Open Access | https://doi.org/10.25080/hprh2773

SciPy 2025

July 7 - July 13, 2025

Proceedings of the 24th Python in Science Conference
ISSN: 2575-9752

A Lightweight Pipeline for Rewards-Guided Synthetic Text Generation Using NeMo and RAPIDS

Jiajia Ding[1], **Arham Mehta**[1], and **Nirmal Juluru**[1]

[1]NVIDIA Corporation

Abstract

Synthetic Data Generation (SDG) plays an increasingly important role in modern machine learning workflows. Although progress has been made in structured data—tabular data with fixed schemas—especially with tools like Synthetic Data Vault (SDV) and Conditional Tabular Generative Adversarial Network (CTGAN), the generation of high-quality synthetic text remains a challenging and underdeveloped area. Many existing SDG pipelines lack mechanisms for semantic control, offer limited quality assurance, and are not designed with computational efficiency in mind.

To address these shortcomings, we introduce a lightweight and modular pipeline designed specifically for synthetic text data generation. Our approach integrates instruction-tuned large language models (LLMs) and reward-scoring models from NVIDIA NeMo with GPU-accelerated data processing tools from the RAPIDS ecosystem including cuDF and cuML.

The pipeline is implemented entirely within a Jupyter Notebook, allowing for transparency, reproducibility, and ease of use. It includes stages for data cleaning, semantic deduplication, reward-guided generation, and iterative data augmentation. It leverages RAPIDS libraries to accelerate operations such as clustering and filtering, enabling large-scale experimentation. Empirical results from applying this pipeline to a legal QA dataset demonstrate improved data quality, enhanced semantic diversity, and efficient execution on GPU hardware. The solution is designed to be accessible to data scientists who may not specialize in LLM development, thereby filling a crucial usability gap in the SDG landscape.

Keywords Synthetic Data Generation, Reward Scoring, Semantic Deduplication, GPU Acceleration, NeMo, RAPIDS, cuML, cuDF

Published Jul 10, 2025

Correspondence to
Jiajia Ding
allisond@nvidia.com

1. Introduction

Synthetic data is indispensable for developing machine learning models, when real data is scarce, sensitive, or restricted by privacy regulations. The ability to generate realistic, high-quality synthetic samples enables model development, fine-tuning, and benchmarking without exposing sensitive content. In structured data domains, tools such as SDV, CTGAN, and Gretel have emerged as practical solutions for generating tabular data that preserves statistical properties while mitigating privacy risks [1], [2]. However, the generation of unstructured text or semi-structured data (with flexible, implicit schemas), such as synthetic text, presents a far greater challenge. This complexity arises from the inherently semantic nature of language, which makes quality difficult to assess and even harder to enforce.

Text generation pipelines face several limitations that reduce their utility in practical workflows. First, many systems rely on large language models without sufficient post-generation validation, resulting in outputs that may be fluent but semantically irrelevant or redundant.

Second, few pipelines offer robust mechanisms to filter out low-quality or factually incorrect samples, which is essential when synthetic data is meant to support downstream tasks such as supervised learning or knowledge extraction. Third, most available implementations are not optimized for interactive or scalable use, often lacking GPU acceleration or modularity for iteration and analysis.

To address these gaps, we introduce a pipeline that is reward-aware, reproducible, and optimized for real-world use. Built entirely within a Jupyter notebook, the pipeline connects NeMo's instruction-tuned language models and reward scorers [3] with RAPIDS' GPU-accelerated data handling and clustering libraries [4]. The design is intentional in its focus on usability, aiming to provide data scientists with a transparent and reproducible tool for generating and curating synthetic text data. The pipeline supports full-cycle generation workflows, from preprocessing and semantic deduplication to filtering and output, all with minimal setup and high computational efficiency.

2. Background and Related Work

Structured data generation has seen considerable progress, with tools like SDV, CTGAN, and Gretel offering accessible APIs and statistical validation mechanisms [1], [2]. These systems have enabled synthetic data use in regulated environments such as healthcare and financial services. In contrast, synthetic text generation has largely been approached through either simple prompt-based generation using pretrained LLMs [5] or through more complex reinforcement learning pipelines, such as reinforcement learning with human feedback (RLHF) [6], [7]. Both approaches have their strengths but also introduce new challenges in terms of complexity, reproducibility, and cost.

Prompt-based pipelines are typically easy to initiate but provide limited control over output quality. Without built-in validation or filtering, the generated text can be inconsistent, incoherent, or repetitive. On the other hand, RLHF frameworks introduce reward models that evaluate generated outputs, allowing quality-guided training or generation. However, implementing RLHF involves substantial engineering efforts, including reward function design, reinforcement learning infrastructure, and hyperparameter tuning, which make it impractical for many real-world cases.

Our approach seeks to fill the methodological gap between these extremes. Rather than training models through reinforcement learning, we integrate pretrained reward models directly into the generation pipeline [3]. This allows us to evaluate and filter synthetic samples after generation using a scalar quality score that encapsulates properties such as helpfulness, coherence, relevance, and alignment with instructions [8]. This mechanism introduces a layer of semantic quality control without the need for retraining or fine-tuning.

In addition, semantic deduplication is incorporated to ensure content diversity. This component addresses the issue of redundancy, which is especially important in iterative generation scenarios. Finally, the pipeline is built around RAPIDS, a suite of GPU-accelerated Python libraries that leverage the same APIs as the traditional pandas dataframe and scikit-learn-based operations [4]. By integrating RAPIDS, we improve the scalability and responsiveness of the pipeline, requiring zero to minimal change in coding habits.

3. Methodology

3.1. Pipeline Overview

The synthetic text generation pipeline comprises five core modular stages: (1) data loading and cleaning, (2) semantic deduplication, (3) prompt-based generation, (4) reward scoring,

and (5) integration of accepted samples. To ensure computational efficiency, these stages are GPU-accelerated using the RAPIDS ecosystem.

The entire pipeline is orchestrated within a single notebook and supports interactive development, reproducibility, and iteration. This structure is essential for fast experimentation, especially when working with domain-specific data such as legal documents or biomedical QA pairs.

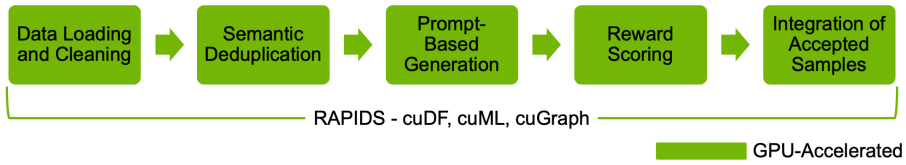

Figure 1. *The Synthetic Text Generation Pipeline*

3.2. Data Loading and Cleaning

The initial stage involves loading raw data, typically stored in JSON Lines (JSONL). The dataset we use as an example contains question-answer pairs and optional metadata such as titles, tags, or confidence scores.

Examples from the legal QA dataset.
Q: What is the difference between a bench trial and a jury trial?
A: In a bench trial the judge is the fact-finder, whereas in a jury trial the jury determines the facts while the judge rules on matters of law.

The data then undergoes a multi-step cleaning process to remove HTML tags, correct malformed Unicode characters, and apply filters based on word count and quality scores. For example, questions and answers are retained only if their lengths fall within acceptable ranges and their associated scores exceed a defined threshold. This step is crucial for establishing a high-quality seed set and preventing malformed data from degrading the performance of downstream generating and scoring models. NeMo Curator supports heuristic filters such as length constraints and reward-score thresholds to eliminate low-information or off-topic samples, improving both embedding quality and reward scoring.

```
# Load JSONL data and preview schema

# Import data
import pandas as pd

path = './peft-curation-with-sdg-70b/data/raw/splits/law-qa-train.jsonl'

# Load your initial dataset
dataset = pd.read_json(
   path, lines = True
)
```

```
# Define HTML/Unicode cleaners and simple filters

from bs4 import BeautifulSoup
import re

def clean_html(text):
   if not isinstance(text, str):
       return ""
   # Use BeautifulSoup to strip HTML tags
   text = BeautifulSoup(text, "lxml").get_text()
   # Replace multiple spaces or newlines with a single space and trim
   return re.sub(r"\s+", " ", text).strip()

import ftfy
from ftfy import TextFixerConfig

# Configuration for fixing Unicode text issues
fix_config = TextFixerConfig()

def fix_unicode(text):
   # Fix encoding and Unicode errors in text using ftfy.
   if not isinstance(text, str):
       return ""
   return ftfy.fix_text(text, config=fix_config)

def word_count(text, min_words=50, max_words=500):
   words = text.strip().split()
   # Check if the number of words in text falls within a specified range.
   return min_words <= len(words) <= max_words

def filter_low_score(score, threshold=0):
   # Determine if a numeric score meets or exceeds a given threshold.
   return score >= threshold
```

```
# Apply row-wise cleaning and content filters

def clean_filter_by_row(row):
   # Clean + fix each text field
   for field in ["title", "question", "answer"]:
       text = clean_html(row[field])
       text = fix_unicode(text)
       row[field] = text

   # Apply filters
   question_bool = word_count(row["question"]) and filter_low_score(float(row["question_score"]))
   answer_bool = word_count(row["answer"]) and filter_low_score(float(row["answer_score"]))

   return question_bool and answer_bool

def data_clean(dataset):
   dataset["keep"] = dataset.apply(clean_filter_by_row, axis=1)
   clean_dataset = dataset[dataset["keep"]].drop(columns=["keep"]).reset_index(drop=True)
   dataset.drop("keep", axis = 1, inplace = True)

   return clean_dataset
```

3.3. Semantic Deduplication

To avoid training or evaluating models on semantically repetitive content, we implement a deduplication step based on sentence embeddings. Each record is converted into a single textual representation by concatenating the title, question, and answer fields. Sentence embeddings are computed using a pretrained SentenceTransformer model and normalized using CuPy to enable efficient GPU operations. K-Means clustering is then applied in the embedding space to identify groups of similar samples. Within each cluster, cosine similarity is used to identify redundant records. Those with high proximity to the cluster centroid are marked for removal. This process significantly improves content diversity and reduces the likelihood of overfitting in downstream tasks. We define "duplicate" as two samples whose concatenated title, question, and answer embeddings have cosine similarity ≥ 0.99 within the same K-Means cluster.

While deduplication is crucial to reduce redundancy and overfitting, very aggressive thresholds can prune in-domain paraphrases or stylistic variations that are valuable for robustness. To mitigate this, we (i) experiment with variable similarity thresholds across data subsets (e.g., by topic/source/length) and (ii) perform human-in-the-loop auditing on a stratified sample to calibrate the operational definition of "semantic duplicate." These safeguards help retain informative variation while removing true repeats.

Examples from the legal QA dataset.
Removed as near-duplicates (≥ 0.99):
Q1: How does a bench trial differ from a jury trial?
Q2: Bench vs. jury trial—who determines the facts?
Kept as non-duplicates (< 0.99):
Q1: When can a defendant waive the right to a jury trial?
Q2: Can a judge overturn a jury verdict?

In tie cases, we retain the item farthest from the cluster centroid to preserve variety. This procedure improves content diversity and reduces overfitting risk in downstream tasks while avoiding the removal of useful paraphrases.

```
# Compute embeddings, cluster with K-Means, and mark near-centroid duplicates

def semantic_dedupe(dataset, n_clusters = 1000, eps_to_extract = 0.01):
   dataset["text"] = dataset["title"] + dataset["question"] + dataset["answer"]


   # Generate normalized sentence embeddings
   from sentence_transformers import SentenceTransformer
   import cupy as cp


   model = SentenceTransformer("all-MiniLM-L6-v2")
   embeddings = model.encode(dataset["text"].tolist(), batch_size = 128, show_progress_bar = True,
convert_to_numpy = True)
   embeddings_norm = cp.array(embeddings)/cp.linalg.norm(cp.array(embeddings), axis = 1, keepdims =
True)


   # Perform KMeans with cosine normalization (Euclidean KMeans on normalized vectors)
   from sklearn.cluster import KMeans
   from sklearn.metrics.pairwise import cosine_similarity


   kmeans = KMeans(n_clusters = n_clusters, random_state = 1234, max_iter = 100)
   labels = kmeans.fit_predict(embeddings_norm)
   centroids = kmeans.cluster_centers_


   # Deduplication logic (find points within eps_to_extract)
   dedup_indices = []
   for cluster_id in range(n_clusters):
       cluster_points = cp.where(cp.array(labels) == cluster_id)[0]
       if len(cluster_points) == 0:
           continue

       cluster_embeddings = embeddings_norm[cluster_points].get()
       centroid = centroids[cluster_id].reshape(1, -1)
       sims = cosine_similarity(cluster_embeddings, centroid).flatten()
       within_eps = cluster_points[sims >= (1 - eps_to_extract)]


       if len(within_eps) > 0:
           dedup_indices.extend(within_eps)

   # Get deduplicated data
   mask = dataset.index.isin(dedup_indices)
   kept_data = dataset[~mask]


   # Drop the added column
   dataset.drop("text", axis = 1, inplace = True)
   kept_data.drop("text", axis = 1, inplace = True)


   return kept_data
```

In the above code snippet, n_clusters (also written as k) is the number of K-Means clusters used to pre-group records in embedding space before duplicate checks; it controls granularity (larger k arrow.r finer, more homogeneous groups; smaller k arrow.r coarser groups). eps_to_extract (ε) is a within-cluster proximity/similarity threshold for marking items as redundant (e.g., "duplicate if cosine similarity $\geq \tau$" or "duplicate if distance to the centroid $\leq \varepsilon$"), which controls how aggressively near-duplicates are removed. Both are user-tunable hyperparameters; appropriate values depend on corpus size, diversity, and the desired precision-recall trade-off for deduplication. Our implementation exposes both as configuration options so readers can adjust them without changing the pipeline.

3.4. Synthetic Data Generation with Rewards

Synthetic text generation is performed using the Llama 3.1 Nemotron 70B instruct model, which supports tasks such as question generation and paraphrasing. Prompt templates are designed for three specific functions: generating questions from answers, paraphrasing questions, and paraphrasing answers. For each function, the generator outputs a candidate sample that is then evaluated by the Llama 3.1 Nemotron 70B Reward model. This reward model assigns a scalar score reflecting the overall quality of the text. Only those samples with scores falling within a specified range are retained. This scoring function enables automated quality control and reduces reliance on manual inspection or external validation. Notably, the Llama 3.1 Nemotron 70B Reward model is ranked No. 1 on the Hugging Face RewardBench leaderboard as of October 1, 2024 [9], outperforming competing reward models across overall, chat, safety, and reasoning categories.

Although our experiments use NVIDIA Nemotron models for both generation and reward scoring, the pipeline is model-agnostic: the generator and reward model are accessed through the same chat-completion interface and can be replaced by any instruction-tuned LLM and any scalar-scoring reward model that expose comparable APIs. Practically, portability requires (i) an instruction-following endpoint with controllable sampling (temperature, top_p, optional seed), (ii) sufficient context length for our prompts, and (iii) a reward signal, either a native reward model or a proxy judge that outputs a single scalar per sample. To remain robust across models, we store raw scores and apply per-model score normalization, selecting candidates by percentile thresholds rather than absolute cutoffs. Prompts are maintained as templated strings and versioned, so swapping models is a configuration change rather than a code change.

```
# Synthetic Data Generation with Rewards

def normalize_score(score, min_val=-34.75, max_val=-5.125):
   # Normalize a scalar score to [-1, 1] range.
   # We use a minimum value of −34.75 and a maximum value of −5.125, computed from the raw dataset,
for normalization in this SDG process.
   normalized = 2 * (score - min_val) / (max_val - min_val) - 1
   return normalized

def initialize_generator():
   # Initialize Nemotron generator and client.
   from openai import OpenAI
   from nemo_curator import OpenAIClient
   from nemo_curator.synthetic import NemotronGenerator

   openai_client = OpenAI(
       base_url="https://integrate.api.nvidia.com/v1",
       api_key=""
   )
   client = OpenAIClient(openai_client)
   generator = NemotronGenerator(client)
   return generator, client

def generate_variants(generator, document, prompt_template, n_variants, sdg_model, sdg_model_kwargs):
   # Generate paraphrased or question variants from a document.
   generated = generator.generate_closed_qa_instructions(
       document=document,
       n_openlines=n_variants,
       prompt_template=prompt_template,
       model=sdg_model,
       model_kwargs=sdg_model_kwargs,
   )
   variant_list = [entry.strip("*").strip() for entry in generated[0].split("\n") if entry]
   return variant_list

def build_synthetic_row(row, i, gen_question, gen_answer, gen_title, rewards_normalized):
   # Build a new row dictionary for synthetic data.
   return {
       "id": f"{row['id']}-synth-{i}",
       "question": gen_question,
       "answer": gen_answer,
       "title": gen_title,
       "file_name": "law-stackexchange-questions-answers.json.synth",
       "tags": row["tags"],
       "question_score": rewards_normalized,
       "answer_score": rewards_normalized
   }

def SDG(dataset, sample_size_perc=0.001, rewards_threshold=0, n_variants=1):
   # Sample rows, generate/paraphrase Q&As, generate reward scores, filter by threshold, and return
curated synthetic pairs as a DataFrame.
   import random
   import pandas as pd

   generator, client = initialize_generator()

   sdg_model = "nvdev/nvidia/llama-3.1-nemotron-70b-instruct"
   sdg_model_kwargs = {
       "temperature": 0.5,
       "top_p": 0.9,
       "max_tokens": 1024,
       "seed": 1234
   }
   reward_model = "nvdev/nvidia/llama-3.1-nemotron-70b-reward"

   PROMPT_GENERATE_QUESTIONS_FROM_ANSWER = """TEXT:
   {document}

   Given the above text, generate exactly {n_openlines} questions that can be answered by the text.
All questions must be answerable by the text and be relevant to the text.
   Do not directly reference the text in the questions.
   Every question should be a complete sentence and end with a question mark. There should be no
other text besides the questions.
   Begin each question with `* ` and end each question with a newline character. Also, each question
```

```
must be concise.
   Make sure to generate exactly {n_openlines} questions.
   """

   PROMPT_PARAPHRASE_TEXT = """TEXT:
   {document}

   Given the above text, paraphrase the text. Produce exactly {n_openlines} variants.
   There should be no other text besides the paraphrased text.
   The paraphrased text must be shorter than the original text. The paraphrased text must be
factually correct and relevant to the original text.
   Begin each variant with `* ` and end each variant with a newline character.
   Make sure to generate exactly {n_openlines} variants.
   """

   output = []
   N = len(dataset)
   n = int(N * sample_size_perc)
   sample_indices = random.sample(range(N), n)

   for idx in sample_indices:
       row = dataset.iloc[idx]
       question = row["question"]
       answer = row["answer"]

       gen_title_lst = generate_variants(generator, answer, PROMPT_GENERATE_QUESTIONS_FROM_ANSWER,
n_variants, sdg_model, sdg_model_kwargs)
       gen_question_lst = generate_variants(generator, question, PROMPT_PARAPHRASE_TEXT, n_variants,
sdg_model, sdg_model_kwargs)
       gen_answer_lst = generate_variants(generator, answer, PROMPT_PARAPHRASE_TEXT, n_variants,
sdg_model, sdg_model_kwargs)

       for i in range(n_variants):
           messages = [
               {"role": "user", "content": f"{gen_title_lst[i]}\n\n{gen_question_lst[i]}"},
               {"role": "assistant", "content": f"{gen_answer_lst[i]}"},
           ]
           rewards = client.query_reward_model(messages=messages, model=reward_model)
           rewards_normalized = normalize_score(rewards)

           if rewards_normalized >= rewards_threshold:
               new_row = build_synthetic_row(row, i, gen_question_lst[i], gen_answer_lst[i],
gen_title_lst[i], rewards_normalized)
               output.append(new_row)

   output_df = pd.DataFrame(output)
   return output_df
```

3.5. Complete SDG Pipeline

The complete pipeline integrates all components into an iterative loop. The process begins with cleaning and deduplicating the original dataset to establish a high-quality baseline. In each subsequent iteration, new synthetic samples are generated and merged with the deduplicated dataset from the previous round. The combined dataset is then subjected to semantic deduplication to eliminate newly introduced redundancy. This iterative procedure supports the progressive enrichment of the dataset while maintaining high semantic diversity and quality standards. All outputs, including intermediate generations and reward scores, are retained to facilitate auditability and further analysis.

```
# Synthetic Data Generation Pipeline

def SDG_Rewards_Pipeline(dataset, round_num, n_clusters = 1000, eps_to_extact = 0.01,
sample_size_perc = 0.001, rewards_threshold = 0, n_variants = 1):

   cleaned_dataset = data_clean(dataset)
   deduped_dataset = semantic_dedupe(dataset, n_clusters = n_clusters, eps_to_extract =
eps_to_extact)
   print(f"After the initial curation, the dataset has {len(deduped_dataset)} records (originally
{len(dataset)}).")

   dataset = deduped_dataset
   for num in range(round_num):

      sdg_dataset = SDG(dataset, sample_size_perc = sample_size_perc, rewards_threshold =
rewards_threshold, n_variants = n_variants)
      dataset = pd.concat([dataset, sdg_dataset], axis = 0, ignore_index = True)
      deduped_dataset = semantic_dedupe(dataset)

      print(f"After round {num + 1}, the dataset has {len(deduped_dataset)} records (originally
{len(dataset)})")

   return deduped_dataset
```

3.6. Accelerating End-to-End Pipeline

To support high throughput execution, the pipeline uses GPU acceleration for key operations through RAPIDS libraries such as cuDF and cuML [4]. With cuDF, common pandas DataFrame operations, including filtering, joining, and aggregating, can be seamlessly accelerated on the GPU. With cuML, machine learning algorithms such as K-Means clustering also benefit from GPU acceleration, resulting in significant reductions in execution time.

Users can enable this acceleration by simply loading the RAPIDS extensions at the beginning of their Jupyter Notebook or JupyterLab, without needing to modify their existing pandas or scikit-learn code. This transparent acceleration allows the pipeline to interactively process tens of thousands or even millions of records, making it practical for rapid prototyping as well as scalable production workflows.

```
# Activate GPU Acceleration of cuML and cuDF in Jupyter Notebook or JupyterLab

%load_ext cuml.accel
%load_ext cudf.pandas
```

For terminal workflows, users can run the same code as a script with the CLI, which activates both the cuML accelerator and the cuDF pandas accelerator for the session.

```
# Activate GPU Acceleration of cuML and cuDF in the terminal

python -m cudf.pandas -m cuml.accel sgd.py
```

4. Experiments and Results

4.1. Use Case Setup

To evaluate the performance and effectiveness of the proposed pipeline, we applied it to a real-world dataset containing 19,474 text records. This dataset served as the basis for a multi-stage process involving semantic deduplication, prompt-based generation, reward scoring, and integration of accepted samples. The evaluation was conducted using a single

GPU-enabled machine to validate the feasibility of interactive, large-scale processing within a notebook environment.

An initial baseline was established by cleaning and deduplicating the original dataset. This step reduced the corpus to 12,244 records, removing approximately 37.2% of entries that failed to meet minimal content quality or uniqueness standards. This baseline served as the foundation for all subsequent generations and filtering stages.

4.2. Quantitative Evaluation

The pipeline was configured to run for 10 rounds, each involving the generation of synthetic samples sampled at a rate of 0.1% from the current dataset, with a number of variants of 2. A reward threshold of 0 was applied to filter outputs based on quality, using normalized scalar reward scores to retain only fluent, relevant, and contextually appropriate text.

As shown in Table 1, each round resulted in a modest increase in the dataset size before deduplication. For example, after round 1, the dataset increased from 12,244 to 12,251 records. This pattern continued across all 10 iterations, with post-generation counts ranging from 12,251 to 12,321. After each merge, semantic deduplication was applied, bringing the dataset size down, confirming that only non-redundant and unique samples were retained.

Table 1. *Dataset size across 10 rounds*

Round	Original Records (Pre-Dedup)	Records After Dedup
Initial	19,474	12,244
1	12,251	12,248
2	12,257	12,257
3	12,266	12,264
4	12,274	12,271
5	12,285	12,282
6	12,293	12,291
7	12,301	12,300
8	12,310	12,309
9	12,313	12,311
10	12,321	12,317

This process illustrates the effectiveness of the deduplication step. Across all 10 rounds, redundant content introduced during generation was systematically identified and removed. Despite generating over 400 synthetic candidates in accumulation, the dataset size only increased slightly, reflecting strict enforcement of semantic quality constraints.

4.3. Visual Outputs

Raw vs Curated. Figure 2 (raw dataset, N=19,474) and Figure 3 (curated dataset, N=77) compare the distribution of the normalized reward score before and after curation. The raw set shows a broad, roughly unimodal spread from $\approx$-0.8 to 0.8 with a mode slightly right of zero and a positive tail to $\sim$0.7–0.8, reflecting high variability in quality. By contrast, the curated set eliminates negatives and concentrates mass in 0.0–0.5 (peaking $\sim$0.10–0.25) with reduced variance—a clear rightward shift.

These visualizations support the filtering logic: clustering and semantic deduplication remove repeated phrasing, and reward-based selection favors higher-coherence, factually

correct content. In short, the procedure is content-selective rather than merely size-reducing, retaining the most valuable and accurate samples for downstream use.

In the raw dataset (Figure 2), the normalized reward score is broadly distributed (≈-0.8 to +0.8) with a single mode slightly right of zero. Density peaks near 0.1 and stays substantial through ∼0.3–0.4; negatives are thinner, while the positive tail extends farther (to ∼0.7–0.8). Overall, the raw set skews mildly positive, with many mid-quality samples and few extreme outliers.

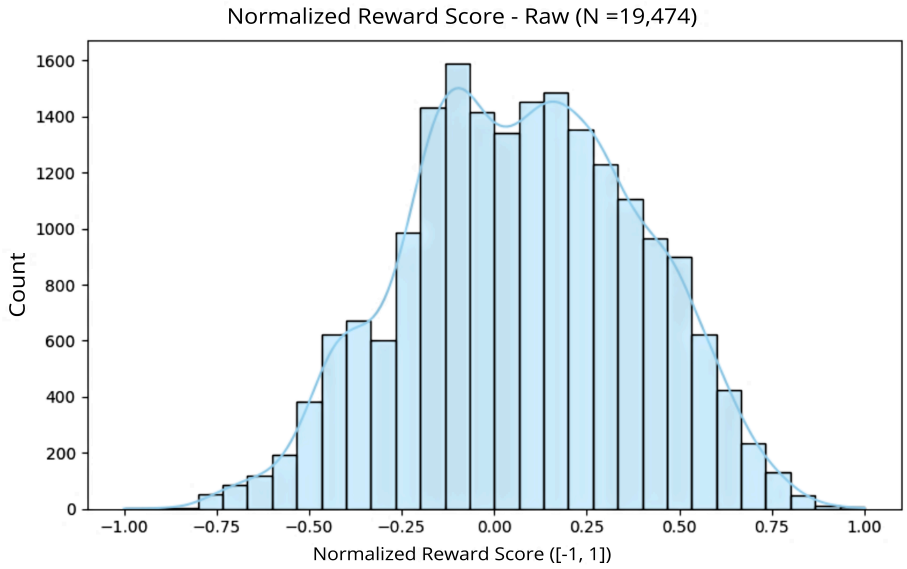

Figure 2. *Normalized Reward Score Distribution of Raw Data*

In the curated dataset (Figure 3), the distribution concentrates at higher scores (≈0.0–0.5), eliminating negatives and tightening the spread. Density peaks around 0.10–0.25, then declines with a thin right tail to ∼0.5; relative to the raw set, it's right-shifted and less variable, indicating curation favored higher-reward examples.

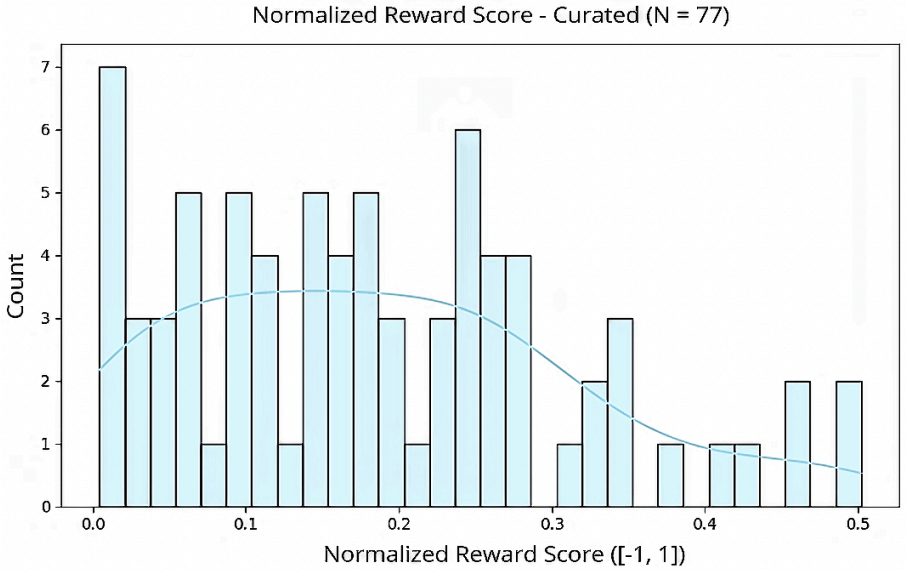

Figure 3. *Normalized Reward Score Distribution of Curated Data*

5. Conclusions

We have presented a reproducible and scalable pipeline for reward-guided synthetic text generation. The pipeline integrates instruction-tuned generation models with semantic deduplication and quality scoring, with end-to-end GPU acceleration. By relying on pre-trained components and avoiding the complexity of reinforcement learning, the pipeline remains accessible to data scientists without deep expertise in LLM training. Its modular structure supports iterative enrichment of datasets while preserving semantic diversity and ensuring quality. All components are implemented in a Jupyter notebook, supporting transparency, auditability, and ease of extension. This work contributes a practical and scientifically grounded approach to responsible synthetic text generation in real-world machine learning settings.

References

[1] N. Patki, R. Wedge, and K. Veeramachaneni, "The synthetic data vault," in *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, 2016, pp. 399–410. doi: 10.1109/DSAA.2016.49.

[2] SDV Developers, "SDV: Synthetic Data Vault." [Online]. Available: https://sdv.dev/

[3] NVIDIA, "NeMo Curator and Nemotron Models." [Online]. Available: https://developer.nvidia.com/nemo

[4] RAPIDS AI, "cuDF and cuML: GPU-accelerated DataFrames and Machine Learning." [Online]. Available: https://rapids.ai/

[5] T. Wolf *et al.*, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2020, pp. 38–45. doi: 10.18653/v1/2020.emnlp-demos.6.

[6] P. F. Christiano, J. Leike, T. B. Brown, M. Martic, S. Legg, and D. Amodei, "Deep Reinforcement Learning from Human Preferences," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. doi: 10.48550/arXiv.1706.03741.

[7] L. Ouyang, J. Wu, X. Jiang, and others, "Training Language Models to Follow Instructions with Human Feedback," *arXiv preprint arXiv:2203.02155*, 2022, doi: 10.48550/arXiv.2203.02155.

[8] H. Gadre, S. Shen, W. L. Hamilton, and C. Raffel, "RewardBench: Benchmarking Alignment Rewards for Language Models," *arXiv preprint arXiv:2410.01257*, 2024, doi: 10.57967/hf/2457.

[9] Hugging Face, "RewardBench leaderboard: Llama 3.1 Nemotron 70B Reward." [Online]. Available: https://huggingface.co/nvidia/Llama-3.1-Nemotron-70B-Reward