Team026 による大規模言語モデルと SymPy 協調を用いた数学タスク推論パイプラインの再現手順

Team026 猫のちくわ: 森 直樹 [1]　岡本 悠弥 [1]　西元 大世 [1]　森脇 康介 [1]　許 聖志 [1]
小川 恭子 [2]
[1] 大阪公立大学　[2] 三菱電機株式会社

1　はじめに

本稿は，LLM-jp Tuning Competition 2026 [1] の数学タスクにおけるオープン枠で Team026 が提出したシステムの再現手順である．このコンペティションでは日本の中高数学を対象としており，運営からテストデータは 500 問であることが事前に示されている．以降では Team026 が提出した内容を本システムと呼ぶ．

本システムはツール統合推論 (Tool-Integrated Reasoning, TIR) に基づくパイプラインであり，大規模言語モデル (Large Language Model, LLM) に直接解答を生成させず，Python の数式処理ライブラリである SymPy [2] コードの生成のみを担当させている．生成されたコードをサンドボックスで実行し，その出力を正規化して最終解答とする．Program-aided Language Models (PAL) [3] および Program of Thoughts (PoT) [4] の枠組みに自己改善ループと出力正規化を統合した構成である．

実行環境は AI Bridging Cloud Infrastructure (ABCI) 3.0 であり，Graphics Processing Unit (GPU) として H200 を 8 基搭載したノードを利用し，Singularity コンテナ上でインターネット接続不可のオフライン環境を想定している．

2　ディレクトリ構成

提出リポジトリの主要な構成を以下に示す．

```
FT-LLM2026/
├── train_full_sft.py
├── train.jsonl / val.jsonl
├── models/
│   ├── llm-jp-4-8b-instruct4/
│   ├── llm-jp-4-8b-math-full/
│   └── embedding/
│       └── intfloat--multilingual-e5-small/
└── singularity_submission/
    ├── main.py
    ├── submission.def
    ├── pyproject.toml / uv.lock
    └── src/
        ├── base_prompt.py
        ├── code_executor.py
        ├── embedding_selector.py
        ├── few_shot_pool.py
        ├── format_converter.py
        ├── generate_code_error_modify.py
        ├── keyword_matcher.py
        ├── process_llm.py
        ├── prompt_builder.py
        └── rules.py
```

3　データセット構築

3.1　ライセンス制約

オープン枠の要件として，最終システムがオープンソースソフトウェア (Open Source Software, OSS) として公開可能であることが求められる．Gemini の出力，Hendrycks らの MATH，OlympiadBench，Creative Commons Attribution-NonCommercial (CC BY-NC) データ，Llama や Gemma の派生生成物などは使用できない．使用可能なデータは，OpenMathInstruct-1 のうち GSM8K 部分，オープンライセンスであることが確認できた DeepSeek 系のモデルの出力，Qwen などの Apache-2.0 系モデル出力，および 100 問からなる運営配布の `dev.jsonl` である．

3.2　データ構造

各エントリは問題文である `problem`，LaTeX 形式の正解である `solution`，SymPy スクリプトである `code` からなる 3 つのフィールドで構成される．`code` は `from sympy import *` で開始し，関数定義としての `def` などを含まないフラットな構造とし，末尾にある `print(<expression>)` で解を出力する．`Rational`，`sqrt`，`pi` などの SymPy オブジェクトによる厳密な表現を維持し，明示的な指示がない限り `.evalf()` を用いた小数化は禁止する．

表 1　SFT のハイパーパラメータ

パラメータ	設定値
最大系列長	2048
エポック数	5
勾配累積ステップ	4 (effective batch size 8)
デバイスあたりのバッチサイズ	2
最適化アルゴリズム	AdamW
学習率	2×10^{-5}
スケジューラ	Linear
Warmup	50 ステップ
演算精度	bf16

3.3　データ拡張

まず運営が公開している例題 100 問からなる `dev.jsonl` の論理構造を保持したまま，数値や条件をプログラム的に置換して類題を生成した．加えて，最大最小，確率，三角関数，数列，複素数，図形などの新規問題を手作業で構築した．Qwen など使用可能な LLM によるコード自動生成も予備実験で試したが，出力の質が低くデータの整合性が崩れるため，規則的な生成と手動での検査を中心とした方針を採用した．またコード生成の失敗率を下げるために，Few-shot も併用している．構築したデータは訓練用，検証用，検索拡張生成 (Retrieval-Augmented Generation, RAG) プール 194 例に分割した．

4　教師ありファインチューニング

4.1　設定

表 1 にハイパーパラメータを示す．ベースモデルは `llm-jp-4-8b-instruct4` [5] である．

4.2　応答部分のみの損失計算

コード生成部分のみに集中するためシステムプロンプトと問題文からなるプロンプト部のトークンラベルを −100 に設定し，損失計算から除外した．具体的には `train_full_sft.py` 内で `ASSISTANT_PREFIX` より前のトークンにマスク処理を適用している．

4.3　実行コマンド

```
python3 train_full_sft.py \
--model_name models/llm-jp-4-8b-instruct4 \
--train_file train.jsonl \
--val_file val.jsonl \
--output_dir models/llm-jp-4-8b-math-full \
--max_length 2048 --batch_size 2 \
--grad_accum 4 --epochs 5 \
--lr 2e-5 --warmup_steps 50
```

表 2　使用したテンプレート

テンプレート名	内容
itertools	組合せ論
binomial	二項分布
hypergeometric	超幾何分布
derivative	微分方程式
integral	積分
limit	極限
vector	ベクトル
complex_numbers	複素数
sequence	数列
kyo_log	対数
kyo_velo	速さの問題
kyo_assemble	集合
kyo_stats_var	標準正規分布
general	上記以外

5　推論パイプライン

5.1　システムプロンプト

システムプロンプトは `src/base_prompt.py` に定義されている．LLM に対して，SymPy を用いた Python コードでの解答を指示する．インポート制約として `from sympy import *` を基本とし，確率や統計など特定の分野に限定して追加のインポート文を加えるよう指示している．また，関数定義の `def` を禁止し，`print(<expression>)` による出力を要求した．

5.2　テンプレート分類

`src/keyword_matcher.py` において，問題文のキーワードを解析し，13 種以上のテンプレートに分類する．`TEMPLATE_MATCHING_RULES` リストに優先度，判定関数，テンプレート名の組を定義し，優先度順に評価する．表 2 に使用したテンプレートを示す．各テンプレートに対応するルール文字列が `src/rules.py` に定義されており，プロンプトに動的に挿入される．

5.3　動的 Few-shot 選択

埋め込みモデル `intfloat/multilingual-e5-small` [6] を用いる．モデルはコンテナ内のローカル環境に配置し，`local_files_only=True` を指定する．選択手順は以下のとおりである．

1. 起動時に Few-shot プールにある 194 例すべての問題文をエンコードし，キャッシュとして保存する．
2. 入力問題文を `query: {問題文}`，プール側を `passage: {問題文}` の形式でエンコードする．この形式は E5 系の推奨仕様である．
3. コサイン類似度で上位 $k = 3$ 件を選択する．
4. 最大類似度が閾値 0.9 未満の場合は Few-shot を挿入せず 0-shot に切り替える．

5.4 プロンプト構造

`src/prompt_builder.py` が構築するプロンプトは SYSTEM メッセージと USER メッセージで構成される．SYSTEM メッセージにはシステムプロンプトを配置し，USER メッセージにはベースプロンプト，テンプレート別のルール，Few-shot 用の例題を最大 3 件，対象問題文を含める．

5.5 コード実行

`src/code_executor.py` がサンドボックス環境でのコード実行を担う．グローバル名前空間には `sympy`，`math`，`Fraction`，`Decimal`，`numpy` のみを用いる．タイムアウトは 10 秒とし，Unix では `SIGALRM`，それ以外では `threading.Timer` を利用する．標準出力が空の場合も実行エラーとして扱う．

5.6 自己改善ループ

Python コードを出力する戦略は，コードが正しく生成されれば必ず正解が得られるという利点がある一方で，正確なコードを出力しなければならないという制約も存在する．予備実験により元モデルのコード生成能力が十分ではないことが確認されたため，`src/generate_code_error_modify.py` で出力コードの自己改善ループを実装した．初回は生成温度を 0.0，最大トークン数を 512 とし vLLM [7] により生成する．エラー検知条件はタイムアウト，Python 実行時例外，標準出力空の 3 種である．

エラー検知時は，修正プロンプトに元の問題文と失敗コードおよびエラーログを統合し再生成する．再試行時の生成温度は試行回数 $j = 0, \ldots, 4$ に対し $0.04(j+1)$，すなわち 0.04，0.08，0.12，0.16，0.20 と段階的に上昇させる．最大 5 回の再試行を実施し，初回を含めて計 6 回の実行すべてが失敗した場合，0 を出力する代替コードを実行する．

表 3 出力正規化の主な変換規則

SymPy 出力	変換後
`Interval.open(5,10)`	$5 < x < 10$，変数名は問題文から推定する
`Piecewise(...)`	主要ケースを抽出し LaTeX 化する
2 要素リスト，無理数	中心値 ± 差の形式
2 要素リスト，有理数	カンマ区切り列挙
論理 OR	不等式列挙
論理 AND	連鎖不等式
その他	`sympify` から `latex(simplify(...))` へ変換

5.7 出力正規化

今回のコンペティションでは，出力の正解判定が厳格であり正解していても出力形式の問題で失敗とされる例が散見された．そこで，可能な範囲で失敗例を確認し `src/format_converter.py` によって SymPy の未整形出力を LaTeX 形式に変換した．表 3 に主な変換規則を示す．

変換後，解答に `$` を付与し，`\boxed{}` 内に格納する．

6 Singularity イメージの構築と実行

6.1 モデルの事前配置

SFT 済みモデルおよび埋め込みモデルを `singularity_submission/models/` 配下に配置する．チェックポイントファイルは除外する．

6.2 イメージのビルド

`submission.def` の `%files` セクションにモデルのコピー先を記述し，以下を実行する．

```
singularity build --fakeroot \
~/submission.sif submission.def \
&& mv ~/submission.sif dist/
```

イメージサイズが 64 GB 以下であることを確認する．

6.3 実行

```
singularity run --nv --writable-tmpfs \
--cleanenv --env CUDA_VISIBLE_DEVICES=0 \
--network none \
--home "$(mktemp -d -p "$(pwd)" \
.singularity_home.XXXXXX)" \
dist/submission.sif \
--input_path input.jsonl \
--output_path output.jsonl
```

`--nv` は GPU アクセス，`--writable-tmpfs` は一時ファイルシステム，`--cleanenv` は環境変数の初期化，`--network none` はオフライン強制である．出力は JSONL 形式となる．

表 4　構成別の正答率

モデル	機能	train	validation
SFT モデル	動的 Few-shot と自己改善	95.6%	92.8%
SFT モデル	固定プロンプト	99.1%	88.4%
ベースモデル	動的 Few-shot と自己改善	80.3%	74.7%
ベースモデル	固定プロンプト	14.4%	9.7%

7　実験結果

表 4 に各構成の正答率を示す.

SFT により正答率は大幅に向上した. 固定プロンプト構成では train データに対する 99.1% と validation データに対する 88.4% で乖離が大きく, プロンプト構造への過適合が示唆される. 動的 Few-shot は入力ごとにプロンプト構造を変化させるため正則化として機能しており, SFT と動的 Few-shot および自己改善を統合した構成で train と validation の乖離を最小化できた.

8　おわりに

本稿では, Team026 による大規模言語モデルと SymPy 協調を用いた数学タスク推論パイプラインの再現手順について述べた. 今後の課題としては, データセットの拡充やコード生成の成功率の向上等が考えられる.

参考文献

[1] LLM-jp tuning competition 2026, 2026. https://llm-jp.nii.ac.jp/.

[2] SymPy Development Team. SymPy: Python library for symbolic mathematics, 2024. https://www.sympy.org/.

[3] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. In **Proceedings of the 40th International Conference on Machine Learning**, 2023.

[4] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. **Transactions on Machine Learning Research**, 2023.

[5] LLM-jp. llm-jp-4-8b-instruct4, 2025. https://huggingface.co/llm-jp/.

[6] Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. Text embeddings by weakly-supervised contrastive pre-training. arXiv preprint arXiv:2212.03533, 2024.

[7] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In **Proceedings of the 29th Symposium on Operating Systems Principles**, 2023.