MULTI-MODULE GRPO: COMPOSING POLICY GRADIENTS AND PROMPT OPTIMIZATION FOR LANGUAGE MODEL PROGRAMS

Noah Ziems[*1], **Dilara Soylu**[*2], **Lakshya A Agrawal**[*3], **Isaac Miller**[7], **Liheng Lai**[3], **Chen Qian**[5], **Karel D'Oosterlinck**[6], **Kaiqiang Song**[8], **Meng Jiang**[1], **Dan Klein**[3], **Matei Zaharia**[3,5], **Christopher Potts**[2], **Omar Khattab**[4]

[1]University of Notre Dame [2]Stanford University [3]UC Berkeley
[4]MIT [5]Databricks [6]Contextual AI [7]Normal Computing [8]Zoom, Inc.

ABSTRACT

Group Relative Policy Optimization (GRPO) has proven to be highly effective as a tool for post-training language models (LMs). However, AI systems are increasingly expressed as modular programs that mix together multiple LM calls and other tools, and it is not clear how best to leverage GRPO to improve these systems. We begin to address this challenge by introducing MMGRPO, a multi-module generalization of GRPO. MMGRPO defines a module-level grouping strategy aligning module calls across distinct rollouts, even in the presence of trajectories of variable length or partial intermediate inputs. We find that MMGRPO improves accuracy by 7% on average across classification, multi-hop claim verification, and privacy-preserving delegation tasks against the unadapted model. In addition, although MMGRPO does not consistently outperform prompt optimization, the two can be very effectively combined using the BetterTogether method of Soylu et al. (2024), leading to over 5% in gains compared to prompt optimization alone. MMGRPO is open-sourced in the DSPy library as the `dspy.GRPO` optimizer.

https://github.com/stanfordnlp/dspy

1 INTRODUCTION

Modern natural language processing (NLP) systems are increasingly implemented as modular systems, in which each module is responsible for a well-specified subtask that contributes to solving a broader objective. A canonical example is "multi-hop" research, where the system responds to a question by iteratively using a *query generation* LM module to produce a search query, passing that query to a retriever, and finally feeding all iteratively retrieved passages into a *response generation* LM module to produce the final output. The explicit modularization of such systems makes their behavior controllable, akin to conventional software, and allows for structured optimization of individual components, leveraging the priors of the LM differently for each module.

Group Relative Policy Optimization (GRPO; Shao et al. 2024) has recently emerged as a powerful method for fine-tuning language models (LMs) in the final stages of training. By leveraging relative rewards within groups of "reasoning" rollouts that share the same prompt, GRPO offers a simple alternative to Proximal Policy Optimization (PPO; Schulman et al. 2017). However, GRPO was originally designed for single-stage settings where each rollout consists of a single autoregressive LM call, and it is not obvious how to best extend it to systems composed of multiple such calls with distinct prompt templates.

In this paper, we ask how post-training RL algorithms such as GRPO could be applied to such multi-module LM programs. The key challenges are as follows. Each rollout of an LM program may invoke several distinct LM modules, each with its own prompt template and context. In addition, rollouts generated from the same input can differ in both length and structure, due to variations in

* Equal contribution.

Strategy	Banking77		PAPILLON		$\text{Hover}_{\text{4-HOP}}$		Avg Scores		
	llama3.1	qwen3	llama3.1	qwen3	llama3.1	qwen3	llama3.1	qwen3	All
Baseline Strategies:									
Vanilla CoT	58.40	64.60	76.19	78.31	59.53	60.60	64.71	67.84	66.27
MIPROv2 (PO)	59.40	65.93	83.90	78.06	63.39	68.58	68.90	70.86	69.88
MMGRPO *Strategies*:									
MMGRPO	63.33	64.93	83.94	**83.29**	60.21	70.97	69.16	73.06	71.11
BetterTogether(PO, MMGRPO)	**63.67**	**69.13**	**86.48**	81.12	**68.27**	**72.15**	**72.80**	**74.13**	**73.47**

Table 1: Percentage accuracy scores for each strategy, averaged across 3 seeds computed on the dev set for each task. LMs names shortened from `llama3.1-8b-instruct` and `qwen3-8b`. **Bold** font indicates the top score in each column.**[todo]** Dilara to check bolds given the statistical framework; need to look into significance. This caption needs to be extended to be standalone.

control flow or early termination from, e.g., parsing failures, and often produce disjoint intermediate inputs and outputs.

In response to these challenges, we implement MMGRPO, a structured framework for applying GRPO to multi-module setups. The core idea is to relax GRPO's requirement for shared inputs by grouping rollouts at the *module-level*, aligning structurally comparable module calls across different trajectories. This approach enables GRPO-style policy gradient updates without requiring shared histories or module-level inputs across rollouts. We open-source MMGRPO as an off-the-shelf optimizer for arbitrary compound AI systems as part of the DSPy library at dspy.ai.

Ours is the first implementation of GRPO that applies to sophisticated pipelines of LMs. This enables us to conduct a controlled comparison of three approaches to optimizing modular AI systems: prompt optimization (PO), online reinforcement learning via MMGRPO, and their combination using the BetterTogether framework (Soylu et al., 2024). Our evaluation spans three diverse LM program tasks: classification (`Banking77`, Casanueva et al. 2020), multi-hop claim verification (`Hover`, Jiang et al. 2020, and privacy-conscious delegation (`PAPILLON`, Siyan et al. 2024). Each involves different reasoning styles and control flow structures. Experiments are run using two open-source LMs, `llama3.1-8b-instruct` (Grattafiori et al., 2024) and `qwen3-8b` (Yang et al., 2025).

Our results are summarized in Table 1. Across these settings, MMGRPO improves performance by 7.3% on average against the model's reasoning performance and by 5.14% compared to prompt-optimized baselines via MIPROv2 (Opsahl-Ong et al., 2024). While MMGRPO does not always surpass the prompt optimized programs, it complements them effectively: staging MMGRPO and MIPROv2 à la BetterTogether consistently yields higher performance than either method alone, by 10.86% compared to the model's baseline reasoning performance. These findings suggest that policy gradient RL and PO offer complementary benefits for LM program training, and we advocate for future work exploring their integration in both offline and online settings.

2 Preliminaries

GRPO is an online policy gradient method for LM optimization that operates over *groups* of trajectories sharing the *same input prompt* in *single-stage* tasks. The GRPO objective encourages the current policy $p_{\theta_{\text{old}}}$, parametrized by LM weights θ_{old}, to upweight relatively high-reward completions within a group, while applying PPO-style clipping and KL divergence regularization to ensure stable updates. This results in an updated policy p_θ.

GRPO also makes use of a reference policy $p_{\theta_{\text{ref}}}$ in the KL-divergence penalty, seeking to prevent the updated policy from drifting too far from its original distribution. Here, we express the original GRPO objective in Equation 1 in terms of the prompt-output-reward triples (q, o_i, r_i) to facilitate the extension to the multi-module setting.

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{\{(q,o_i,r_i)\}_{i=1}^{G}}, \text{ where } \theta \text{ indicates the parameters for an LM shared by all groups}$$

$$\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|o_i|}\sum_{t=1}^{|o_i|}\left\{\min\left(\omega_t \hat{A}_{i,t},\ \text{clip}\left(\omega_t,\ 1-\epsilon,\ 1+\epsilon\right)\hat{A}_{i,t}\right) - \beta\mathbb{D}_{\text{KL}}[p_\theta \,\|\, p_{\theta_{\text{ref}}}]\right\} \quad (1)$$

$$\text{where} \quad \omega_t = \frac{p_\theta(o_{i,t} \mid q, o_{i,<t})}{p_{\theta_{\text{old}}}(o_{i,t} \mid q, o_{i,<t})}, \text{ and } \hat{A}_{i,t} \text{ is determined with } r_i \text{ by the supervision format}$$

Each GRPO group is defined as a set of triples $\mathcal{G} = \{(q, o_i, r_i)\}_{i=1}^{G}$, constructed by first sampling a fixed prompt from a distribution of questions $q \sim P(Q)$, and then generating a batch of G completions $\{o_i\}_{i=1}^{G} \sim p_{\theta_{\text{old}}}(O \mid q)$ from the current policy. Finally, a scalar reward r_i for each o_i is computed with a reward function. The term ω_t denotes the importance sampling ratio between the new and old policies for the tth token in a given output. The scalar reward r_i is then normalized within the group to compute an advantage $\hat{A}_i$ in the *outcome supervision* formulation of GRPO, which is applied uniformly across all tokens t in the corresponding completion, as shown in Equation 2.

$$\hat{A}_{i,t} = \tilde{r}_i = \frac{r_i - \text{mean}(\mathcal{R})}{\text{std}(\mathcal{R})}, \quad \mathcal{R} = \{r_i, \text{reward for } o_i\}_{i=1}^{G} \quad (2)$$

LM program formulation An LM program Φ is composed of tools and LM modules orchestrated by the control flow of Φ. Let $\mathcal{M} = \{M_1, \ldots, M_{|\mathcal{M}|}\}$ denote the set of LM modules used therein, each of which communicates via natural language.

Given an input x (e.g., a dictionary with fields such as `question`), executing $\Phi(x)$ yields a distribution over y, ρ pairs where y is the final output and ρ is the trajectory:

$$(y, \rho) \sim \Phi(x), \quad \rho = [\zeta_1, \zeta_2, \ldots, \zeta_{|\rho|}], \quad (3)$$

where trajectory ρ records the sequence of module calls, and trace $\zeta_t = \langle M_t, q_t, o_t \rangle$ captures the module identity as well as the module-level inputs and outputs at step t.[1] During execution, the control flow of Φ orchestrates module invocations, transforming inputs and routing outputs between modules. While the trajectory ρ logs only the LM-level calls, it omits this glue logic. Hence, the final output y is not directly included in ρ, as it may be computed from module outputs in a program-specific manner. We assume that ρ records module calls in execution order, both across different modules and sequentially within each module, while allowing for possible ambiguity in the ordering of parallel calls.

Each module $M \in \mathcal{M}$ is parameterized by a prompt template π_M and LM weights θ_M. During execution at module invocation t, the prompt template π_{M_t} transforms the input q_t into a prompt: $q_t \leftarrow \pi_{M_t}(q_t)$.[2] This prompt is then passed to an LM parameterized by θ_{M_t}, which samples an output $o_t \sim p_{\theta_{M_t}}(\cdot \mid q_t)$, which is returned to the control flow of Φ for subsequent steps. This setup differs substantially from standard multi-turn LM generation settings, where the full interaction history is passed to an LM at each turn (Jin et al. 2025; Zeng et al. 2025; Wang et al. 2025). Instead, the control flow dictates what context is visible to each module by selecting its inputs, enabling more modular and interpretable execution.

This modularity offers several benefits. It allows for privacy-preserving delegation, e.g., a module may call a proprietary LM that should not access previous interactions, as in our `PAPILLON` task, and better context length management, which is particularly important in RAG-style pipelines like `Hover`, where large numbers of retrieved passages may need to be processed independently. This is a core reason why multi-step GRPO formulations wouldn't be suitable for LM programs out-of-the-box and motivates us to explore alternative multi-module formulations. Throughout this paper, we treat LM policy inputs as being defined strictly at the module-level.

[1]Here, t indexes steps in the LM program trajectory ρ, whereas in the GRPO objective in Equation 1, t indexes tokens within an LM output. They are both positions, but are distinct: the former refers to module calls, the latter to positions within a generated sequence.

[2]We distinguish between a *prompt template*, which includes learnable components (e.g., few-shot examples or instructions) and fixed structure (e.g., expected input/output format), and a *prompt*, which is the resulting string.

```
dspy.configure(rm=dspy.ColBERT(..), lm=lm)
program = MultiHop()
                                    Supports local models via Arbor!

prompt_optimizer = dspy.MIPROv2(metric, …)
po_program = prompt_optimizer.compile(program, trainset, …)

weight_optimizer = dspy.GRPO(metric, …)
better_program = weight_optimizer.compile(po_program, trainset, …)

                  Combine online RL with prompt optimization seamlessly!
```

Figure 1: **[todo]** This is WIP

LM program optimization Given a finite training dataset $\mathcal{D} = \{(x, m)\}$ consisting of inputs x and optional metadata m, the goal is to optimize the parameters of an LM program Φ, namely, the prompt templates π_M and LM weights θ_M for each module $M \in \mathcal{M}$, to maximize expected reward over executions. Let $\Pi = [\pi_{M_1}, \ldots, \pi_{M_{|\mathcal{M}|}}]$ and $\Theta = [\theta_{M_1}, \ldots, \theta_{M_{|\mathcal{M}|}}]$ denote $\Phi_{\langle \Pi, \Theta \rangle}$'s module-level prompt templates and LM weights, respectively. The LM program optimization objective is:

$$\max_{\Pi, \Theta} \; \mathbb{E}_{(x,m) \in \mathcal{D}} \left[\mu \left(y, \rho, m \right) \right], \quad \text{where } (y, \rho) \sim \Phi_{\langle \Pi, \Theta \rangle}(x) \tag{4}$$

The reward function $\mu(y, \rho, m)$ scores the execution, typically based on final output correctness using y optionally on intermediate behaviors such as step efficiency or correctness using ρ. The metadata m (e.g., gold answers) is not visible to the program during execution but may be used by μ for evaluation.

3 Applying GRPO to multi-module LM programs

We now present the mmGRPO framework, allowing for a straightforward generalization of GRPO to the multi-module LM program setting. Like GRPO, mmGRPO is a policy gradient method, but adapted to handle the unique structure of LM programs, namely, modularity, lack of shared inputs across rollouts, and variable trajectory lengths. Given a fixed LM program Φ with a set of modules $\mathcal{M}$, where each module $M \in \mathcal{M}$ is parameterized by a learnable prompt template π_M and LM weights θ_M, our goal is to update the weights θ_M to maximize the expected reward over executions as defined in Equation 4, while keeping the prompt templates fixed. Formally, mmGRPO takes as input: (1) a *student* LM program Φ with modules $\mathcal{M}$, (2) a dataset $\mathcal{D}$ of task examples, and (3) a reward function μ over program executions. It returns an updated program Φ with the same prompt templates but improved LM weights, i.e., $\{\pi_{M_i}, \theta^*_{M_i}\}_{i=1}^{|\mathcal{M}|}$.

A complete specification of mmGRPO is given in Appendix A. At its core, mmGRPO extends GRPO with two simple abstractions: (1) a generalized group-forming strategy that operates at the module-level, aligning structurally similar module calls across rollouts; and (2) optional support for sampling trajectories from a list of fixed *teacher* LM programs, enabling flexible combinations of online, off-policy, and on-policy training. When applied to a single-stage program without teacher programs, mmGRPO is behaviorally identical to standard GRPO. We describe each abstraction in the following subsection, and defer the full algorithmic details to Appendix A for brevity.

Algorithm 1 FORMMODULELEVELGROUPS: Create module-level GRPO groups for MMGRPO

Require:

Student program Φ, with modules $M \in \mathcal{M}$
Rollouts $\mathcal{R} = \{(\rho_j, r_j)\}_{j=1}^{R}$, sampled trajectories with trajectory-level rewards
Group size G

1: **function** FORMMODULELEVELGROUPS(Φ, $\mathcal{R}$, G)
2:     `grpo_groups_dict` $\leftarrow$ DEFAULTDICT(list)
3:     **for** each $(\rho, r) \in \mathcal{R}$ **do**
4:         `relative_invocation_orders` $\leftarrow$ DEFAULTDICT(LIST)
5:         **for** each trace $\zeta = (M, q, o) \in \rho$ **do**
6:             Append (q, o, r) to `grpo_groups`$[(M,$ `relative_invocation_orders`$[M])]$
7:             `relative_invocation_orders`$[M]$ $+= 1$
8:     `grpo_groups_dict` $\leftarrow$ PADGROUPS(`grpo_groups`)
9:     `grpo_groups` $\leftarrow$ [SELECTKDIVERSEELEMENTS($\mathcal{G}, G$) $\mid \mathcal{G} \in$ VALUES(`grpo_groups_dict`)]
10:     $\Theta \leftarrow$ [Get M's weights $\theta_M \mid (M,$ `relative_invocation_order`$) \in$ KEYS(`grpo_groups_dict`)]
11:     **return** `grpo_groups`, Θ

Assume DEFAULTDICT, KEYS, and VALUES are provided
Refer to Section 3.1 for descriptions of PADGROUPS and SELECTKDIVERSEELEMENTS

3.1 FORMING MODULE-LEVEL GROUPS

We now describe how MMGRPO constructs GRPO-style groups at the module level for LM programs. For each training example $(x, m) \in \mathcal{D}$, we begin by sampling R rollouts.[3] Each rollout yields a final output y and a trajectory ρ; $(y, \rho) \sim \Phi(x)$, for which we compute a reward $r = \mu(y, \rho, m)$. We then put together a set of tuples $\mathcal{R} = \{(\rho_j, r_j)\}_{j=1}^{R}$, which we will use to construct the module-level GRPO groups. If a rollout fails to complete, e.g., due to the LM call at a particular module producing unparsable output that prevents the program from performing the subsequent steps, we retain the partial trajectory up to and including the failed module call and assign a fallback reward to the full run (i.e., a formatting error penalty).

Once the rollouts are sampled, we construct module-level GRPO groups using the FORMMODULELEVELGROUPS function described in Algorithm 1. Each group is defined as a list of G triples $\{(q_g, o_g, r_g)\}_{g=1}^{G}$, where each element consists of a module-level input prompt q, the corresponding output o, and the final trajectory-level reward r. In practice, one can use $G < R$, the number of rollouts, to leave room for post-hoc adjustments to group size (discussed later in this section).

FORMMODULELEVELGROUPS takes as input the student program Φ, the list of trajectory-reward tuples $\mathcal{R}$, and the desired GRPO group size G (Line 1). It begins by initializing `grpo_groups_dict` as a dictionary mapping keys to lists (Line 2), where each key corresponds to a tuple of a module and its relative invocation index within a trajectory. The function then iterates over each trajectory-reward pair in $\mathcal{R}$ (Line 3), and traces in each trajectory (Line 5). Each trace contributes a triple (q, o, r) consisting of the module-level input, output, and final trajectory reward. This triple is added to the group corresponding to (M, k), where k is the relative invocation index of M in the trajectory (Line 6), where the relative index is incremented after each occurrence (Line 7). To ensure uniform group sizes despite variability in module invocation counts across trajectories, Lines 8 and 9 apply post-processing steps that adjust each group to have exactly G elements, as detailed later in this section. Finally, Line 10 constructs a list of LM weight references, one corresponding to each group, and both this list and the final GRPO groups are returned (Line 11).

As a result, FORMMODULELEVELGROUPS creates GRPO groups by both the module identity and their relative position within the trajectory with respect to the other calls to the same module. Let K_{M_i,ρ_j} denote the number of times module M_i is invoked in trajectory ρ_j for $(\rho_j, r_j) \in \mathcal{R}$; then the total number of GRPO groups formed across all trajectories is $\sum_i \max_j K_{M_i,\rho_j}$, where $M_i \in \mathcal{M}$ for the given `runs`. Each resulting group is a list of module-level (q, o, r) triples, corresponding to structurally aligned invocations of a given module at a specific position in the trajectory. In contrast to standard GRPO, which produces a single group per set of rollouts in single-stage settings, MMGRPO

[3] See Appendix A for details on how N is computed and how the optional use of teacher programs influence this sampling process.

yields a list of groups, one for each module and relative invocation position. To ensure uniform group sizes and handle variation across trajectories, we apply two *post-processing* steps: PADGROUPS and SELECTKDIVERSEELEMENTS, described later in this section.

The MMGRPO loss, shown in Equation 5, is applied independently to each of the constructed module-level GRPO groups, with two key modifications to the original GRPO objective in Equation 1. First, instead of updating a single shared LM with weights θ, each group updates the weights θ_M of the specific module M from which its data is drawn. Second, unlike standard GRPO where all completions share the same prompt, each datapoint in a module-level GRPO group may have a distinct prompt q_i, reflecting differences in upstream context. These changes are highlighted in Equation 5. Each GRPO group here refers to an element in the `grpo_groups` list returned by the FORMMODULELEVELGROUPS function. The corresponding module-level LM weights θ_M are returned in a separate list that is index-aligned with `grpo_groups`.

$$\mathcal{J}_{\text{mmGRPO}}(\theta_M) = \mathbb{E}_{\{(q_i, o_i, r_i)\}_{i=1}^{G}}, \text{ where } \theta_M \text{ indicates the LM weights for module } M$$

$$\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|o_i|}\sum_{t=1}^{|o_i|}\left\{\min\left(\omega_t \hat{A}_{i,t},\ \text{clip}\left(\omega_t,\ 1-\epsilon,\ 1+\epsilon\right)\hat{A}_{i,t}\right) - \beta\mathbb{D}_{\text{KL}}\left[p_{\theta_M} \,\|\, p_{\theta_{M_{\text{ref}}}}\right]\right\} \tag{5}$$

$$\text{where} \quad \omega_t = \frac{p_{\theta_M}(o_{i,t} \mid q_i, o_{i,<t})}{p_{\theta_{M_{\text{old}}}}(o_{i,t} \mid q_i, o_{i,<t})}, \text{ and } \hat{A}_{i,t} \text{ is computed with } r_i \text{ via Equation 2}$$

Handling variably invoked trajectories with PADGROUPS If every module M_i in the student program is invoked the same number of times $K_{M_i,*}$ across all trajectories ρ_j where $(\rho_j, r_j) \in \mathcal{R}$, then each constructed GRPO group will contain exactly R triples prior to the call to Line 8 in Algorithm 1. For example, suppose the LM program consists of two modules, M_1 and M_2, and we collect $R = 3$ trajectories. If in every trajectory, the program calls M_1 exactly twice and M_2 exactly once, then MMGRPO will form three GRPO groups: two for M_1 (corresponding to its first and second calls) and one for M_2. Each of these groups will contain exactly three triples, one from each trajectory, without requiring any padding or truncation. This scenario arises when all executions yield structurally identical trajectories and none encounter parsing or runtime errors.

However, in practice, these conditions may not hold: some modules may be invoked fewer times due to variation in control flow, while others may terminate early due to parsing failures or other runtime errors. In such cases, certain module, module invocation level GRPO groups may contain fewer than N elements. To address this, we apply post-processing strategies to ensure that each group has a uniform size, with a call to the PADGROUPS function (see Algorithm 2 in Appendix A for the full algorithm; key details are summarized here).

The behavior of PADGROUPS is controlled by a `padding_mode` hyperparameter, which supports two values: `truncate` and `fill`. Under the `truncate` strategy, we discard all GRPO groups for module M_i whose invocation index exceeds $\min_j K_{M_i,\rho_j}$, ensuring that only groups with complete representation across all trajectories are retained. Under the `truncate` strategy, we discard all GRPO groups for a module M_i whose invocation index exceeds $\min_j K_{M_i,\rho_j}$, ensuring that only those invocation positions represented in every trajectory are retained. We use the `fill` setting for our experiments.

Ensuring diversity in groups with SELECTKDIVERSEELEMENTS After standardizing group sizes across trajectories, we further adjust each group to ensure it contains exactly G elements, the target GRPO group size. Rather than sampling elements uniformly at random, we invoke a SELECTKDIVERSEELEMENTS function, which selects (or duplicates) elements to form a group of size G while maximizing diversity within the group. This function handles both down-sampling (when the group has more than G elements) and up-sampling (when it has fewer), favoring selections that increase reward variance in the sampled prompt-output pairs. Contemporaneously, Xu et al. (2025) propose a similar variance-based selection strategy, demonstrating that promoting diversity in GRPO groups improves held-out generalization.

3.2 Allowing sampling with teacher programs

In addition to the student program, mmGRPO accepts a list of optional *teacher programs*, which are used to generate the set of trajectories that populate the `runs` list. At each GRPO step, rather than sampling all rollouts from the student program alone, mmGRPO samples trajectories from a specified mixture of teacher programs, this list must include the student itself. All teacher programs share the same structural interface, meaning they operate over the same LM program and module-level input/output fields, but may differ in their module-level prompt-templates (e.g., alternative instructions or few-shot examples) or LM weights (e.g., larger LMs). These variations, described in more detail in Appendix A, enable mmGRPO framework to allow performing training that is online but partially off-policy, providing greater flexibility in guiding learning using curated or higher-performing policies. We do not separately pass teacher programs to mmGRPO for our experiments.

4 Composing Online RL with Prompt Optimization via BetterTogether

The BetterTogether approach from prior work (Soylu et al., 2024) demonstrated that combining prompt optimization (PO) with weight optimization yields stronger results than using either technique alone, specifically in the context of offline RL via rejection fine-tuning on outcome-filtered trajectories. Rather than applying weight optimization directly to an unmodified program, the authors first optimize the program's prompt templates and then apply weight optimization on the resulting prompt-optimized program.[4] We extend this approach to the online RL setting using mmGRPO, and combine it with a state-of-the-art prompt optimizer, MIPROv2 (Opsahl-Ong et al., 2024), from the DSPy library (Khattab et al., 2024).

5 Experiments

5.1 LMs and datasets

The LM programs for each of these datasets along with examples and their trajectories are shared in Appendix B. We use the LM program implementations shared by Tan et al. (2025) as a starting point for all tasks, but make modifications for `Hover`. For more information on the LMs and datasets used along with their license information, refer to Appendix C.

LMs We run all of our experiments on two open-source LMs: `llama3.1-8b-instruct` (Grattafiori et al., 2024) and `qwen3-8b` (Yang et al., 2025). Although the formulation for mmGRPO allows for different LMs to be used for the different modules of a program, we use the same underlying LM weights for each module for simplicity, training in a multi-task manner.

Classification `Banking77` is an intent classification benchmark involving 13,083 labeled customer service queries from the banking domain Casanueva et al. (2020). The task is to assign each user query to one of 77 intent classes. We implement a simple program for this task using a single Chain-of-Thought (CoT) module Wei et al. (2023), which first produces a reasoning trace before predicting the intent label. For evaluation we compute the exact match between the ground-truth label and the generated label. Since the program we have for `Banking77` has is a single module, running the mmGRPO algorithm on it is the same as the standard GRPO setup. For training and evaluation, we randomly sample 250 training examples and 500 for development.

Privacy-conscious delegation The Private User Prompt Annotations (PUPA) benchmark constructed by Siyan et al. (2024) focuses on privacy-preserving question answering, where the goal is to respond to user queries without exposing private information to external APIs. We use PAPILLON, also from Siyan et al. (2024), a delegation pipeline consisting of two modules that generates redacted versions of private queries to be sent to an un-trusted external model, while retaining the original for local processing. The program first calls a CoT module that redacts the query to remove private

[4] Soylu et al. (2024) also experiment with alternative compositions, such as running prompt optimization after weight tuning, but in our work, we focus on the former: applying mmGRPO to a prompt-optimized program.

information, then queries an external large LM to answer the redacted version of the query, finally calling the second CoT module that uses the returned answer from the large LM to create a private final answer for the query. We utilize `openai/gpt-4.1-mini-2025-04-14` (OpenAI, 2025) as the large LM. As described in Siyan et al. (2024), the evaluation metric is a composite score which takes into account the correctness of the response and the amount of private information that was leaked, both of which are judged by the same large LM. We evaluate this setup using 111 training examples and 221 for development.

Multi-hop claim verification `Hover` (HOppy VERification, Jiang et al., 2020) is a claim verification benchmark where the task is to extract facts from multiple relevant Wikipedia articles and deciding whether a given claim is supported. The claims in `Hover` are *multi-hop* in that they require multi-hop reasoning by connecting information found in different articles. The original dataset has $18,171$ train and 4000 development and test examples derived from the examples in the HotPotQA dataset (Yang et al., 2018). Our program for `Hover` consists of 2 modules, a query generation module and a fact summarization module, called iteratively over 4 hops, along with a ColBERTv2 (Santhanam et al., 2022) retriever indexed on the short Wikipedia snippets provided with the HotPotQA dataset, shared with `Hover`. We refer to the particular 4-hop variant Hover program we use with $\texttt{Hover}_{\texttt{4-HOP}}$, in order to differentiate it from the one provided in Tan et al. (2024). The program returns up to 100 passages at the end, and the final metric evaluates whether the gold passages are found within the returned passages using Recall@100. We build our splits from the original train split, randomly sampling 500 examples each for our train and development splits; while ensuring that we don't sample any two examples derived from the same HotPotQA question.

5.2 Baseline and method details

We evaluate each of our LM and task pairs with vanilla Chain-of-Thought (CoT) and a prompt-optimizer, to serve as baselines. We demonstrate our mmGRPO optimizer in two flavors: mmGRPO, and BetterTogether mmGRPO. While each method assumes access to a program-level evaluation metric, none relies on an external oracle dataset. Instead, we generate training data dynamically by running the program itself and bootstrapping from model outputs and associated program-level metrics. We use the DSPy framework (Khattab et al., 2024) to run our baseline experiments and develop our new mmGRPO optimizers. We use DSPy's RL training library, Arbor (Ziems et al., 2025), which draws inspiration from the Verifiers library Brown (2025).

Inference We use the `vLLM` (Kwon et al., 2023) engine for sampling with max context length of 32,768 tokens for inference. For `qwen3-8b`, we use $\text{sampling\_temperature} = 0.6$, $\text{top\_p} = 0.95$ and $\text{top\_k} = 20$ following the parameters used for its instruction training as noted in Yang et al. (2025). For `llama3.1-8b-instruct`, we use $\text{sampling\_temperature} = 0.6$ and $\text{top\_p} = 0.9$ following the official model card's generation configuration in HuggingFace (MetaAI, 2024).

Vanilla CoT We adopt the Chain-of-Thought (CoT) prompting method introduced by Wei et al. (2023), where each module's prompt instructs the language model to first generate a *reasoning* field before producing its final answer. Unless stated otherwise, both the prompt-optimization and mmGRPO methods described below begin training from this base CoT prompt. We refer to this initial prompt configuration as the "Vanilla CoT" program.

MIPROv2 We use the state-of-the-art prompt optimizer Multiprompt Instruction PRoposal Optimizer Version 2 (MIPROv2, Opsahl-Ong et al. 2024) as our prompt-optimized baseline. For our experiments, we use the `auto=medium` setting, which uses `12` trials; `12` few-shot and `6` instruction candidates, and automatically uses a 80% of the train set for validation. We refer to the program we optimize using MIPROv2 with these settings as the prompt-optimized program and re-use it for the BetterTogether strategy below.

mmGRPO We train `llama3.1-8b-instruct` `qwen3-8b` using the HuggingFace `GRPOTrainer`, each with a maximum context length of 8192 tokens. Training is performed with a temperature of 0.6, a learning rate of 1×10^{-5}, gradient accumulation steps of 20, with per device train batch size of 1. We use $\beta = 0.01$ and gradient norm clipping of 0.1 for `qwen3-8b`; and $\beta = 0.04$ and gradient norm clipping of 0.5 for `llama3.1-8b-instruct`.

We run mmGRPO for 750 steps, using 4 training examples per step. At each step, we randomly draw 4 examples from the training dataset. For each example, we generate 12 rollouts, which are then grouped into module-level GRPO groups using the procedure in Algorithm 1. We use a train context length of $8,192$ tokens, which is used to filter any trajectory with a module level prompt and completion longer than this. We apply Low-Rank Adaptation (LoRA, Hu et al. 2021) with rank $\mathrm{r} = 16$, $\mathrm{lora\_alpha} = 64$, $\mathrm{lora\_dropout} = 0.05$, targeting the projection modules $[\mathrm{q}, \mathrm{k}, \mathrm{v}, \mathrm{o}, \mathrm{up}, \mathrm{down}, \mathrm{gate}]$. We run all of our mmGRPO experiments below using these same settings. Pseudocode of the mmGRPO algorithm can be found in Algorithm 2.

mmGRPO with BetterTogether We further experiment with a setting where we combine prompt optimization with the weight optimization of mmGRPO following the BetterTogether algorithm introduced by Soylu et al. (2024). Specifically, instead of directly optimizing the weights used in an LM program, we first use prompt optimization to find high quality prompts to be used by the LM program. The prompts are then kept fixed in the LM program and the program weights are then optimized with mmGRPO. We use mmGRPO(MIPROv2) for short to represent this setup.

5.3 Main results

Our main experimental results are shared in Table 1, evaluated on the dev set and averaged over 3 seeds.

mmGRPO and BetterTogether(PO, mmGRPO) consistently improve over their respective baselines. We can see that the mmGRPO row is consistently higher than the "Vanilla CoT" row, 7.3% on average. Similarly, BetterTogether(PO, mmGRPO) shows consistent gains over the "MIPROv2 (PO)" row. These show that mmGRPO is effective at finding better policies for the provided program across all LM, task pairs.

PO is competitive with lower computational budgets. When averaged across all tasks and models, MIPROv2 alone only improved upon baseline CoT by 5.4% compared to mmGRPO's 7.3% improvement. However, MIPROv2 achieved these results significantly faster while using fewer GPU-hours. On average, our vanilla mmGRPO experiments took 18.7 hours using 2 `H100` GPUs whereas MIPROv2 took only 1.4 hours on average and only required 1 `H100` GPU. These results indicate that PO approaches like MIPROv2 are likely much more feasible for settings which have lower computation budgets.

BetterTogether(PO, mmGRPO) performs the best in 5 out of the 6 LM, task pairs. BetterTogether(PO, mmGRPO) approach improves over the Vanilla CoT baseline by 10.86% and MIPROv2 baseline by 5.14%. This shows the value of high-quality rollouts at the start of mmGRPO training, as performing PO generates stronger rollouts, leading to a more robust training signal early in the training runs.

6 Related work

LM optimization There are two broad approaches for optimizing the prompts and weights of LMs.

Prompt optimization A variety of recent works have explored ways to optimize natural language prompts over time in an effort to maximize downstream LM performance such as prompt evolution approaches (Fernando et al., 2024), approaches that use gradients to optimize the prompt (Shin et al., 2020; Wen et al., 2023), ranking based approaches (Gao et al., 2021), approaches that prompt other LMs (Yang et al., 2024; Zhou et al., 2023; Pryzant et al., 2023), and RL-based prompt optimization approaches (Deng et al., 2022; Zhang et al., 2023; Hao et al., 2023).

Weight optimization Proximal Policy Optimization (PPO) has been widely used for post-training language models with reinforcement learning, particularly when aligning language models with human preferences or feedback (Schulman et al., 2017; Ouyang et al., 2022). In the context of Reinforcement Learning from Human Feedback (RLHF) introduced by Ouyang et al. (2022), PPO uses an actor-critic approach where a reward model is trained to approximate human preferences and a policy model is optimized based on predicted rewards from the reward model. However, PPO is computationally intensive, as it requires training and running both the policy model and the reward model during optimization.

Recently, Direct Preference Optimization (DPO) emerged as a simpler, fully supervised alternative that avoids explicit reward modeling and instead learns from contrastive preference pairs (Rafailov et al., 2023). Similarly, Group Relative Policy Optimization (GRPO) offers an efficient alternative to PPO by avoiding the need for a value model and instead relying on estimated advantages through relative rewards within a group of rollouts (Shao et al., 2024).

LM Program Optimization Existing work has explored optimizing LM programs with prompt optimizers including those that focus primarily on rejection sampling (Khattab et al., 2024) and others that extend this to use Bayesian optimization for selecting the instruction-demonstration candidates that are most promising (Opsahl-Ong et al., 2024).

Additional work (Soylu et al., 2024) has explored combining weight optimizer along with prompt optimizers for additional benefit, but in the context of offline RL.

However, as discussed in Section 2, adapting some techniques to LM Programs is often non-trivial. While PPO is straightforward to apply to language model (LM) programs, its high computational cost is a significant limitation. In contrast, vanilla GRPO is significantly more efficient, but does not directly map to LM programs composed of multiple modules and requires substantial modification to do so—an issue we address in this work.

7 Conclusion

We introduce mmGRPO, a novel extension of GRPO that enables online weight optimization for multi-module LM programs by propagating final rewards backward across disjoint modules. Our experiments demonstrate that mmGRPO consistently outperforms standard baselines across tasks and models, validating its effectiveness in navigating the challenging credit assignment problem without requiring intermediate supervision. We further show that combining mmGRPO with state-of-the-art prompt optimization methods via BetterTogether yields the strongest overall performance in the majority of settings, revealing that complementary relationship between weight and prompt optimization holds for online RL methods.

8 Limitations

While our experiments show the promise of multi-module RL formulations, this work is not without limitations. Our experiments use language models with 8-billion parameters, which may not reflect the performance of mmGRPO when applied to larger models. We also rely on LoRA for fine-tuning, which, while efficient, may limit training performance relative to full fine-tuning. Finally, we evaluate only one mmGRPO implementation despite many possible alternative formulations.

Acknowledgments

D.S. thanks the members of the StanfordNLP group for their support and feedback. The authors also thank Zoom, Inc. for providing access to their compute clusters for running the later batch of experiments.

References

William Brown. Verifiers: Reinforcement learning with llms in verifiable environments. https://github.com/willccbb/verifiers, 2025.

Iñigo Casanueva, Tadas Temčinas, Daniela Gerz, Matthew Henderson, and Ivan Vulić. Efficient intent detection with dual sentence encoders. In *Proceedings of the 2nd Workshop on Natural Language Processing for Conversational AI*, pp. 38–45, 2020.

Mingkai Deng, Jianyu Wang, Cheng-Ping Hsieh, Yihan Wang, Han Guo, Tianmin Shu, Meng Song, Eric Xing, and Zhiting Hu. RLPrompt: Optimizing discrete text prompts with reinforcement learning. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (eds.), *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 3369–3391, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.222. URL https://aclanthology.org/2022.emnlp-main.222/.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: self-referential self-improvement via prompt evolution. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 13481–13544, 2024.

Tianyu Gao, Adam Fisch, and Danqi Chen. Making pre-trained language models better few-shot learners. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 3816–3830, 2021.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Yaru Hao, Zewen Chi, Li Dong, and Furu Wei. Optimizing prompts for text-to-image generation. *Advances in Neural Information Processing Systems*, 36:66923–66939, 2023.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. URL https://arxiv.org/abs/2106.09685.

Yichen Jiang, Shikha Bordia, Zheng Zhong, Charles Dognin, Maneesh Singh, and Mohit Bansal. HoVer: A dataset for many-hop fact extraction and claim verification. In Trevor Cohn, Yulan He, and Yang Liu (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 3441–3460, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.309. URL https://aclanthology.org/2020.findings-emnlp.309/.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning, 2025. URL https://arxiv.org/abs/2503.09516.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into self-improving pipelines. In *ICLR*, 2024.

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 ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

MetaAI. Meta llama 3.1-8b-instruct: Generation configuration. https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct/blob/main/generation_config.json, 2024. Accessed: 2025-07-29.

OpenAI. Gpt-4.1-mini. https://platform.openai.com/docs/models/gpt-4.1-mini, 2025. Accessed: 2025-07-17.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 9340–9366, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.525. URL https://aclanthology.org/2024.emnlp-main.525/.

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

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 7957–7968, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.494. URL https://aclanthology.org/2023.emnlp-main.494/.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741, 2023.

Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. Colbertv2: Effective and efficient retrieval via lightweight late interaction, 2022. URL https://arxiv.org/abs/2112.01488.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (eds.), *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 4222–4235, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.346. URL https://aclanthology.org/2020.emnlp-main.346/.

Li Siyan, Vethavikashini Chithrra Raghuram, Omar Khattab, Julia Hirschberg, and Zhou Yu. Papillon: Privacy preservation from internet-based and local language model ensembles. *arXiv preprint arXiv:2410.17127*, 2024.

Dilara Soylu, Christopher Potts, and Omar Khattab. Fine-tuning and prompt optimization: Two great steps that work better together. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 10696–10710, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.597. URL https://aclanthology.org/2024.emnlp-main.597/.

Shangyin Tan, Lakshya A Agrawal, Arnav Singhvi, Liheng Lai, Michael J Ryan, Dan Klein, Omar Khattab, Koushik Sen, and Matei Zaharia. Langprobe: a language programs benchmark, 2025. URL https://arxiv.org/abs/2502.20315.

Sijun Tan, Siyuan Zhuang, Kyle Montgomery, William Y. Tang, Alejandro Cuadron, Chenguang Wang, Raluca Ada Popa, and Ion Stoica. Judgebench: A benchmark for evaluating llm-based judges, 2024. URL https://arxiv.org/abs/2410.12784.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, Eli Gottlieb, Yiping Lu, Kyunghyun Cho, Jiajun Wu, Li Fei-Fei, Lijuan Wang, Yejin Choi, and Manling Li. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning, 2025. URL https://arxiv.org/abs/2504.20073.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903.

Yuxin Wen, Neel Jain, John Kirchenbauer, Micah Goldblum, Jonas Geiping, and Tom Goldstein. Hard prompts made easy: Gradient-based discrete optimization for prompt tuning and discovery. *Advances in Neural Information Processing Systems*, 36:51008–51025, 2023.

Yixuan Even Xu, Yash Savani, Fei Fang, and Zico Kolter. Not all rollouts are useful: Down-sampling rollouts in llm reinforcement learning, 2025. URL https://arxiv.org/abs/2504.13818.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=Bb4VGOWELI.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259/.

Siliang Zeng, Quan Wei, William Brown, Oana Frunza, Yuriy Nevmyvaka, and Mingyi Hong. Reinforcing multi-turn reasoning in llm agents via turn-level credit assignment, 2025. URL https://arxiv.org/abs/2505.11821.

Tianjun Zhang, Xuezhi Wang, Denny Zhou, Dale Schuurmans, and Joseph E. Gonzalez. TEMPERA: Test-time prompt editing via reinforcement learning. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=gSHyqBijPFO.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=92gvk82DE-.

Noah Ziems, Lakshya A Agrawal, Dilara Soylu, Liheng Lai, Isaac Miller, Chen Qian, Meng Jiang, and Omar Khattab. Arbor: Open source language model post training. https://github.com/Ziems/arbor, 2025.

APPENDIX

A MMGRPO ALGORITHM

Overview of MMGRPO (Algorithm 2) The MMGRPO algorithm extends GRPO to the multi-module setting by improving the LM weights of each module in a program through module-level policy gradients. Two core abstractions distinguish MMGRPO in Algorithm 2: (1) the ability to sample trajectories from multiple teacher programs, and (2) the construction of module-level GRPO groups based on relative invocation order. These components are highlighted in the algorithm, while the remaining steps follow standard GRPO procedure and are included for completeness.

Algorithm 2 MMGRPO: GRPO for multi-module LM programs

Require:
- **Student program** Φ, with modules $M \in \mathcal{M}$
- **Training set** $\mathcal{D}$
- **Metric** μ
- **Teacher programs** $\mathcal{T}$ (optional), defaults to a list containing only the student program if left empty
- **Data collection hyper-parameters** Ψ_{data} (optional):
  - number of training steps N_{steps}
  - batch size B
  - rollout configuration $K : \mathcal{T} \to \mathbb{N}$, specifying the number of rollouts per example for each teacher
- **Model training hyper-parameters** Ψ_{train} (optional): learning rate η, weight decay λ, and others
- **Shared hyper-parameters** Ψ_{shared} (optional): group size G

1: **function** MMGRPO(Φ, $\mathcal{D}$, μ, $\mathcal{T}$, Ψ_{data}, Ψ_{train}, Ψ_{shared})
2: **for** step $= 1$ to N_{steps} **do**
3:   $\mathcal{B} =$ SAMPLEBATCH$(\mathcal{D}, B)$
4:   **for** $(x, m) \in \mathcal{B}$ **do**
5:    $\mathcal{R} \leftarrow$ SAMPLETEACHERROLLOUTS$(\mathcal{T}, K, x, m)$
6:    `grpo_groups`, $\Theta \leftarrow$ FORMMODULELEVELGROUPS$(\Phi, \mathcal{R}, G)$
7:    **for** each group $\mathcal{G} \in$ `grpo_groups` and corresponding module LM weights $\theta_M \in \Theta$ **do**
8:     Update θ_M via the GRPO objective in Equation 5 using hyper-parameters $\Psi_{\text{train}} \cup \Psi_{\text{shared}}$
9: **return** Φ with the same prompt-templates but improved LM weights, i.e., $\{\pi_{M_i}, \theta^*_{M_i}\}_{i=1}^{|\mathcal{M}|}$
10:
11: **function** SAMPLETEACHERROLLOUTS($\mathcal{T}$, K, x, m)
12: $\mathcal{R} \leftarrow \emptyset$
13: **for** each teacher program $\Phi^{(t)} \in \mathcal{T}$ **do**
14:   `num_samples` $\leftarrow K[\Phi^{(t)}]$
15:   **for** $k = 1$ to `num_samples` **do**
16:    $(y, \rho) \sim \Phi^{(t)}(x)$
17:    $r \leftarrow \mu(y, \rho, m)$
18:    $\mathcal{R} \leftarrow \mathcal{R} \cup \{(\rho, r)\}$
19: **return** $\mathcal{R}$

Assume SAMPLEBATCH is provided
Refer to Algorithm 1 for FORMMODULELEVELGROUPS

MMGRPO takes as input a student program Φ, a training dataset $\mathcal{D}$, a reward metric μ, an optional set of teacher programs $\mathcal{T}$, and optional hyper-parameters (Line 1). If unspecified, the set of teacher programs $\mathcal{T}$ defaults to a singleton set containing only the student program. At each training step (Line 2), the algorithm samples a batch $\mathcal{B}$ of examples from the training dataset $\mathcal{D}$ using the configured batch size B (Line 3). For each example $(x, m) \in \mathcal{B}$ (Line 4), the algorithm collects rollouts from the teacher programs via the SAMPLETEACHERROLLOUTS function (Line 5), which returns a set of trajectory-reward tuples. These rollouts are passed to FORMMODULELEVELGROUPS from Algorithm 1 (Line 6), which constructs module-level GRPO groups and returns them along with the corresponding references to the module-level LM weights θ_M to be updated. The algorithm then iterates over each group and its associated LM weights (Line 7), and applies the GRPO loss (as defined in Equation 5) independently to each group (Line 8), using the specified training hyper-parameters. After N_{steps} iterations, the algorithm returns the updated student program Φ, preserving its original prompt templates while incorporating improved LM weights (Line 9).

The SAMPLETEACHERROLLOUTS function samples trajectories from each teacher program in $\mathcal{T}$, using a rollout configuration K that specifies the number of rollouts to generate per teacher. This per-teacher control enables flexible data mixtures across programs. For each rollout, the function extracts the final output y, trajectory ρ, and computes the reward r, collecting the resulting (ρ, r) pairs into the rollout set $\mathcal{R}$ returned for training.[5]

B TASK DETAILS

The DSPy implementations for the LM programs for `Banking77`, `PAPILLON`, and `Hover` are presented alongside example datapoints in their respective sections. Code snippets assume respective DSPy imports are available. Programs edited for presentation.

B.1 `Banking77`

The following snippet presents an example datapoint from `Banking77`. The task is to predict the appropriate label given a user query.

Example datapoint for `Banking77`

text: I have been waiting over a week. Is the card still coming?

label: card_arrival

The DSPy program we use for `Banking77` is shared in Python Snippet 1.

```python
Banking77 = dspy.ChainOfThought(f"text -> label: Literal{CLASSES}"
    )
```

Python Snippet 1: DSPy program for Banking77.

B.2 `PAPILLON`

The following snippet shows an example datapoint from `PAPILLON`. The task is to generate the target response using a proprietary language model, while ensuring that no personally identifiable information (PII) is leaked to it.

Example datapoint for `PAPILLON`

user_query: Send a friendly e-mail opening to Sara telling her thank you for calling me back so quickly, taking the time to discuss with me FRC's IPAEP, and for your immediate willingness to make reasonable accommodations to help break down the barriers caused by my disabilities so that I can have equal access to your program. Your kindness and compassion is appreciated.

target_response: Dear Sara, I hope this email finds you well. I just wanted to take a moment to express my sincere gratitude for your prompt response to my call and the time you took to discuss FRC's IPAEP with me. I truly appreciate your immediate willingness to make reasonable accommodations to ensure that I have equal access to your program and to help break down the barriers caused by my disabilities...

pii_str: sara||ipaep||frc

The DSPy program we use for `PAPILLON` is shared in Python Snippet 2.

```python
class CraftRedactedRequest(dspy.Signature):
    """
```

[5]When using teacher programs to sample trajectories, the modules M recorded in the traces reflect those of the teacher rather than the student program. In practice, however, we ensure that the module keys used to form module-level GRPO groups correspond to the student program's modules for each respective teacher module, since we assume that student and teachers programs share the "same structure".

```
    Given a private user query, create a privacy-preserving
        request for a powerful external LLM.
    The LLM may assist without learning private information about
        the user.
    """

    user_query = dspy.InputField()
    llm_request = dspy.OutputField()


class RespondToQuery(dspy.Signature):
    """
    Respond to a user query.
    For inspiration, we found a potentially related request to a
        powerful external LLM and its response.
    """

    related_llm_request = dspy.InputField()
    related_llm_response = dspy.InputField(desc="information from
        a powerful LLM responding to a related request")
    user_query = dspy.InputField(desc="the user's request you need
         to fulfill")
    response = dspy.OutputField(desc="your final response to the
        user's request")


class PAPILLON(dspy.Module):
    def __init__(self, untrusted_model):
        self.craft_redacted_request = dspy.ChainOfThought(
            CraftRedactedRequest)
        self.respond_to_query = dspy.Predict(RespondToQuery)
        self.untrusted_model = untrusted_model

    def forward(self, user_query):
        llm_request = self.craft_redacted_request(user_query=
            user_query).llm_request
        llm_response = self.untrusted_model(llm_request)[0]
        response = self.respond_to_query(
            related_llm_request=llm_request, related_llm_response=
                llm_response, user_query=user_query
        ).response

        return dspy.Prediction(llm_request=llm_request,
            llm_response=llm_response, response=response)
```

Python Snippet 2: DSPy program for Papillon.

B.3 Hover

The following snippet shows an example datapoint from `Hover`. The task is to retrieve all gold Wikipedia titles that support the given claim.

Example datapoint for `Hover`

claim: This director is known for his work on Miss Potter. The Academy of Motion Picture Arts and Sciences presents the award in which he was nominated for his work in "Babe".

titles: ['Miss Potter', 'Chris Noonan', 'Academy Award for Best Director']

The DSPy program we use for `PAPILLON` is shared in Python Snippet 3.

```
# # Assume that a function called deduplicate is defined
```

```python

class GenerateThreeQueries(dspy.Signature):
    """
    Given a claim and some key facts, generate up to 3 followup search query to find the next most essential clue towards verifying or refuting the claim. If you think fewer queries are sufficient, generate None for the search query outputs you don't need. The goal ultimately is to find all documents implicated by the claim.
    """
    claim = dspy.InputField()
    key_facts = dspy.InputField()
    search_query1 = dspy.OutputField()
    search_query2 = dspy.OutputField()
    search_query3 = dspy.OutputField()


class AppendNotes(dspy.Signature):
    """
    Given a claim, some key facts, and new search results, identify any new learnings from the new search results, which will extend the key facts known so far about the whether the claim is true or false. The goal is to ultimately collect all facts that would help us find all documents implicated by the claim.
    """
    claim = dspy.InputField()
    key_facts = dspy.InputField()
    new_search_results = dspy.InputField()
    new_key_facts = dspy.OutputField()


class Hover(dspy.Module):
    def __init__(
            self,
            num_hops=4,
            k_per_search_query=10,
            k_per_search_query_last_hop=30,
            num_total_passages=100,
        ):
        # Value is fixed to simplify signature construction in presented snippet
        self.num_search_queries_per_hop = 3

        self.num_hops = num_hops
        self.k_per_search_query = k_per_search_query
        self.k_per_search_query_last_hop = k_per_search_query_last_hop
        self.num_total_passages = num_total_passages

        self.rm = dspy.ColBERTv2()
        self.generate_query = dspy.ChainOfThought(GenerateThreeQueries)
        self.append_notes = dspy.ChainOfThought(AppendNotes)

    def forward(self, claim: str) -> list[str]:
        key_facts = []
        committed_docs = []

        for hop_ind in range(self.num_hops):
            is_last_hop = hop_ind == self.num_hops - 1
            is_first_hop = hop_ind == 0
            hop_k = self.k_per_search_query_last_hop if is_last_hop else self.k_per_search_query
```

```python
            num_docs_to_keep = (self.num_total_passages - len(committed_docs)) if is_last_hop else self.k_per_search_query

            if is_first_hop:
                search_queries = [claim]
            else:
                pred = self.generate_query(claim=claim, key_facts=key_facts)
                search_queries = [pred.search_query1, pred.search_query2, pred.search_query3]
            search_queries = deduplicate(search_queries)

            search_results = [r for q in search_queries for r in search_raw(q, k=hop_k, rm=self.rm)]
            search_results = sorted(search_results, key=lambda r: r["score"], reverse=True)

            unique_docs = []
            for result in search_results:
                if result["long_text"] not in unique_docs:
                    unique_docs.append(result["long_text"])
            unique_docs = unique_docs[:num_docs_to_keep]
            committed_docs.extend(unique_docs)

            if not is_last_hop:
                pred = self.append_notes(claim=claim, key_facts=key_facts, new_search_results=unique_docs)
                key_facts.append(pred.new_key_facts)

        return dspy.Prediction(key_facts=key_facts, retrieved_docs=committed_docs)
```

Python Snippet 3: DSPy program for HoVer.

C Asset Information

The license information for the models and datasets we used are shared below. All models and datasets are access via HuggingFace.

qwen3-8b is shared with `Apache License 2.0`, accessed via the model ID `Qwen/Qwen3-8B`

llama3.1-8b-instruct is shared with `Meta Llama 3 Community License` at https://llama.meta.com/llama3/license/, accessed via the model ID `meta-llama/Meta-Llama-3.1-8B-Instruct`

Banking77 is shared with `CC BY 4.0`

Hover is shared with `CC BY 4.0`

PAPILLON is shared with `MIT License`