reference Solution

```
tensor_of_tensors = torch.stack((list_of_tensors))
```

Automatic Evaluation

Test case 1

```
torch.random.manual_seed(42)
list_of_tensors = [torch.randn(3), torch.randn(3),
torch.randn(3)]
ans = ... # generated by Reference solution
```

Test code

```
torch.testing.assert_close(tensor_of_tensors, ans,
                           check_dtype = False)
```

Figure 13: An example problem of PyTorch, with failed attempt and error message given in the description.

Problem:

I'm using the excellent `read_csv()` *function from pandas, which gives:*

```
In [31]: data = pandas.read_csv("lala.csv",
delimiter=",")
In [32]: data
Out[32]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 12083 entries, 0 to 12082
Columns: 569 entries, REGIONC to SCALEKER
dtypes: float64(51), int64(518)
```

but when i apply a function from scikit-learn i loose the informations about columns:

```
from sklearn import preprocessing
preprocessing.scale(data)
```

gives numpy array.

Is there a way to apply preprocessing.scale to DataFrames without loosing the information(index, columns)?

```
A:
<code>
import numpy as np
import pandas as pd
from sklearn import preprocessing
data = load_data()
</code>
BEGIN SOLUTION
<code>
[insert]
</code>
END SOLUTION
<code>
print(df_out)
</code>
```

Reference Solution

```
df_out = pd.DataFrame(preprocessing.scale(data),
                 index=data.index, columns=data.columns)
```

Automatic Evaluation

Test case 1

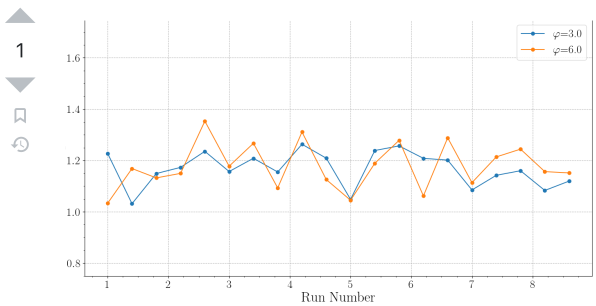
```
np.random.seed(42)
data = pd.DataFrame(np.random.rand(3, 3),
                    index=['first', 'second', 'third'],
                    columns=['c1', 'c2', ‘c3’])

ans = ... # generated by Reference Solution
```

Test code

```
# tolerate rounding error
pd.testing.assert_frame_equal(df_out, ans,
                    check_dtype=False, check_exact=False)
```

Figure 14: An example problem of Scikit-learn, requiring applying sklearn preprocessing method to Pandas data frame.

Consider the figure below:



This image has been set up with the following code.

```
plt.rc('text', usetex=True)
plt.rc('font', family='serif')
fig, ax = plt.subplots()

ax.set_xlabel("Run Number", fontsize=25)
plt.grid(True, linestyle='--')
...
```

Rewrite prompt

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

x = np.arange(10)
y = np.arange(10)

# Plot y over x and show blue dashed grid lines
# SOLUTION START
```

Reference Solution

```
plt.plot(y, x)
plt.grid(color="blue", linestyle="dashed")
```

Automatic Evaluation

Test case 1

```
x,y = ...[shown in prompt]
```

Test code

```
# Precisely matching images with np.array
from PIL import Image
code_img, oracle_img = ... # load images

sample_image_stat = (
    code_img.shape == oracle_img.shape
    and np.allclose(code_img, oracle_img)
)
try:
    assert sample_image_stat
```

```
# IF Failed, matching image components
ax = plt.gca()

assert ax.xaxis._major_tick_kw["gridOn"]
assert "grid_color" in ax.xaxis._major_tick_kw
assert ax.xaxis._major_tick_kw["grid_color"] in
["blue", "b"]
assert "grid_linestyle" in ax.xaxis._major_tick_kw
assert ax.xaxis._major_tick_kw["grid_linestyle"] in
["dashed", "--", "-.", ":"]
```

Figure 15: An example problem of Plotting. Plotting original questions often contain example figures which cannot be processed by current code models. We rewrite original questions into standalone questions in the form of comments.

Problem:
I have this example of matrix by matrix multiplication using numpy arrays:

```
import numpy as np
m = np.array([[1,2,3],[4,5,6],[7,8,9]])
c = np.array([0,1,2])
m * c
array([[ 0,  2,  6],
       [ 0,  5, 12],
       [ 0,  8, 18]])
```

How can i do the same thing if m is scipy sparse CSR matrix? The result should be csr_matrix as well.
This gives dimension mismatch:

```
sp.sparse.csr_matrix(m)*sp.sparse.csr_matrix(c)
```

```
A:
<code>
from scipy import sparse
import numpy as np
sa = sparse.csr_matrix(np.array([[1,2,3],[4,5,6],
[7,8,9]]))
sb = sparse.csr_matrix(np.array([0,1,2]))
</code>
BEGIN SOLUTION
<code>
[insert]
</code>
END SOLUTION
<code>
print(result)
</code>
```

```
A:
<code>
from scipy import sparse
import numpy as np
example_sA = sparse.csr_matrix(np.array([[1,2,3],
[4,5,6],[7,8,9]]))
example_sB = sparse.csr_matrix(np.array([0,1,2]))
def f(sA = example_sA, sB = example_sB):
</code>
BEGIN SOLUTION
<code>
[insert]
</code>
END SOLUTION
<code>
    return result
</code>
```

Figure 16: An example problem of surface perturbation. We expect model complete the function(on the right).

Origin

Problem:
How do I convert data from a Scikit-learn Bunch object (from sklearn.datasets) to a Pandas DataFrame?

```
from sklearn.datasets import load_iris
import pandas as pd
data = load_iris()
print(type(data))
data1 = pd. # Is there a Pandas method to accomplish this?
```

Problem:
Can you give me any suggestion that transforms a sklearn Bunch object (from sklearn.datasets) to a dataframe? I'd like to do it to iris dataset.
Thanks!

```
from sklearn.datasets import load_iris
import pandas as pd
data = load_iris()
print(type(data))
data1 = pd. # May be you can give me a Pandas method?
```

Figure 17: An example problem of surface perturbation. The description in the prompt has been paraphrased.

Origin

Problem:

How to convert a numpy array of dtype=object to torch Tensor?

```
array([
   array([0.5, 1.0, 2.0], dtype=float16),
   array([4.0, 6.0, 8.0], dtype=float16)
], dtype=object)
```

Problem:

How to convert a numpy array of dtype=object to torch Tensor?

```
x = np.array([
    np.array([1.23, 4.56, 9.78, 1.23, 4.56, 9.78], dtype=np.double),
    np.array([4.0, 4.56, 9.78, 1.23, 4.56, 77.77], dtype=np.double),
    np.array([1.23, 4.56, 9.78, 1.23, 4.56, 9.78], dtype=np.double),
    np.array([4.0, 4.56, 9.78, 1.23, 4.56, 77.77], dtype=np.double),
    np.array([1.23, 4.56, 9.78, 1.23, 4.56, 9.78], dtype=np.double),
], dtype=object)
```

Figure 18: An example problem of surface perturbation. The example input in the prompt has been replaced with another one.

Origin

Problem:

Sample dataframe:

```
df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
```

I'd like to add inverses of each existing column to the dataframe and name them based on existing column names with a prefix, e.g. inv_A is an inverse of column A and so on.

The resulting dataframe should look like so:

```
result = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6], "inv_A": [1/1, 1/2, 1/3], "inv_B": [1/4, 1/5, 1/6]})
```

...[omitted for brevity]

Problem:

Sample dataframe:

```
df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
```

I'd like to add ***exponentials*** *of each existing column to the dataframe and name them based on existing column names with a prefix, e.g.* ***exp_A*** *is an* ***exponential*** *of column A and so on.*

The resulting dataframe should look like so:

```
result = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6], "exp_A": [e^1, e^2, e^3], "exp_B": [e^4, e^5, e^6]})
```

Notice that e is the natural constant.

...[omitted for brevity]

Figure 19: An example problem of semantic perturbation. "inverse" has been replaced with an analogy word "exponential".

Origin

Problem:
I have a 2D array `a` to represent a many-many mapping :
0 3 1 3
3 0 0 0
1 0 0 0
3 0 0 0
What is the quickest way to 'zero' out rows and column entries corresponding to a particular index (e.g. zero_rows = 0, zero_cols = 0 corresponds to the 1st row/column) in this array?

Problem:
I have a 2D array `a` to represent a many-many mapping :
0 3 1 3
3 0 0 0
1 0 0 0
3 0 0 0
What is the quickest way to 'zero' out ***the second row and the first column?***

Figure 20: An example problem of semantic perturbation. The required index of rows and columns has been changed.

Origin

Problem:
I have the following dataframe:
text
1 "abc"
2 "def"
3 "ghi"
4 "jkl"
How can I merge these rows into a dataframe with a single row like the following one?
text
1 "abc, def, ghi, jkl"

Problem:
I have the following dataframe:
text
1 "abc"
2 "def"
3 "ghi"
4 "jkl"
How can I merge these rows into a dataframe with a single row like the following one?
text
1 "jkl, ghi, def, abc"

Figure 21: An example problem of semantic perturbation. The order of the desired string has been reversed.

Origin

Problem:

I have a square correlation matrix in pandas, and am trying to divine the most efficient way to return all values where the value (always a float -1 <= x <= 1*) is above* 0.3.

The `pandas.DataFrame.filter` *method asks for a list of columns or a RegEx, but I always want to pass all columns in. Is there a best practice on this?*

desired DataFrame:

```
                Pearson Correlation Coefficient
Col1 Col2
0    3                  0.373153
1    3                  0.419219
     4                  0.356149
3    4                  0.389972
```

Problem:

I have a square correlation matrix in pandas, and am trying to divine the most efficient way to return all values where the value (always a float -1 <= x <= 1*) is above* 0.3.

The `pandas.DataFrame.filter` *method asks for a list of columns or a RegEx, but I always want to pass all columns in. Is there a best practice on this?*

desired Series:

```
0 3   0.373153
1 3   0.419219
  4   0.356149
3 4   0.389972
dtype: float64
```

Figure 22: An example problem of semantic perturbation. The type of the desired result has been changed but the content still keeps the same.

Origin

Problem:

I have a logistic regression model using Pytorch, where my input is high-dimensional and my output must be a scalar – `0,` 1 *or* 2.

I'm using a linear layer combined with a softmax layer to return a n x 3 tensor, where each column represents the probability of the input falling in one of the three classes (`0,` 1 *or* 2*).*

However, I must return a n x 1 tensor, so I need to somehow pick the highest probability for each input and create a tensor indicating which class had the highest probability. How can I achieve this using Pytorch?

To illustrate, my Softmax outputs this:

```
[[0.2, 0.1, 0.7],
 [0.6, 0.2, 0.2],
 [0.1, 0.8, 0.1]]
```

And I must return this:

```
[[2],
 [0],
 [1]]
```

Problem:

...[omit for brevity]

*However, I must return a **1 x n** tensor, and I want to somehow pick the **lowest** probability for each input and create a tensor indicating which class had the **lowest** probability. How can I achieve this using Pytorch?*

To illustrate, my Softmax outputs this:

```
[[0.2, 0.1, 0.7],
 [0.6, 0.3, 0.1],
 [0.15, 0.8, 0.05]]
```

And I must return this:

`[1, 2, 2]`*, which has the type* `torch.LongTensor`

Figure 23: An example problem that is difficult re-written with a combination of surface and semantic perturbations

Origin

Problem:

I can't figure out how to do a Two-sample KS test in Scipy.

...[omit for brevity]

```
test_stat = kstest(x, 'norm')
#>>> test_stat
#(0.021080234718821145, 0.76584491300591395)
```

Which means that at p-value of 0.76 we can not reject the null hypothesis that the two distributions are identical.

However, I want to compare two distributions and see if I can reject the null hypothesis that they are identical.

...[omit for brevity]

I tried the naive:

```
test_stat = kstest(x, z)
```

and got the following error:

```
TypeError: 'numpy.ndarray' object is not callable
```

Is there a way to do a two-sample KS test in Python? If so, how should I do it?

Thank You in Advance

Problem:

...[omit for brevity]

Is there a way to do a two-sample KS test in Python, ***then test whether I can reject the null hypothesis that the two distributions are identical(result=True means able to reject, and the vice versa) based on alpha?*** *If so, how should I do it?*

Thank You in Advance

Figure 24: An example problem that is difficult re-written for more complexity

Problem:

Sample dataframe:

```
df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
```

I'd like to add inverses of each existing column to the dataframe and name them based on existing column names with a prefix, e.g. inv_A is an inverse of column A and so on.

... [omitted for brevity]

Obviously there are redundant methods like doing this in a loop, ***but*** *there should exist much more pythonic ways of doing it ... [omitted for brevity]*

```
A:
<code>
import pandas as pd
df = pd.DataFrame({"A": [1, 2, 3],"B": [4, 5, 6]})
</code>
result = ...# put solution in this variable
BEGIN SOLUTION
<code>
```

Figure 25: Completion prompt corresponding to Figure 1.

Problem:
Say that I want to train `BaggingClassifier` *that uses* `DecisionTreeClassifier`*:*

```
dt = DecisionTreeClassifier(max_depth = 1)
bc = BaggingClassifier(dt, n_estimators = 20, max_samples = 0.5, max_features = 0.5)
bc = bc.fit(X_train, y_train)
```

I would like to use `GridSearchCV` *to find the best parameters for both* `BaggingClassifier` *and* `DecisionTreeClassifier`*(e.g.* `max_depth` *from* `DecisionTreeClassifier` *and* `max_samples` *from* `BaggingClassifier`*), what is the syntax for this? Besides, you can just use the default arguments of* `GridSearchCV`.

```
A:
<code>
import numpy as np
import pandas as pd
from sklearn.ensemble import BaggingClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier
X_train, y_train = load_data()
assert type(X_train) == np.ndarray
assert type(y_train) == np.ndarray
X_test = X_train
param_grid = {
    'base_estimator__max_depth': [1, 2, 3, 4, 5],
    'max_samples': [0.05, 0.1, 0.2, 0.5]
}
dt = DecisionTreeClassifier(max_depth=1)
bc = BaggingClassifier(dt, n_estimators=20,
max_samples=0.5, max_features=0.5)
</code>
BEGIN SOLUTION
<code>
[insert]
</code>
END SOLUTION
<code>
proba = clf.predict_proba(X_test)
print(proba)
</code>
```

```
A:
<code>
import numpy as np
import pandas as pd
from sklearn.ensemble import BaggingClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier

X_train, y_train = load_data()
assert type(X_train) == np.ndarray
assert type(y_train) == np.ndarray
X_test = X_train
param_grid = {
    'base_estimator__max_depth': [1, 2, 3, 4, 5],
    'max_samples': [0.05, 0.1, 0.2, 0.5]
}
dt = DecisionTreeClassifier(max_depth=1)
bc = BaggingClassifier(dt, n_estimators=20,
max_samples=0.5, max_features=0.5)
</code>
solve this question with example variable `clf` and
put result in `proba`
BEGIN SOLUTION
<code>
```

Figure 26: More complex Completion (on the right) prompt that requires additional information for a solution.

Problem:
I am using Pandas to get a dataframe like this:

```
   name  a  b  c
0  Aaron 3  5  7
1  Aaron 3  6  9
2  Aaron 3  6  10
3  Brave 4  6  0
4  Brave 3  6  1
```

I want to replace each name with a unique ID so output looks like:

```
  name  a  b  c
0  1    3  5  7
1  1    3  6  9
2  1    3  6  10
3  2    4  6  0
4  2    3  6  1
```

How can I do that?

Reference Solution

```
# df: pd.DataFrame as input
result = df.replace(df['name'].unique(),
 range(1, len(df['name'].unique()) + 1))
```

Wrong Solution

```
# create a column named "ID"
df['ID'] = df.groupby(['name']).ngroup()

result = df
```

Figure 27: An example wrong solution that misunderstands the requirements and modifies on the wrong column.

Problem:

I'm using tensorflow 2.10.0.

I have a list of bytes and I want to convert it to a list of strings:

```
x=[b'\xd8\xa8\xd9\x85\xd8\xb3\xd8\xa3\xd9\x84\xd
8\xa9',
   b'\xd8\xa5\xd9\x86\xd8\xb4\xd8\xa7\xd8\xa1',
    b'\xd9\x82\xd8\xb6\xd8\xa7\xd8\xa1',
    b'\xd8\xac\xd9\x86\xd8\xa7\xd8\xa6\xd9\x8a',
    b'\xd8\xaf\xd9\x88\xd9\x84\xd9\x8a']
```

How can I get the string result list in Tensorflow?

Reference Solution

```
# x: list of bytes as input
result = [tf.compat.as_str_any(a) for a in x]
```

Wrong Solution

```
# Not using method in Tensorflow
result = [item.decode('utf-8') for item in x]
```

Figure 28: An example wrong solution that uses a common function instead of a function of TensorFlow.

102

I think it's a pretty common message for PyTorch users with low GPU memory:

```
RuntimeError: CUDA out of memory. Tried to allocate 😊 MiB (GPU 😊; 😊 GiB total
capacity; 😊 GiB already allocated; 😊 MiB free; 😊 cached)
```

I tried to process an image by loading each layer to GPU and then loading it back:

```
for m in self.children():
    m.cuda()
    x = m(x)
    m.cpu()
    torch.cuda.empty_cache()
```

But it doesn't seem to be very effective. I'm wondering is there any tips and tricks to train large deep learning models while using little GPU memory.

python deep-learning pytorch object-detection low-memory

Figure 29: An example untestable problem involving hardware problems.

247

I am using python 2.7 in Ubuntu 14.04. I installed scikit-learn, numpy and matplotlib with these commands:

```
sudo apt-get install build-essential python-dev python-numpy \
python-numpy-dev python-scipy libatlas-dev g++ python-matplotlib \
ipython
```

But when I import these packages:

```
from sklearn.cross_validation import train_test_split
```

It returns me this error:

```
ImportError: No module named sklearn.cross_validation
```

What I need to do?

python scikit-learn

Figure 30: An example untestable problem involving software errors.

55

I would like to understand what `tf.global_variables_initializer` does in a bit more detail. A sparse description is given here:

> Returns an Op that initializes global variables.

But that doesn't really help me. I know that the op is necessary to initialize the graph, but what does that actually mean? Is this the step where the graph is complied?

tensorflow deep-learning

Figure 31: An example untestable problem involving explanations.