DataDreamer: A Tool for Synthetic Data Generation and Reproducible LLM Workflows

Anonymous ACL submission

Abstract

Large language models (LLMs) have become a dominant and important tool for NLP researchers in a wide range of tasks. Today, many researchers use LLMs in synthetic data generation, task evaluation, fine-tuning, distillation, and other model-in-the-loop research workflows. However, challenges arise when using these models that stem from their scale, their closed source nature, and the lack of standardized tooling for these new and emerging workflows. The rapid rise to prominence of these models and these unique challenges has had immediate adverse impacts on open science and on the reproducibility of work that uses them. In this ACL 2024 theme track paper, we introduce DataDreamer, an open source Python library that allows researchers to write simple code to implement powerful LLM workflows. DataDreamer also helps researchers adhere to best practices that we propose to encourage open science and reproducibility.

1 Introduction

While large language models (LLMs) have established a new era in NLP research through the prompt-and-predict paradigm that has proven effective on a wide variety of tasks, the use of these models has come with significant drawbacks (Liu et al., 2023). Many popular models like GPT-4 (OpenAI et al., 2023) are closed source and behind a remote API, while running models locally can be technically complex and expensive due to their scale. Moreover, the now well-established prompting paradigm can be brittle with results widely varying between different models, configurations, and environments (Sclar et al., 2023; Jaiswal et al., 2023). These challenges have made it difficult for researchers to share, reproduce, extend, and compare work, hindering the rate of research progress.

In context of the rapid shift to using these large models in research, this year's 2024 ACL theme track calls for "stimulating discussion about open science and reproducible NLP research, as well as supporting the open source software movement" and invites "high-quality open source software implementations".[4] In concordance with this theme, we introduce DataDreamer, our open source Python package that provides both practical utility to researchers and scientific utility to the community:

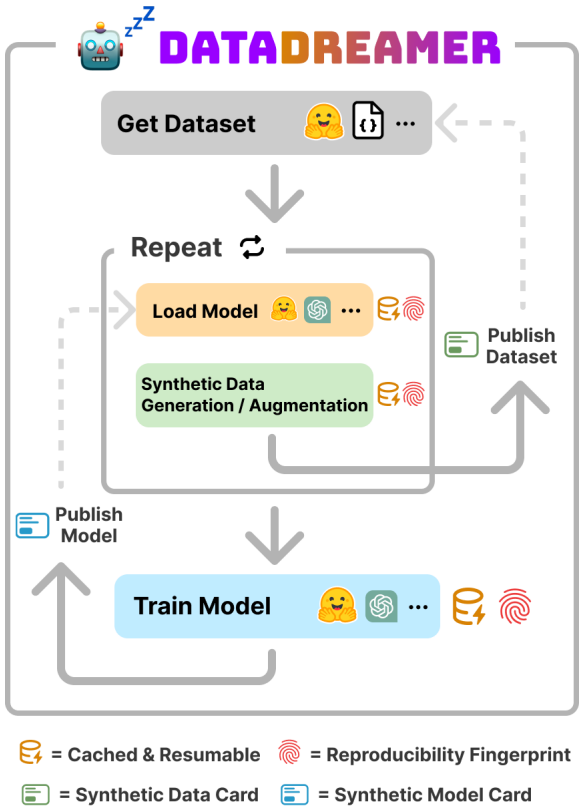

Figure 1: DataDreamer helps researchers implement many types of LLM workflows easier and makes reproducibility automatic and simple. These workflows often involve synthetic data generation with a LLM-in-the-loop and/or fine-tuning, aligning, and distilling models.

[4] https://2024.aclweb.org/calls/main_conference_papers/#theme-track-open-science-open-data-and-open-models-for-reproducible-nlp-research

- DataDreamer helps researchers implement state-of-the-art emerging workflows involving LLMs such as synthetic data generation, fine-tuning, instruction-tuning, and alignment. It simplifies implementations by providing a single library with a standardized interface for many of these tasks while reducing technical complexity around switching between models, caching, resumability, logging, multi-GPU inference and training, using adapter and quantization optimizations, and publishing open datasets and models.
- DataDreamer makes chaining data between tasks, an increasingly common practice, simple. For example, a user can generate data with a synthetic data workflow and then fine-tune on that synthetic data.
- DataDreamer helps researchers implement workflows while crucially producing output that is compatible with open science and reproducible ideals with minimal effort, through automatic caching, reproducibility fingerprints, and more best-practice artifacts.

2 LLM Workflows

To motivate DataDreamer, we first discuss the LLM workflows that it supports. We discuss challenges to open science that arise from these usage patterns. In this paper, we do not seek to validate or critique these approaches. Instead, we offer a solution to implement them and make them reproducible. These LLM workflows are often used in combination with each other (Yuan et al., 2024), and orchestration of multi-stage workflows is frequently implemented through multiple shell or Python scripts. Reproducing these multi-stage workflows is challenging as shell scripts may rely upon a particular author's job scheduler or environment and require execution in a specific order. In Section 4 and 5, we discuss how DataDreamer's task orchestration, caching system, and simple multi-GPU training make it easier to implement these multi-stage workflows in a single Python program, minimizing these issues.

Synthetic Data Generation Recent work has explored using LLMs to create synthetic data for tasks or to augment existing datasets to boost task performance (Yu et al., 2023; Kumar et al., 2020a,b;

Feature	LangChain[1]	Axlotl[2]	HF Transformers + TRL[3]	DataDreamer
Implementation				
Accessible via Python API	✓	✗	✓	✓
Built for Researchers	✗	✗	✓	✓
Integrations				
Open Source Models	✓	✓	✓	✓
Commercial & API-based Models	✓	✗	✗	✓
Tasks				
Prompting & Prompt "Chaining"	✓	✗	✗	✓
Synthetic Data Generation & Augmentation	✓	✗	✗	✓
Fine-tuning LLMs	✗	✓	✓	✓
Instruction-tuning LLMs	✗	✓	✓	✓
Aligning LLMs	✗	✓	✓	✓
Training Classifier Models	✗	✗	✓	✓
Training Embedding Models	✗	✗	✗	✓
Conveniences				
Caching	○	✗	✗	✓
Resumability	✗	✓	○	✓
Simplifies Boilerplate Code (tokenization, etc.)	✓	✓	✗	✓
Simplifies Multi-GPU Inference and Training	✗	○	✗	✓
Publishing Datasets & Models	✗	○	✓	✓
Open Science and Reproducibility				
Reproducibility Fingerprints	✗	✗	✗	✓
Saves Intermediate Outputs	✗	✗	✗	✓
Synthetic Data and Model Cards	✗	✗	✗	✓

Table 1: We compare feature coverage between other popular libraries and solutions available to researchers today that target similar workflows. DataDreamer integrates these features into a single library with a standardized interface making experimentation and chaining data between tasks simple. (✗= No; ✓= Yes; ○ = Partial Support)

Yoo et al., 2021; Han et al., 2021a; Ye et al., 2022; Honovich et al., 2022, *inter alia*). Synthetic data generation involves using a LLM once or multiple times in a multi-stage workflow to process data, sometimes referred to as "chaining" (Rush, 2023). When prompting LLMs to generate or augment datasets, a reproducibility challenge that arises is "prompt sensitivity" where even small variations in a prompt can lead to significantly different results (Sclar et al., 2023). Moreover, it is imperative to tag synthetically generated datasets because of model degradation concerns (Shumailov et al., 2023).

LLMs for Task Evaluation Another increasingly common workflow is using LLMs as judges or as automatic metrics for evaluating a model's performance on a task (Zheng et al., 2023; Fu et al., 2023; Dubois et al., 2023; Chiang and Lee, 2023, *inter alia*). Many of the reproducibility challenges applicable to synthetic data also arise here.

Fine-tuning and Alignment Another common workflow is the creation of task-specific expert models using knowledge from larger models to create smaller, more efficient models via fine-tuning and distillation (Han et al., 2021b; Liu et al., 2022; Hsieh et al., 2023). Instruction-tuning is fine-tuning that allows base pre-trained models to better follow natural language human instruction and improve their generalized task performance (Ouyang et al., 2022; Wei et al., 2021; Sanh et al., 2021; Mishra et al., 2021). Closely related, alignment techniques steer model responses towards those more preferable to humans (Stiennon et al., 2020; Bai et al., 2022; Rafailov et al., 2023). Implementing resumability and efficient training techniques are practical challenges often faced. Reproducibility challenges include sharing exact data and hyperparameters.

Self-improving LLMs Self-improving LLMs through self-feedback training loops is an increasingly active area of research interest (Huang et al., 2022; Wang et al., 2022; Li et al., 2023; Chen et al., 2024; Yuan et al., 2024; Gunasekar et al., 2023). These workflows can be uniquely complex to both implement and reproduce due to requiring multiple rounds that chain together synthetic data generation, automatic evaluation, and model re-training. DataDreamer supports all of these workflows and makes it simple to chain data between them.

[1] https://github.com/langchain-ai/langchain

[2] https://github.com/OpenAccess-AI-Collective/axolotl

3 Demonstration and Examples

Before delving into the structure and implementation of DataDreamer, we first provide a simple demonstration of DataDreamer's capabilities and API through an example synthetic data generation and distillation workflow in Example 1. The LLM used in this example is GPT-4 (OpenAI et al., 2023). As an initial step, the example uses the LLM to generate 1,000 NLP research paper abstracts. The LLM is then used to summarize those abstracts in a tweet-like style. These two steps result in a fully synthetic dataset of abstracts and tweets summarizing them. Using a trainer, this synthetic dataset is then distilled to a small, local model that is capable of summarizing paper abstracts in a tweet-like style. As a final step, the example demonstrates how both the synthetic dataset and the trained model can be published and shared. For illustrative purposes, we demonstrate a sample generation of the trained model's output on this paper's abstract:

> "Introducing DataDreamer, an open source Python library for advanced #NLP workflows. It offers easy code to create powerful LLM workflows, addressing challenges in scale, closed source nature, and tooling. A step towards open science and reproducibility! #AI #MachineLearning"

Further example workflows can be found in the Appendix (Example 2, Example 3, Example 4, Example 5).

4 DataDreamer

DataDreamer is an open source Python package that allows researchers to implement all of the LLM workflows discussed in Section 2 using a single library. DataDreamer provides a standardized interface for prompting and training models, abstracting away vendor-specific libraries and tooling. This makes research code simpler to implement, modify, experiment with, and share with others. DataDreamer integrates with other open source LLM libraries like `transformers` (Wolf et al., 2019) and `trl` (von Werra et al., 2020), as well as commercial model APIs like OpenAI and Anthropic[5] for commercial LLMs (Brown et al., 2020). Moreover, DataDreamer automatically implements the best practices for reproducibility discussed in Section 5.

[3] Wolf et al. (2019); von Werra et al. (2020)

[5] https://www.anthropic.com/

```python
from datadreamer import DataDreamer
from datadreamer.llms import OpenAI
from datadreamer.steps import DataFromPrompt, ProcessWithPrompt
from datadreamer.trainers import TrainHFFineTune
from peft import LoraConfig

with DataDreamer("./output"):
    # Load GPT-4
    gpt_4 = OpenAI(model_name="gpt-4")

    # Generate synthetic arXiv-style research paper abstracts with GPT-4
    arxiv_dataset = DataFromPrompt(
        "Generate Research Paper Abstracts",
        args={
            "llm": gpt_4,
            "n": 1000,
            "temperature": 1.2,
            "instruction": (
              "Generate an arXiv abstract of an NLP research paper."
              " Return just the abstract, no titles."
            ),
        },
        outputs={"generations": "abstracts"},
    )

    # Use GPT-4 to convert the abstracts to tweets
    abstracts_and_tweets = ProcessWithPrompt(
        "Generate Tweets from Abstracts",
        inputs={"inputs": arxiv_dataset.output["abstracts"]},
        args={
          "llm": gpt_4,
          "instruction": "Given the abstract, write a tweet to summarize the work.",
          "top_p": 1.0,
        },
        outputs={"inputs": "abstracts", "generations": "tweets"},
    )

    # Create training data splits
    splits = abstracts_and_tweets.splits(train_size=0.90, validation_size=0.10)

    # Train a model to convert research paper abstracts to tweets with the
    # synthetic dataset
    trainer = TrainHFFineTune(
        "Train an Abstract => Tweet Model",
        model_name="google/t5-v1_1-base",
        peft_config=LoraConfig(),
    )
    trainer.train(
        train_input=splits["train"].output["abstracts"],
        train_output=splits["train"].output["tweets"],
        validation_input=splits["validation"].output["abstracts"],
        validation_output=splits["validation"].output["tweets"],
        epochs=30,
        batch_size=8,
    )

    # Publish and share the synthetic dataset
    abstracts_and_tweets.publish_to_hf_hub("repo_id")

    # Publish and share the trained model
    trainer.publish_to_hf_hub("repo_id")
```

Example 1: In this demonstration snippet, DataDreamer generates a fully synthetic dataset of tweets summarizing research paper abstracts and then trains a smaller T5 distilled model (Raffel et al., 2020) to perform the task and publishes both the synthetic dataset and the trained model. DataDreamer makes it simple to chain data from each step in the workflow to the next and automatically caches each step of this workflow to the `./output/` folder to allow interruption and resumability at any point in the script. The standardized API also makes it easy to switch to and experiment with different models, both open source and commercial, for generation and training.

4.1 Installation

DataDreamer can be installed with the wheel file:

```
pip install <WHEEL_FILE>
```

4.2 Sessions

All code using the DataDreamer library is placed within a "session" using a Python context manager instantiated using the `with` keyword:

```
from datadreamer import DataDreamer

with DataDreamer("./output"):
    ...
```

Workflow tasks can be run within the session context manager. These tasks are called "steps" (loading a dataset, prompting a model, etc.) or "trainers". The session allows DataDreamer to automatically organize the resulting datasets, outputs, caches, training checkpoints, and trained models that result from tasks run within the session into the `./output/` folder. Each step in a workflow assigns a custom descriptive name for its subfolder under `./output/`. DataDreamer sessions automatically provide user-friendly logging around workflow tasks run within the session (see Figure 2).

4.3 Steps

Steps are the core operators in a DataDreamer session. A step in DataDreamer transforms from an input dataset to an output dataset (Lhoest et al., 2021). This is useful for tasks like generating synthetic data from LLMs, or data augmentation for existing datasets. The output of one step can be directly used as the input to another step or as the input to a trainer, allowing users to chain together multiple steps/trainers to create complex workflows. DataDreamer comes with a number of built-in steps for common operations in LLM workflows, some examples of which can be seen in Table 2. Useful standard data processing operations such as `.map()`, `.filter()`, and `.shuffle()` can also quickly be applied to the output of a step for custom processing. DataDreamer uses memory-mapping to handle large datasets stored on disk and can be run lazily over iterable, streaming datasets.

4.4 Models

Models can be loaded in a DataDreamer session and then be passed as an argument to steps like `FewShotPrompt` and `ProcessWithPrompt`. DataDreamer creates a standardized interface for accessing open source and commercial LLMs. It includes interfaces for embedding models as well as LLMs. Examples of supported models and model providers can be found in Table 2.

4.5 Trainers

Trainers can train on a dataset produced by a step in a DataDreamer workflow. The dataset may be loaded from an external source or produced as the output of a step in a multi-step workflow. DataDreamer's trainers support a wide variety of techniques and tasks including fine-tuning, instruction-tuning, alignment via RLHF (Ouyang et al., 2022) and DPO (Rafailov et al., 2023), distillation, training classifiers, and training embedding models. Examples of supported techniques are shown in Table 2.

4.6 Caching and Sharing Workflows

Caching has practical utility in LLM workflows as these large models can be both computationally and financially expensive to run. Therefore, eliminating re-computation can save both time and resources. Caching in DataDreamer happens at multiple levels. When a step or trainer is completed, its resulting dataset or trained model is saved to disk and loaded

Type		Examples
Steps	Load a Dataset	`DataSource`, `HFHubDataSource`, `JSONDataSource`, `CSVDataSource`, ...
	Prompting	`Prompt`, `RAGPrompt`, `ProcessWithPrompt`, `FewShotPrompt`, `DataFromPrompt`, `DataFromAttributedPrompt`, `FilterWithPrompt`, `RankWithPrompt`, `JudgeGenerationPairsWithPrompt`, ...
	Other	`Embed`, `Retrieve`, `CosineSimilarity`, ...
Models		`OpenAI`, `OpenAIAssistant`, `HFTransformers`, `CTransformers`, `VLLM`, `Petals`, `HFAPIEndpoint`, `Together`, `MistralAI`, `Anthropic`, `Cohere`, `AI21`, `Bedrock`, `Vertex`, ...
Trainers		`TrainOpenAIFineTune`, `TrainHFClassifier`, `TrainHFFineTune`, `TrainSentenceTransformer`, `TrainHFDPO`, `TrainHFPPO`, ...

Table 2: A few examples of built-in steps, models, and trainers available in DataDreamer.

from disk if the step or trainer is executed again with the same inputs and arguments, instead of being run again. Additionally, DataDreamer caches at the model-level, caching the results of prompts or texts being run against a model to a SQLite database file. During training, DataDreamer similarly automatically saves checkpoints and resumes from them if interrupted and restarted. Caching uses minimal disk space (storing mainly text) and adds minimal overhead in these workloads dominated by heavy model inference computation, but can be granularly disabled if desired.

DataDreamer's cache system allows a researcher to share both their workflow script and their session output folder with others, giving them access to useful caches and saved outputs. These allow others to easily reproduce and extend the entire workflow while also benefiting from avoiding expensive computations when unnecessary. For example, a researcher could extend another researcher's workflow by adding another step at the end. Only the additional added step would need to be computed, while all of the original steps could have their results loaded from disk.

4.7 Resumability

Caching allows resumability during development, so scripts can be interrupted and resumed. This allows graceful handling of crashes, server preemption, and other situations where only a portion of a workflow was previously computed. Furthermore, caching can be useful during experimentation of a workflow. For example, when modifying a single prompt in the middle of a multi-step synthetic data generation workflow, the change may only affect a certain number of inputs to the next step. If so, only that portion of the work will be re-computed.

4.8 Sharing Open Data and Open Models

DataDreamer provides convenient utilities for exporting and publishing datasets and trained models produced by steps or trainers. Resources can be exported to disk or published to the Hugging Face Hub.[6] When resources are published, DataDreamer can automatically upload a demonstration snippet and set up the live demonstration widget on the Hugging Face Hub, which makes shared resources easily usable. Additionally, these resources are automatically given appropriate metadata such as tags clearly indicating when data is synthetically generated and its source LLM. DataDreamer also produces what we call "synthetic data cards" and "synthetic model cards". Synthetic data and model cards are automatically produced by recursively tracing through all steps, models, and trainers that DataDreamer used to produce the dataset or model. Each step, model, and trainer has associated metadata including license information and citation information. DataDreamer collects this information and produces a synthetic data card (or model card) that reports the information along with reproducibility information for each step, model, and trainer in the workflow. The information collected in our cards is defined in Table 3.

These automatically generated synthetic data cards and model cards can aid in preventing contamination of pre-training sources with model-generated synthetic data. As synthetic data generation becomes more prevalent, contamination can

Date & Time
The date and time the step or trainer was run. This is important to document when using API-based LLMs that can be updated over time.
Dataset Name & Card
The name of any datasets used as part of a step or trainer's operation along with their data cards.
Model Name & Card
The name of any models used in a step or trainer's operation along with their model cards.
URL
A URL that can be referenced for more information about the step or trainer.
License
Any known license that may apply as a result of a model or dataset being used in a step or trainer.
Citations
Citations for datasets and models used in a trainer.
Reproducibility Fingerprint
A hash of all inputs, arguments, and configurations that may affect reproducibility for a step or trainer. When steps and trainers are chained in a multi-stage workflow, the reproducibility hash is computed recursively through the chain. These fingerprints can be used to compare if two workflows within DataDreamer are exactly identical.
Other Reproducibility Information
Other miscellaneous reproducibility information such as environment information, system information, and versions of packages and dependencies.

Table 3: Information automatically recorded in a synthetic data card or synthetic model card. An example synthetic data card can found in Appendix E.

[6] https://huggingface.co/

be a concern due to the performance degradation that has been observed when synthetic datasets are shared and trained on, possibly without the knowledge of the model developer (Shumailov et al., 2023). DataDreamer's cards can also help other researchers understand what license restrictions may apply to the synthetically generated data, among other usability concerns. These automatically generated cards are not a replacement for traditional data cards and model cards (Pushkarna et al., 2022; Mitchell et al., 2019) that recommend a wider set of important attributes such as potential dataset biases. Instead, they provide supplemental information that is crucial to the usability and reproducibility of LLM workflows. We encourage researchers to review and add information that cannot be automatically detected to our generated cards.

4.9 Efficiency and Optimizations

LLMs workflows often benefit from or require certain optimizations to be applied in order to load or process the scale of data and models typically used. DataDreamer supports many of the common optimizations that researchers may want to apply.

Parallelization DataDreamer supports running steps in background processes and running steps concurrently to easily implement parallel task orchestration in a workflow.

Quantization and Adapters DataDreamer supports quantization of model weights that can reduce memory usage (Dettmers and Zettlemoyer) as well as parameter-efficient fine-tuning techniques like LoRA adapters (Hu et al., 2021; Mangrulkar et al., 2022). It standardizes using these optimizations across different model architectures and minimizes boilerplate, making it as simple as a single argument to configure training with LoRA in Example 1. DataDreamer attempts to create uniform support for features across all of its supported integrations when possible. So while the underlying `sentence_transformers` and `transformers` libraries do not support training embedding models with LoRA (Reimers and Gurevych, 2019; Wolf et al., 2019), DataDreamer supports this, which extends the benefits of LoRA to these models.

Multi-GPU Usage DataDreamer makes it simple to load models on multiple GPUs and train models on multiple GPUs with PyTorch FSDP (Paszke et al., 2019; Zhao et al., 2023). For example, training a model on multiple GPUs is as simple as passing a list of `torch.devices` to the `device` parameter of a trainer (`device=["cuda:0", "cuda:1"]`). DataDreamer automatically configures FSDP and launches distributed processes within the session so that a command line launcher like `torchrun` never has to be used, simplifying multi-GPU training. The use of `torchrun` can often force complex, multi-stage workflows being split into multiple scripts launched via shell scripts since training portions need to be isolated from data generation or data processing portions. This added complexity in running the workflow end-to-end can make reproducibility challenging. With DataDreamer, workflows do not need to be re-orchestrated around portions needing to be launched via `torchrun`. Since DataDreamer handles this distributed orchestration automatically, users can build multi-stage workflows involving data generation, data processing, and training on multiple GPUs all in a single Python program, obviating the use of orchestration through multiple shell scripts. Example 4 in the Appendix provides an example of such a workflow.

4.10 Configuration and Extensibility

DataDreamer seeks to minimize configuration and boilerplate code that for most research workflows do not need to be customized, for example automatically handling tokenization and applying the correct padding, among other tasks. DataDreamer applies sensible defaults and standard research practices to minimize configuration. Some researchers, however, may need to customize these choices and the option to override and extend is provided and well-documented.

5 Reproducibility

We outline a few best practices, specific to the emerging use of LLMs in research workflows that DataDreamer adopts. We believe instituting these practices can alleviate a number of reproducibility concerns. Of course, when closed-source models are involved, these concerns can never be fully eliminated (see Section 6 for further discussion on limitations). We discuss how DataDreamer makes it easier to implement these practices or automatically implements these practices in this section.

Adaptable to Model Substitution While experimental workflows can often be sensitive to model choice and the transferability of prompts can be unreliable (Liu et al., 2023), for reproducibility purposes and for ease of experimentation, workflow

Figure 2: DataDreamer logs produced by the workflow in Example 1 when resuming from a prior interrupted run.

implementation code should attempt to minimize dependence on a specific model and should allow other researchers to easily substitute one LLM for another. This can also be useful if a model is not accessible to another researcher or if a model has become obsolete. DataDreamer's API and model abstractions make model substitution simple.

Sharing Prompts Exact prompts used should be shared since even minor variations can significantly impact performance (Sclar et al., 2023). DataDreamer makes it easy to share an entire workflow and session output folder. DataDreamer can also help ensure a re-implementation is exactly identical between two experimental setups by comparing the reproducibility fingerprints of individual steps or the entire workflow in aggregate.

Sharing Intermediate Outputs In multi-stage workflows, intermediate outputs should be shared for inspection and analysis by other researchers as well as for extendability purposes. DataDreamer makes this simple by automatically saving the results of each step in a multi-stage workflow in an easily inspectable Hugging Face `datasets` format (Lhoest et al., 2021). When API-based LLMs are used, there is greater risk to reproducibility. DataDreamer allows workflows to be exactly reproduced from caches in the session output folder, even if the remote API is no longer available.

Synthetic Data Cards and Model Cards Synthetic data and model cards can help other researchers understand the source of synthetic data, license restrictions that may apply, citations that may apply, among other attributes. Importantly, these cards and other metadata-like tags can help prevent contamination of pre-training data (Shumailov et al., 2023). Finally, these cards carry reproducibility information, useful for validating two experimental setups as identical.

Sharing Optimization Configurations Optimizations like quantization can have an effect on generations (Jaiswal et al., 2023). DataDreamer's reproducibility fingerprints account for these configurations and with its easily shareable workflows, DataDreamer makes it easy to reproduce an exact workflow, along with configured optimizations.

Environment-Agnostic Code For reproducibility, code should attempt to minimize dependence on local environments, job schedulers, shell scripts, etc. DataDreamer helps make this easier by providing tools for workflow orchestration (steps, parallelization, managed distributed processes for multi-GPU training) that can be all be done within Python. DataDreamer also minimizes dependencies on local file paths, by organizing results and outputs into the session output folder automatically.

6 Conclusion

The current moment in NLP research and recent progress is exciting yet raises important questions for the community. We introduce DataDreamer, an open source Python package for implementing common patterns and workflows involving LLMs. We believe DataDreamer provides both practical and scientific utility to the research community and that its adoption can help advance the rate of research progress in workflows involving LLMs by making implementation easier and making research output reproducible and extendable.

Limitations

In this work, we outline best practices and implement these practices in an open source system called DataDreamer. We believe these contributions can help aid open science in our field, however, we acknowledge that as long as the research community chooses to use closed-source models for experiments, especially those served behind an API on remote servers, challenges to reproducibility are inevitable. With DataDreamer, we provide a way to reproduce and further analyze some of these experiments long after these remote APIs may be changed or unavailable through the session-based caching system as well as provide a way to easily substitute models where needed through abstractions. To the best of our knowledge, there are no significant ethical considerations that arise from this work. We believe the broader impacts of this work to be largely positive, making state-of-the-art LLM workflows both easier and more accessible to implement and reproduce as well as reducing carbon emissions through DataDreamer's caching system that helps researchers avoid expensive recomputation when possible.

References

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. 2022. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*.

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

Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. 2024. Self-play fine-tuning converts weak language models to strong language models. *arXiv preprint arXiv:2401.01335*.

Cheng-Han Chiang and Hung-yi Lee. 2023. Can large language models be an alternative to human evaluations? *arXiv preprint arXiv:2305.01937*.

Tim Dettmers and Luke Zettlemoyer. The case for 4-bit precision: k-bit inference scaling laws, 2022. *URL https://arxiv. org/abs/2212.09720*.

Yann Dubois, Xuechen Li, Rohan Taori, Tianyi Zhang, Ishaan Gulrajani, Jimmy Ba, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. 2023. Alpacafarm: A simulation framework for methods that learn from human feedback. *arXiv preprint arXiv:2305.14387*.

Jinlan Fu, See-Kiong Ng, Zhengbao Jiang, and Pengfei Liu. 2023. Gptscore: Evaluate as you desire. *arXiv preprint arXiv:2302.04166*.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio Cesar Teodoro Mendes, Allison Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero C. Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, S. Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuan-Fang Li. 2023. Textbooks are all you need. *ArXiv*, abs/2306.11644.

Jesse Michael Han, Igor Babuschkin, Harrison Edwards, Arvind Neelakantan, Tao Xu, Stanislas Polu, Alex Ray, Pranav Shyam, Aditya Ramesh, Alec Radford, et al. 2021a. Unsupervised neural machine translation with generative language models only. *arXiv preprint arXiv:2110.05448*.

Jesse Michael Han, Igor Babuschkin, Harrison Edwards, Arvind Neelakantan, Tao Xu, Stanislas Polu, Alex Ray, Pranav Shyam, Aditya Ramesh, Alec Radford, et al. 2021b. Unsupervised neural machine translation with generative language models only. *arXiv preprint arXiv:2110.05448*.

Or Honovich, Thomas Scialom, Omer Levy, and Timo Schick. 2022. Unnatural instructions: Tuning language models with (almost) no human labor. *arXiv preprint arXiv:2212.09689*.

Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. 2023. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. *arXiv preprint arXiv:2305.02301*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. 2022. Large language models can self-improve. *arXiv preprint arXiv:2210.11610*.

Ajay Jaiswal, Zhe Gan, Xianzhi Du, Bowen Zhang, Zhangyang Wang, and Yinfei Yang. 2023. Compressing llms: The truth is rarely pure and never simple. *arXiv preprint arXiv:2310.01382*.

Varun Kumar, Ashutosh Choudhary, and Eunah Cho. 2020a. Data augmentation using pre-trained transformer models. *arXiv preprint arXiv:2003.02245*.

Varun Kumar, Ashutosh Choudhary, and Eunah Cho. 2020b. Data augmentation using pre-trained transformer models. *arXiv preprint arXiv:2003.02245*.

Quentin Lhoest, Albert Villanova del Moral, Yacine Jernite, Abhishek Thakur, Patrick von Platen, Suraj Patil, Julien Chaumond, Mariama Drame, Julien Plu, Lewis Tunstall, et al. 2021. Datasets: A community library for natural language processing. *arXiv preprint arXiv:2109.02846*.

Xiang Lisa Li, Vaishnavi Shrivastava, Siyan Li, Tatsunori Hashimoto, and Percy Liang. 2023. Benchmarking and improving generator-validator consistency of language models. *arXiv preprint arXiv:2310.01846*.

Wing Lian, Bleys Goodson, Eugene Pentland, Austin Cook, Chanvichet Vong, and "Teknium". 2023. Openorca: An open dataset of gpt augmented flan reasoning traces. https://https://huggingface.co/Open-Orca/OpenOrca.

Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. 2022. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems*, 35:1950–1965.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35.

Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. 2022. Peft: State-of-the-art parameter-efficient fine-tuning methods. https://github.com/huggingface/peft.

Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. 2021. Cross-task generalization via natural language crowdsourcing instructions. In *Annual Meeting of the Association for Computational Linguistics*.

Margaret Mitchell, Simone Wu, Andrew Zaldivar, Parker Barnes, Lucy Vasserman, Ben Hutchinson, Elena Spitzer, Inioluwa Deborah Raji, and Timnit Gebru. 2019. Model cards for model reporting. In *Proceedings of the conference on fairness, accountability, and transparency*, pages 220–229.

Subhabrata Mukherjee, Arindam Mitra, Ganesh Jawahar, Sahaj Agarwal, Hamid Palangi, and Ahmed Awadallah. 2023. Orca: Progressive learning from complex explanation traces of gpt-4.

OpenAI, :, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mo Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Pre-

ston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2023. Gpt-4 technical report.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

Mahima Pushkarna, Andrew Zaldivar, and Oddur Kjartansson. 2022. Data cards: Purposeful and transparent dataset documentation for responsible ai. In *Proceedings of the 2022 ACM Conference on Fairness, Accountability, and Transparency*, pages 1776–1826.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D Manning, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint arXiv:2305.18290*.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Alexander M Rush. 2023. Minichain: A small library for coding with large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 311–317.

Victor Sanh, Albert Webson, Colin Raffel, Stephen H. Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal V. Nayak, Debajyoti Datta, Jonathan D. Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng-Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Févry, Jason Alan Fries, Ryan Teehan, Stella Biderman, Leo Gao, Tali Bers, Thomas Wolf, and Alexander M. Rush. 2021. Multitask prompted training enables zero-shot task generalization. *ArXiv*, abs/2110.08207.

Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. 2023. Quantifying language models' sensitivity to spurious features in prompt design or: How i learned to start worrying about prompt formatting. *arXiv preprint arXiv:2310.11324*.

Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Yarin Gal, Nicolas Papernot, and Ross Anderson. 2023. The curse of recursion: Training on generated data makes models forget.

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. 2020. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford alpaca: An instruction-following llama model. `https://github.com/tatsu-lab/stanford_alpaca`.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, and Shengyi Huang. 2020. Trl: Transformer reinforcement learning. `https://github.com/huggingface/trl`.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language model with self generated instructions. *arXiv preprint arXiv:2212.10560*.

Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W Cohen, Ruslan Salakhutdinov, and Christopher D Manning. 2018. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600*.

Jiacheng Ye, Jiahui Gao, Qintong Li, Hang Xu, Jiangtao Feng, Zhiyong Wu, Tao Yu, and Lingpeng Kong. 2022. Zerogen: Efficient zero-shot learning via dataset generation. *arXiv preprint arXiv:2202.07922*.

Kang Min Yoo, Dongju Park, Jaewook Kang, Sang-Woo Lee, and Woomyeong Park. 2021. Gpt3mix: Leveraging large-scale language models for text augmentation. *arXiv preprint arXiv:2104.08826*.

Yue Yu, Yuchen Zhuang, Jieyu Zhang, Yu Meng, Alexander Ratner, Ranjay Krishna, Jiaming Shen, and Chao Zhang. 2023. Large language model as attributed training data generator: A tale of diversity and bias. In *Thirty-Seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. 2024. Self-rewarding language models. *arXiv preprint arXiv:2401.10020*.

Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024. Tinyllama: An open-source small language model.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. 2023. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *arXiv preprint arXiv:2306.05685*.

A Instruction-Tuning a LLM

```python
from datadreamer import DataDreamer
from datadreamer.steps import HFHubDataSource
from datadreamer.trainers import TrainHFFineTune
from peft import LoraConfig

with DataDreamer("./output"):
    # Get the Alpaca instruction-tuning dataset (cleaned version)
    instruction_dataset = HFHubDataSource(
        "Get Instruction-Tuning Dataset", "yahma/alpaca-cleaned", split="train"
    )

    # Keep only 1000 examples as a quick demo
    instruction_dataset = instruction_dataset.take(1000)

    # Some examples taken in an "input", we'll format those into the instruction
    instruction_dataset.map(
        lambda row: {
            "instruction": (
                row["instruction"]
                if len(row["input"]) == 0
                else f"Input: {row['input']}\n\n{row['instruction']}"
            ),
            "output": row["output"],
        },
        lazy=False,
    )

    # Create training data splits
    splits = instruction_dataset.splits(train_size=0.90, validation_size=0.10)

    # Define what the prompt template should be when instruction-tuning
    chat_prompt_template = "### Instruction:\n{{prompt}}\n\n### Response:\n"

    # Instruction-tune the base TinyLlama model to make it follow instructions
    trainer = TrainHFFineTune(
        "Instruction-Tune TinyLlama",
        model_name="TinyLlama/TinyLlama-1.1B-intermediate-step-1431k-3T",
        chat_prompt_template=chat_prompt_template,
        peft_config=LoraConfig(),
        device=["cuda:0", "cuda:1"],
        dtype="bfloat16",
    )
    trainer.train(
        train_input=splits["train"].output["instruction"],
        train_output=splits["train"].output["output"],
        validation_input=splits["validation"].output["instruction"],
        validation_output=splits["validation"].output["output"],
        epochs=3,
        batch_size=1,
        gradient_accumulation_steps=32,
    )
```

Example 2: In this demonstration snippet, we instruction-tune a model (Ouyang et al., 2022; Zhang et al., 2024; Taori et al., 2023). DataDreamer reduces boilerplate around tokenization, caching, training resumability, multi-GPU training, parameter-efficient fine-tuning, and more.

B Aligning a LLM

```python
from datadreamer import DataDreamer
from datadreamer.steps import HFHubDataSource
from datadreamer.trainers import TrainHFDPO
from peft import LoraConfig

with DataDreamer("./output"):
    # Get the DPO dataset
    dpo_dataset = HFHubDataSource(
        "Get DPO Dataset", "Intel/orca_dpo_pairs", split="train"
    )

    # Keep only 1000 examples as a quick demo
    dpo_dataset = dpo_dataset.take(1000)

    # Create training data splits
    splits = dpo_dataset.splits(train_size=0.90, validation_size=0.10)

    # Align the TinyLlama chat model with human preferences
    trainer = TrainHFDPO(
        "Align TinyLlama-Chat",
        model_name="TinyLlama/TinyLlama-1.1B-Chat-v1.0",
        peft_config=LoraConfig(),
        device=["cuda:0", "cuda:1"],
        dtype="bfloat16",
    )
    trainer.train(
        train_prompts=splits["train"].output["question"],
        train_chosen=splits["train"].output["chosen"],
        train_rejected=splits["train"].output["rejected"],
        validation_prompts=splits["validation"].output["question"],
        validation_chosen=splits["validation"].output["chosen"],
        validation_rejected=splits["validation"].output["rejected"],
        epochs=3,
        batch_size=1,
        gradient_accumulation_steps=32,
    )
```

Example 3: In this demonstration snippet, we align a model using DPO (Rafailov et al., 2023; Zhang et al., 2024; Lian et al., 2023; Mukherjee et al., 2023). DataDreamer reduces boilerplate around tokenization, caching, training resumability, multi-GPU training, parameter-efficient fine-tuning, and more.

C Self-Rewarding LLMs

```python
from datadreamer import DataDreamer
from datadreamer.steps import (
    HFHubDataSource,
    Prompt,
    JudgeGenerationPairsWithPrompt,
)
from datadreamer.trainers import TrainHFDPO
from datadreamer.llms import HFTransformers
from peft import LoraConfig

with DataDreamer("./output"):
    # Get a dataset of prompts
    prompts_dataset = HFHubDataSource(
        "Get Prompts Dataset", "Intel/orca_dpo_pairs", split="train"
    ).select_columns(["question"])

    # Keep only 3000 examples as a quick demo
    prompts_dataset = prompts_dataset.take(3000)

    # Define how many rounds of self-reward training
    rounds = 3

    # For each round of self-reward training
    adapter_to_apply = None
    for r in range(rounds):
        # Use a partial set of the prompts for each round
        prompts_for_round = prompts_dataset.shard(
            num_shards=rounds, index=r, name=f"Round #{r+1}: Get Prompts"
        )

        # Load the LLM
        llm = HFTransformers(
            "TinyLlama/TinyLlama-1.1B-Chat-v1.0",
            adapter_name=adapter_to_apply,
            device_map="auto",
            dtype="bfloat16",
        )

        # Sample 2 candidate responses from the LLM
        candidate_responses = []
        for candidate_idx in range(2):
            candidate_responses.append(
                Prompt(
                    f"Round #{r+1}: Sample Candidate Response #{candidate_idx}",
                    inputs={"prompts": prompts_for_round.output["question"]},
                    args={
                        "llm": llm,
                        "batch_size": 2,
                        "top_p": 1.0,
                        "seed": candidate_idx,
                    },
                )
            )

        # Have the LLM judge its own responses
        judgements = JudgeGenerationPairsWithPrompt(
            f"Round #{r+1}: Judge Candidate Responses",
            args={
                "llm": llm,
                "batch_size": 1,
                "max_new_tokens": 5,
            },
            inputs={
                "prompts": prompts_for_round.output["question"],
                "a": candidate_responses[0].output["generations"],
                "b": candidate_responses[1].output["generations"],
            },
        )
```

```
        # Unload the LLM
        llm.unload_model()

        # Process the judgements into a preference dataset
        dpo_dataset = judgements.map(
            lambda row: {
                "question": row["prompts"],
                "chosen": (
                    row["a"]
                    if row["judgements"] == "Response A"
                    else row["b"]
                ),
                "rejected": (
                    row["b"]
                    if row["judgements"] == "Response A"
                    else row["a"]
                ),
            },
            lazy=False,
            name=f"Round #{r+1}: Create Self-Reward Preference Dataset",
        )

        # Create training data splits
        splits = dpo_dataset.splits(train_size=0.90, validation_size=0.10)

        # Align the TinyLlama chat model with its own preferences
        trainer = TrainHFDPO(
            f"Round #{r+1}: Self-Reward Align TinyLlama-Chat",
            model_name="TinyLlama/TinyLlama-1.1B-Chat-v1.0",
            peft_config=LoraConfig(),
            device=["cuda:0", "cuda:1"],
            dtype="bfloat16",
        )
        trainer.train(
            train_prompts=splits["train"].output["question"],
            train_chosen=splits["train"].output["chosen"],
            train_rejected=splits["train"].output["rejected"],
            validation_prompts=splits["validation"].output["question"],
            validation_chosen=splits["validation"].output["chosen"],
            validation_rejected=splits["validation"].output["rejected"],
            epochs=3,
            batch_size=1,
            gradient_accumulation_steps=32,
        )

        # Unload the trained model from memory
        trainer.unload_model()

        # Use the newly trained adapter for the next round of self-reward
        adapter_to_apply = trainer.model_path
```

Example 4: This demonstration snippet implements a simplified version of the self-rewarding LLMs (Yuan et al., 2024) procedure. This workflow involves using an LLM to judge its own generations in order to self-align and self-improve itself over a number of rounds. DataDreamer allows this complex multi-stage workflow to be implemented intuitively, without needing to split generation and training logic into separate files and without needing to involve a launcher like `torchrun` to perform multi-GPU training. DataDreamer also makes this complex multi-round, multi-stage workflow automatically cachable and resumable.

D Augmenting an Existing Dataset

```python
from datadreamer import DataDreamer
from datadreamer.llms import OpenAI
from datadreamer.steps import ProcessWithPrompt, HFHubDataSource

with DataDreamer("./output"):
    # Load GPT-4
    gpt_4 = OpenAI(model_name="gpt-4")

    # Get HotPot QA questions
    hotpot_qa_dataset = HFHubDataSource(
        "Get Hotpot QA Questions",
        "hotpot_qa",
        config_name="distractor",
        split="train",
    ).select_columns(["question"])

    # Keep only 1000 questions as a quick demo
    hotpot_qa_dataset = hotpot_qa_dataset.take(1000)

    # Ask GPT-4 to decompose the question
    questions_and_decompositions = ProcessWithPrompt(
        "Generate Decompositions",
        inputs={"inputs": hotpot_qa_dataset.output["question"]},
        args={
            "llm": gpt_4,
            "instruction": (
                "Given the question which requires multiple steps to solve,"
                " give a numbered list of intermediate questions required to"
                " solve the question. Return only the list, nothing else."
            ),
        },
        outputs={"inputs": "questions", "generations": "decompositions"},
    ).select_columns(["questions", "decompositions"])
```

Example 5: In this demonstration snippet, we augment an existing dataset, HotpotQA (Yang et al., 2018), a multi-hop QA dataset. DataDreamer makes it easy to perform synthetic dataset augmentation with a LLM. In this example, we add intermediate questions required to solve the multi-hop question.

E Example Synthetic Data Card


```
{
    "data_card": {
        "Generate Research Paper Abstracts": {
            "Date & Time": "<DATE_TIME_HERE>",
            "Model Name": [
                "gpt-4"
            ],
            "Model Card": [
                "https://cdn.openai.com/papers/gpt-4-system-card.pdf"
            ],
            "License Information": [
                "https://openai.com/policies"
            ],
            "Citation Information": [
                "@article{OpenAI2023GPT4TR,\n  title={GPT-4 Technical Report},\n  author={OpenAI},\n  journal={ArXiv},\n  year={2023},\n  volume={abs/2303.08774},\n  url={https://api.semanticscholar.org/CorpusID:257532815}\n}",
                "@article{ouyang2022training,\n  title={Training language models to follow instructions with human feedback},\n  author={Ouyang, Long and Wu, Jeffrey and Jiang, Xu and Almeida, Diogo and Wainwright, Carroll and Mishkin, Pamela and Zhang, Chong and Agarwal, Sandhini and Slama, Katarina and Ray, Alex and others},\n  journal={Advances in Neural Information Processing Systems},\n  volume={35},\n  pages={27730--27744},\n  year={2022}\n}"
            ]
        },
        "Generate Tweets from Abstracts": {
            "Date & Time": "<DATE_TIME_HERE>",
            "Model Name": [
                "gpt-4"
            ],
            "Model Card": [
                "https://cdn.openai.com/papers/gpt-4-system-card.pdf"
            ],
            "License Information": [
                "https://openai.com/policies"
            ],
            "Citation Information": [
                "@article{OpenAI2023GPT4TR,\n  title={GPT-4 Technical Report},\n  author={OpenAI},\n  journal={ArXiv},\n  year={2023},\n  volume={abs/2303.08774},\n  url={https://api.semanticscholar.org/CorpusID:257532815}\n}",
                "@article{ouyang2022training,\n  title={Training language models to follow instructions with human feedback},\n  author={Ouyang, Long and Wu, Jeffrey and Jiang, Xu and Almeida, Diogo and Wainwright, Carroll and Mishkin, Pamela and Zhang, Chong and Agarwal, Sandhini and Slama, Katarina and Ray, Alex and others},\n  journal={Advances in Neural Information Processing Systems},\n  volume={35},\n  pages={27730--27744},\n  year={2022}\n}"
            ]
        }
    },
    "__version__": "0.1.0",
    "datetime": "<DATE_TIME_HERE>",
    "type": "ProcessWithPrompt",
    "name": "Generate Tweets from Abstracts",
    "version": 1.0,
    "fingerprint": "28b5e209bdad7d15",
    "pickled": false,
    "req_versions": {
        "dill": "0.3.7",
        "sqlitedict": "2.1.0",
        "torch": "2.1.2",
        "numpy": "1.26.3",
        "transformers": "4.36.2",
        "datasets": "2.16.1",
        "huggingface_hub": "0.20.2",
```

```
        "accelerate": "0.26.1",
        "peft": "0.7.1",
        "tiktoken": "0.5.2",
        "tokenizers": "0.15.0",
        "petals": "2.2.0",
        "openai": "1.9.0",
        "ctransformers": "0.2.27",
        "optimum": "1.16.2",
        "bitsandbytes": "0.42.0",
        "litellm": "1.15.3",
        "trl": "0.7.6",
        "setfit": "1.0.3",
        "together": "0.2.10",
        "google.generativeai": "0.2.1",
        "google-cloud-aiplatform": "1.35.0"
    },
    "interpreter": "3.11.7 (main, Dec  4 2023, 18:10:11) [Clang 15.0.0
    ↪ (clang-1500.1.0.2.5)]"
}
```


Example 6: A JSON representation of an example automatically generated synthetic data card produced by DataDreamer for Example 1. Synthetic data cards and model cards are automatically produced by recursively tracing through any steps, models, and trainers used to produce a given dataset or model. Each step, model, and trainer has associated metadata such as license information and citation information. DataDreamer collects this information and produces a synthetic data card (or model card) that reports the information along with reproducibility information like the reproducibility fingerprint.