Appendix

Reproducibility Resources

Our code and benchmark are available in https://anonymous.4open.science/r/repo-g7p1/

Limitation

While DSCodeBench significantly advances the evaluation of LLMs in data science code generation, there are still limitations.

First, DSCodeBench focuses exclusively on Python and ten popular data science libraries. Although these libraries cover a wide range of real-world data science workflows, DSCodeBench does not currently assess LLM capabilities across other programming languages. Extending coverage beyond Python remains an important direction for future work.

Second, although the code problems are drawn from real-world GitHub repositories, the extraction and curation process inherently involve some filtering and simplifications. In particular, to improve the robustness of automatic test case generation, we simplify error-handling logic: error-raising code segments are either removed or replaced with default behaviors such as returning None. While this choice enables more reliable automated evaluation, it slightly reduces the fidelity of error-related coding patterns in the dataset. Furthermore, some highly complex or multi-file codebases are excluded, meaning that DSCodeBench primarily targets single-function or single-file tasks rather than full project-level development workflows.

Third, DSCodeBench evaluates functional correctness based on unit tests, without explicitly assessing other important dimensions of code quality, such as computational efficiency, coding style, readability, or security. Models may generate functionally correct but suboptimal or unsafe code, which is not penalized under the current evaluation framework.

Despite these limitations, we believe DSCodeBench offers a strong and necessary step toward realistic, large-scale, and rigorous evaluation of LLMs for real-world data science programming and provides a valuable foundation for further benchmark development.

Improvement Strategy

To address the current limitations of DSCodeBench and further enhance its utility for the community, we outline several concrete improvement strategies for future iterations of the benchmark.

Expanding Language and Library Coverage. We plan to broaden DSCodeBench beyond Python by incorporating tasks from additional programming languages commonly used in data science and scientific computing, such as R. Moreover, we aim to include tasks that leverage emerging libraries and frameworks to better capture evolving trends in the data science ecosystem.

Restoring Realistic Error Handling and Complex Workflows. To increase the realism of coding scenarios, we will revisit the current simplification of error-handling logic. Future versions of DSCodeBench will retain authentic error-raising behaviors and incorporate exception management patterns, enabling models to demonstrate robustness in handling imperfect inputs and edge cases. Additionally, we intend to expand beyond single-file tasks by introducing multi-function and multi-module coding problems, bridging the gap toward project-level code generation.

Broadening Evaluation Dimensions. Although DSCodeBench currently focuses on functional correctness, future evaluations will also consider additional quality metrics, such as runtime efficiency, code readability, adherence to best practices, and basic security properties. Incorporating these dimensions will enable a more holistic evaluation of LLM-generated code and better reflect real-world coding standards.

Through these improvement strategies, we aim to make DSCodeBench not only more comprehensive and realistic but also more reflective of the diverse competencies required in real-world data science programming.

Broader Impact

DSCodeBench is designed to advance the evaluation of LLMs for real-world data science code generation, and its broader impacts span research, education, and industry.

Promoting Realistic and Rigorous Evaluation. By providing a benchmark rooted in real-world coding scenarios with rigorous testing and diverse problem contexts, DSCodeBench encourages the development of LLMs that move beyond solving synthetic or overly simplified problems. This shift helps align research progress with practical requirements faced by data scientists, researchers, and engineers in professional settings.

Improving LLM Robustness and Practicality. Through its complexity and breadth, DSCodeBench reveals areas where current LLMs struggle, such as handling complex data workflows, generating robust and generalizable solutions, and reasoning over longer contexts. As models improve on DSCodeBench, they are more likely to develop skills needed for reliable deployment in real-world data science and engineering workflows, ultimately leading to safer and more productive AI-assisted coding tools.

Supporting Education and Training. DSCodeBench can also serve as a resource for education and training, helping instructors and students explore the capabilities and limitations of LLMs in data science contexts. Benchmark examples drawn from realistic workflows offer valuable case studies for teaching practical coding skills, evaluating model behavior, and fostering critical discussions about AI in programming.

Potential Risks and Ethical Considerations. Despite its benefits, widespread reliance on benchmarks like DSCodeBench could introduce risks if models are over-optimized for benchmark performance without improving underlying reasoning or robustness. Furthermore, increased automation in code generation, while enhancing productivity, may lead to skill atrophy among developers if used in-

discriminately. To mitigate these risks, we emphasize that DSCodeBench should be used alongside diverse evaluation methods and in settings that promote responsible, human-centered deployment of AI tools.

In summary, we believe DSCodeBench will have a positive impact on the field by raising the standards for evaluating LLMs in real-world programming tasks while simultaneously opening important discussions around responsible and human-centered AI development.

Randomness Control

Random Seed In our evaluation framework, we set the random seed to 42 to maintain the consistency of identical experiments. For some libraries, such as NumPy, Tensorflow, and Pytorch, we can easily set the random seed by using their default API function, like *np.random.seed()*, *tf.random.set_seed()*, and *torch.manual_seed()*. But there are also libraries, such as Sklearn, which do not have specific API functions to control the global random seed. Instead, they control randomness by setting the value of *random_state* in the API functions, such as *RandomForestClassifier(random_state=None)*. For these situations, we explicitly reset the value of *random_state* to 42 by iterating through the AST nodes of the code.

Temperature In our experiment, we set the temperature to 0.2. In addition, we run the experiments on the open-source model with temperature=0.6.

Table 1 demonstrates how temperature impacts various LLMs on DSCodeBench. For DeepSeek models, increasing temperature from 0.2 to 0.6 often results in a slight decrease in pass@3 (e.g., from 0.195 to 0.214 for the DeepSeek-Coder-6.7B-Instruct) and a modest drop or stability in the average number of correct code generated, suggesting a controlled diversity gain. In contrast, Qwen2.5 models show a more volatile response. For example, the Qwen2.5-Coder-7B-Instruct and Qwen2.5-Coder-14B-Instruct models see a significant drop in pass@1 and correctness at temperature 0.6, indicating a degradation in performance due to over-randomization. However, the Qwen2.5-Coder-32B-Instruct model maintains high pass@3 performance at both temperatures, showing robustness. These nuanced shifts demonstrate that DSCodeBench can still follow the scaling law, where larger models generally perform better than smaller ones, even under varying temperature settings, highlighting its robustness and effectiveness in evaluating LLM performance.

Table 2 and Table 3 showcase the per-library performance of various LLMs on DSCodeBench at temperature 0.6, highlighting both the number of fully correct code generations and pass@1 across ten major data science libraries. DeepSeek models, especially the V2-Lite and 33B variants, exhibit strong results across libraries like NumPy, Scikit-learn, TensorFlow, and PyTorch, while Qwen2.5 models show more variability, with the 32B model partially closing the performance gap. The results also reveal library-specific difficulty—LightGBM, Seaborn, and Matplotlib consistently yield lower performance across all models, indicating either higher complexity or lower representation in model training. These findings underscore DSCodeBench's strength as a fine-grained diagnostic benchmark capable of capturing nuanced model behaviors across subdomains. It not only reflects overall model performance but also exposes domain-specific weaknesses, making it a powerful tool for evaluating and advancing LLMs in real-world data science code generation.

Alignment

To ensure the reliability of our benchmark, alignment between the three core components—test cases, ground truth code, and code problem descriptions—is critical.

First, we ensure alignment between test cases and ground truth code by using test case generation scripts tailored to the ground truth implementation. All generated test cases must pass the ground truth code; otherwise, we iteratively prompt LLMs to refine the generation script. We further validate sufficiency by conducting test case coverage analysis, quantifying how thoroughly the test cases can cover the code logic branches (details in Appendix: Test Case Coverage).

Second, to align test cases with code problem descriptions, we manually ensure that the input/output formats in the problem description precisely match the parameter structure used in the test case generation script. Additionally, example input-output pairs are directly generated randomly from the test case generation scripts.

Third, we align the ground truth code and problem description by generating the latter via LLMs and filtering out misaligned cases during the pipeline. In the final manual editing, we define alignment based on whether a human developer could correctly implement the ground truth solution based on the problem description, ensuring the benchmark's clarity and usability. We employ both human experts (two of our authors) and LLMs (GPT-4o-mini and GPT-4o) as judges to assess alignment, achieving 97.4% agreement among all the tasks. For the remaining 2.6%, two authors conducted targeted manual corrections.

Data Leakage Mitigation

Although DSCodeBench is constructed from source code retrieved from GitHub, which inherently introduces potential data leakage risks, we have taken multiple measures to mitigate this issue throughout the construction pipeline. Specifically, we avoid directly reusing the original code snippets by reconstructing the code context and performing careful manual editing to ensure the resulting problems differ substantially from the source.

To assess the severity of any remaining leakage, we conduct a comprehensive similarity analysis between the LLM-generated code and the ground truth code across three dimensions: semantic similarity, syntactic similarity, and structural similarity. Semantically, the pass@k performance of LLMs is significantly lower than the ideal 100% pass@k of ground truth code, indicating a lack of direct memorization. Syntactically, we compute text-level similarity and observe scores consistently below 0.4 across different libraries and models. Structurally, we evaluate AST-based similarity and find values below 0.5, further confirming minimal overlap. Figure 1 and Figure 2 show the similarity distributions between LLM-generated code and ground truth code across different

Table 1: Experiment result on DSCodeBench for different temperatures.

Temperature	Model	pass@1	pass@3	Avg. count_correct	Avg. count_part_correct	Avg. count_wrong
0.2	DeepSeek-Coder-1.3B-Instruct	0.076	0.103	76.3±3.3	27.0±2.2	896.7±5.4
	DeepSeek-Coder-6.7B-Instruct	0.163	0.195	162.7±2.1	47.0±1.6	790.3±2.1
	DeepSeek-Coder-V2-Lite-Instruct(15.7B)	0.205	0.234	205.0±1.4	48.0±0.8	747.0±0.8
	DeepSeek-Coder-33B-Instruct	0.222	0.258	222.3±2.6	51.0±5.4	726.7±4.0
	Qwen2.5-Coder-7B-Instruct	0.116	0.164	116.3±7.1	28.7±0.5	855.0±7.5
	Qwen2.5-Coder-14B-Instruct	0.213	0.251	212.7±5.6	49.0±1.4	738.3±5.3
	Qwen2.5-Coder-32B-Instruct	0.229	0.260	228.7±2.6	45.7±4.2	725.7±2.5
0.6	DeepSeek-Coder-1.3B-Instruct	0.057	0.103	57.3±5.2	24.3±7.7	918.3±11.8
	DeepSeek-Coder-6.7B-Instruct	0.157	0.214	157.3±11.1	49.7±1.2	793.0±9.9
	DeepSeek-Coder-V2-Lite-Instruct(15.7B)	0.196	0.245	196.3±6.8	54.7±4.6	749.0±11.0
	DeepSeek-Coder-33B-Instruct	0.203	0.265	202.7±6.8	46.0±3.7	746.3±5.6
	Qwen2.5-Coder-7B-Instruct	0.028	0.054	28.3±2.5	12.0±1.6	959.7±4.1
	Qwen2.5-Coder-14B-Instruct	0.055	0.081	54.7±2.4	25.3±2.6	920.0±2.8
	Qwen2.5-Coder-32B-Instruct	0.159	0.267	158.7±65.2	41.7±6.6	799.7±71.1

Table 2: Experiment result on DSCodeBench for each library at temperature=0.6 (number of code solutions correctly passing all the test cases, the results are shown in the format of mean±standard deviation).

Model	NumPy	Pandas	Scipy	Scikit-learn	Matplotlib	Seaborn	TensorFlow	PyTorch	Keras	LightGBM
DeepSeek-Coder-1.3B-Instruct	9.3±0.5	2.7±0.5	1.3±0.5	14.7±2.1	2.0±0.8	3.3±0.9	6.7±4.0	12.7±3.3	2.3±1.7	2.3±0.5
DeepSeek-Coder-6.7B-Instruct	25.3±3.1	7.0±0.0	3.7±0.5	34.3±1.2	3.0±0.0	8.0±0.0	32.7±5.2	29.7±4.1	11.0±0.0	2.7±0.5
DeepSeek-Coder-V2-Lite-Instruct(15.7B)	25.7±0.9	13.0±0.8	3.0±0.0	42.3±1.2	2.3±0.5	8.7±0.5	36.3±2.9	42.3±3.3	16.7±1.2	6.0±0.0
DeepSeek-Coder-33B-Instruct	32.3±5.2	10.0±1.6	3.0±0.8	41.7±1.7	2.3±0.5	7.7±0.5	41.0±2.8	37.0±5.0	12.3±0.5	5.3±0.5
Qwen2.5-Coder-7B-Instruct	18.3±3.1	6.0±1.4	1.0±0.8	0.0±0.0	0.0±0.0	0.0±0.0	0.0±0.0	0.0±0.0	0.0±0.0	3.0±0.0
Qwen2.5-Coder-14B-Instruct	30.7±2.1	17.7±1.2	2.3±0.5	0.0±0.0	1.0±0.0	0.0±0.0	0.0±0.0	0.0±0.0	0.0±0.0	3.0±0.0
Qwen2.5-Coder-32B-Instruct	37.0±0.8	16.3±2.6	3.0±0.0	20.0±14.2	1.0±0.0	3.7±2.6	28.3±20.2	29.7±21.0	12.0±8.6	7.7±1.9

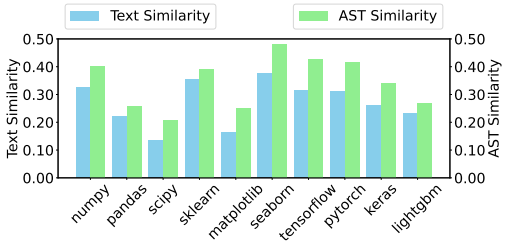

Figure 1: Similarity between LLM-generated solution and ground truth code by different libraries.

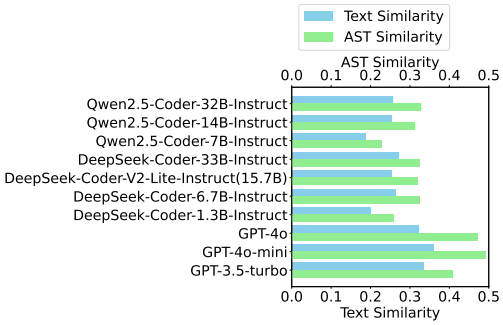

Figure 2: Similarity between LLM-generated solution and ground truth code by different models.

libraries and different models, respectively. These results collectively demonstrate that DSCodeBench has made strong efforts to mitigate data leakage and preserve benchmark integrity.

Test Case Coverage

To demonstrate the quality and completeness of our automatically generated test case scripts, we conducted a comprehensive test case coverage experiment. The purpose of this experiment is to assess how well our test case scripts can cover the logic of the ground truth code, which ultimately serves to test the correctness of LLM-generated code. The experiment was conducted as follows.

For each problem, the ground truth code and the corresponding test case script were saved into two separate files. We developed an auxiliary script that first executes the main function in the test case script to automatically generate 200 test cases. These test cases were then systematically fed into the main function of the ground truth code, simulating the full verification workflow of LLM-generated code. To measure how thoroughly the test cases exercised the code paths of the ground truth implementation, we employed the widely used Python library *coverage*. The coverage tool provides a detailed report of code execution, from which we recorded the final line coverage percentage for each problem.

We aggregated the results and calculated the mean coverage for each library as well as the overall mean across all problems. Figure 3 presents the test case coverage results. The overall mean coverage across all problems reaches

Table 3: Experiment result on DSCodeBench for each library at temperature=0.6 (pass@1).

Model	NumPy	Pandas	Scipy	Scikit-learn	Matplotlib	Seaborn	TensorFlow	PyTorch	Keras	LightGBM
DeepSeek-Coder-1.3B-Instruct	0.071	0.029	0.012	0.136	0.019	0.040	0.061	0.125	0.022	0.043
DeepSeek-Coder-6.7B-Instruct	0.193	0.076	0.033	0.318	0.029	0.096	0.297	0.294	0.106	0.049
DeepSeek-Coder-V2-Lite-Instruct(15.7B)	0.196	0.141	0.027	0.392	0.022	0.104	0.330	0.419	0.160	0.111
DeepSeek-Coder-33B-Instruct	0.247	0.109	0.027	0.386	0.022	0.092	0.373	0.366	0.119	0.099
Qwen2.5-Coder-7B-Instruct	0.140	0.065	0.009	0.000	0.000	0.000	0.000	0.000	0.000	0.056
Qwen2.5-Coder-14B-Instruct	0.234	0.192	0.021	0.000	0.010	0.000	0.000	0.000	0.000	0.056
Qwen2.5-Coder-32B-Instruct	0.282	0.178	0.027	0.185	0.010	0.044	0.258	0.294	0.115	0.142

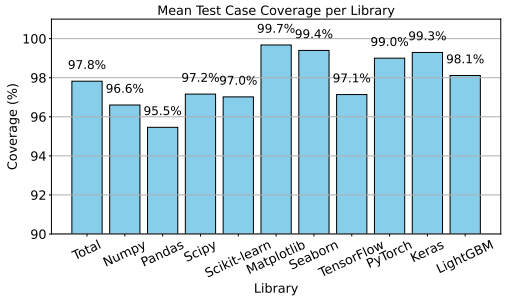

Figure 3: Mean test case coverage on each library.

97.8%, indicating that our test case scripts are highly effective in exercising the ground truth code.

Analyzing individual libraries, we observe that the scientific plotting libraries, Matplotlib (99.7%) and Seaborn (99.4%), achieved the highest coverage. Deep learning frameworks PyTorch (99.3%), TensorFlow (99.0%), and Keras (99.3%) also show excellent coverage, reflecting the strong compatibility between our test case generation script and ground truth code. Meanwhile, data manipulation libraries such as Numpy (96.6%), Pandas (95.5%), and Scipy (97.2%) report slightly lower coverage. Overall, the consistently high coverage across all libraries demonstrates the robustness and reliability of our test case generation pipeline and its suitability for evaluating LLM-generated code in realistic, diverse data science scenarios.

Test Suite Design

In our benchmark, the test suite is systematically divided into two categories: plot-drawing tasks and non-plotting tasks. For plot-drawing libraries, including Matplotlib and Seaborn, we evaluate the similarity between the generated and reference images using the structural_similarity function from skimage.metrics. A test case is considered successful if the mean similarity score across the RGB channels exceeds 0.5; otherwise, it is deemed to have failed. For non-plotting libraries, we first identify the return type of the output. Common return types include standard Python types such as lists, dictionaries, and numeric types (e.g., integers or floats), as well as library-specific types such as tf.Tensor, pd.DataFrame, and torch.nn.Sequential. When the return values are directly comparable, we perform a direct value comparison. In cases where direct comparison is not feasible, we extract and compare relevant attributes of the returned objects, as exemplified by the attribute-wise comparison of torch.nn.Sequential instances.

Comparison between DS-1000 and DSCodeBench

DS-1000 and DSCodeBench differ significantly in terms of task complexity, clarity, and robustness of evaluation. DS-1000 tasks are loosely defined, often derived from informal user queries (e.g., StackOverflow), leading to ambiguous problem statements, inconsistent formats, and limited input diversity.

The following examples compare the code problem description, code solutions, and test case scripts between DS-1000 and DSCodeBench. The DS-1000 code solution is short and focuses on a single function call (*CrossEntropyLoss*), with almost no surrounding logic or structure. In contrast, the DSCodeBench code solution presents a more complex scenario, requiring more steps, including vector normalization, matrix operations (*einsum*), and parameter scaling. This reflects DSCodeBench's emphasis on more realistic and multi-step coding tasks that better mimic real-world development workflows. The test case scripts further highlight the differences. DS-1000 provides only a single hard-coded test case, with fixed random seeds and dimensions, offering minimal variability and limited robustness for model evaluation. By comparison, DSCodeBench introduces a test case generator function that, by default, generates 200 diverse input samples, with test cases spanning a much wider range of values and supporting customizable random seeds. This test case design ensures better coverage of potential input spaces, promoting a more robust model evaluation. The code problem descriptions in DS-1000 often suffer from excessive verbosity, informal tone, and ambiguous requirements, which can lead to inconsistent model behavior. Many prompts are long, poorly formatted, and derived from conversational or irrelevant content in online forums, resulting in vague task objectives and unclear constraints. This reflects the unstructured nature of DS-1000 and poses significant challenges for reliable evaluation. In contrast, DSCodeBench provides clear, well-structured prompts aligned with real-world data science workflows. Each problem includes a precise function signature, a detailed breakdown of the implementation logic, and explicit input-output specifications, enabling deterministic evaluation and improving reproducibility.

Code Problem Description (DS1000)

Problem: I am doing an image segmentation task. There are 7 classes in total so the final outout is a tensor like [batch, 7, height, width] which is a softmax output. Now intuitively I wanted to use CrossEntropy loss but the pytorch implementation doesn't work on channel wise one-hot encoded vector So I was planning to make a function on my own. With a help from some stackoverflow, My code so far looks like this

```
from torch.autograd import Variable
import torch
import torch.nn.functional as F
def cross_entropy2d(input, target,
    weight=None, size_average=True):
    # input: (n, c, w, z), target: (n
        , w, z)
    n, c, w, z = input.size()
    # log_p: (n, c, w, z)
    log_p = F.log_softmax(input, dim
        =1)
    # log_p: (n*w*z, c)
    log_p = log_p.permute(0, 3, 2, 1)
        .contiguous().view(-1, c)  #
        make class dimension last
        dimension
    log_p = log_p[
       target.view(n, w, z, 1).repeat
            (0, 0, 0, c) >= 0]  # this
            looks wrong -> Should
            rather be a one-hot vector
    log_p = log_p.view(-1, c)
    # target: (n*w*z,)
    mask = target >= 0
    target = target[mask]
    loss = F.nll_loss(log_p, target.
        view(-1), weight=weight,
        size_average=False)
    if size_average:
        loss /= mask.data.sum()
    return loss
images = Variable(torch.randn(5, 3,
    4, 4))
labels = Variable(torch.LongTensor(5,
     4, 4).random_(3))
cross_entropy2d(images, labels)
```

I get two errors. One is mentioned on the code itself, where it expects one-hot vector. The 2nd one says the following RuntimeError: invalid argument 2: size '[5 x 4 x 4 x 1]' is invalid. For example purpose I was trying to make it work on a 3 class problem. So the targets and labels are (excluding the batch parameter for simplification !)
(test case is omitted here for brevity)
So how can I fix my code to calculate channel wise CrossEntropy loss ? Or can you give some simple methods to calculate the loss? Thanks Just use the default arguments
A:

```
<code>
import numpy as np
import pandas as pd
from torch.autograd import Variable
import torch
import torch.nn.functional as F
images, labels = load_data()
</code>
loss = ... # put solution in this
    variable
BEGIN SOLUTION
<code>
```

Code Problem Description (DSCodeBench)

You are tasked with implementing a function that computes the contrastive loss between two sets of embeddings. The function should normalize the input embeddings, compute the logits using the dot product of the normalized embeddings, and then apply the cross-entropy loss to obtain the final loss value. The function signature is as follows:

```
def contrastive_loss(q: torch.Tensor,
     k: torch.Tensor, T: float) ->
    torch.Tensor:
```

In this function:
- 'q' is a tensor representing the first set of embeddings. - 'k' is a tensor representing the second set of embeddings. - 'T' is a float constant that acts as a temperature parameter for scaling the logits. The constant used in the main code is '2 * T', which is used to scale the final loss value.

The logic of the main code is as follows:

1. Normalize the input tensors 'q' and 'k' along the specified dimension (dim=1). This ensures that each embedding has a unit norm, which is essential for contrastive learning.
2. Compute the logits by calculating the dot product of the normalized embeddings using the Einstein summation convention ('torch.einsum'). The logits are scaled by dividing by the temperature parameter 'T'.
3. Determine the number of samples 'N' from the shape of the logits tensor.
4. Create a tensor 'labels' that contains the indices of the samples, which will be used as the target labels for the cross-entropy loss. This tensor is created using 'torch.arange(N)'.
5. Compute the cross-entropy loss using 'nn.CrossEntropyLoss()', passing in the logits and the labels. This loss measures how well the model's predictions (logits) match the true labels.
6. Finally, multiply the computed loss by '2 * T' to scale the loss appropriately before returning it.

Input format:
- ‘q‘: A tensor of shape (N, D) where N is the number of samples and D is the dimensionality of the embeddings.
- ‘k‘: A tensor of shape (N, D) where N is the number of samples and D is the dimensionality of the embeddings.
- ‘T‘: A float representing the temperature parameter.

Output format:
- Returns a tensor representing the scaled contrastive loss value.

Input:

```
q = torch.tensor([[ 0.5, -0.2,  0.1],
                  [ 0.3,  0.4, -0.6],
                  [-0.1,  0.8,
                      0.2]])
k = torch.tensor([[ 0.4, -0.1,  0.3],
                  [ 0.2,  0.5, -0.7],
                  [-0.3,  0.6,
                      0.1]])
T = 0.5
```

Output:

```
loss = contrastive_loss(q, k, T)
# loss = tensor(0.2475)
```

Ground Truth Code (DS1000)

```
def generate_ans(data):
    images, labels = data
    loss_func = torch.nn.
        CrossEntropyLoss()
    loss = loss_func(images, labels)
    return loss
```

Ground Truth Code (DSCodeBench)

```
def contrastive_loss(q, k, T):
    q = nn.functional.normalize(q,
        dim=1)
    k = nn.functional.normalize(k,
        dim=1)
    logits = torch.einsum('nc,mc->nm'
        , [q, k]) / T
    N = logits.shape[0]
    labels = torch.arange(N, dtype=
        torch.long)
    return nn.CrossEntropyLoss()(
        logits, labels) * (2 * T)
```

Test Case Script (DS1000)

```
def define_test_input(test_case_id):
    if test_case_id == 1:
        torch.random.manual_seed(42)
        images = torch.randn(5, 3, 4,
            4)
        labels = torch.LongTensor(5,
            4, 4).random_(3)
    return images, labels
```

Test Case Script (DSCodeBench)

```
def test_case_input_generator(n=200):
    test_cases = []
    for _ in range(n):
        batch_size = random.randint
            (2, 128)
        feature_dim = random.randint
            (16, 512)
        T = random.uniform(0.01, 1.0)
        q = torch.randn(batch_size,
            feature_dim)
        k = torch.randn(batch_size,
            feature_dim)
        test_cases.append((q, k, T))
    return test_cases
```

Prompt Format

Several stages of our benchmark construction process involve the use of LLMs. First, we leverage LLMs to generate test case scripts. Second, we use LLMs to generate natural language problem descriptions from ground truth code. Finally, for alignment behavior analysis, we prompt the LLM to generate code solutions based on the given problem descriptions. In our experiments, we also use the prompt that was used in alignment behavior analysis to generate code solutions for benchmarking the state-of-the-art LLMs. The prompts we used in this paper are shown as follows.

Code Problem Description Generation Prompt

Please generate a code problem description of the following code. The description must include the function signature of the main code. Please specify the constant used in the main code. Please explain the logic detail step by step in the main code. Please provide input and output format of the code, but do not provide any input or output examples. Do not provide constrains. Do not provide any utility code. All the code must be able to be generated based on the description standalone.

Code
Here is the code.

Response
The return should follow the following format (replace {} into the code problem):
Code problem description:
{}

Code Generation Prompt

Please generate Python3 solution for the following code problem description:

Code problem description
Here is the code problem description.

Response
The return should follow the following format (replace {} with the solution). Do not generate additional code, such as "__main__" block.
Solution:
{}

Test Case Script Generation Prompt

Please generate a script to automatically generate (n=200) test cases input for the following main code function, make sure the tensor shapes match properly (please return with the following format):

```
def test_case_input_generator(n=200):
    test_cases = []
    for _ in range(n):
        XXX


        test_cases.append(XXX)
    return test_cases
```

Code
Here is the code.

Test Case Script Repair Prompt

Please re-generate a python function named 'test_case_input_generator' to generate high-quality test cases input for the following ground truth solution. The previous version of the script has an error. The script should take n (the total number of test cases input, set default as 200) as input. Do not use any example test case in the code.
Ground Truth Solution
Here is the ground truth code.

Previous Test Case Generation Script
Here is the test case generation script.

Error Info
Here is the error information.

LLM judge Prompt

Please check whether the 'Code Problem' is sufficient enough and proper to test a data science developer's capability in finishing the task. (i.e., whether a human could generate the 'Solution' based on the 'code problem')

Code Problem
Here is the code problem description.

Solution
Here is the code.

Return Format
Please only return True or False.
If the answer is False, please state the reason on the second line, starting with 'REASON:'
If the answer is False, please also suggest the new code problem based on the 'Solution'