Optimizing Instructions and Demonstrations for Multi-Stage Language Model Programs

Krista Opsahl-Ong[1*], Michael J Ryan[1*], Josh Purtell[2], David Broman[3], Christopher Potts[1], Matei Zaharia[4], Omar Khattab[1]

[1]Stanford University, [2]Basis, [3]KTH Royal Institute of Technology [4]UC Berkeley

Abstract

Language Model Programs, i.e. sophisticated pipelines of modular language model (LM) calls, are increasingly advancing NLP tasks. However, building these pipelines requires crafting prompts that are jointly effective for all modules. We study prompt optimization for LM programs, i.e. how to update these prompts to maximize a downstream metric without access to module-level labels or gradients. To make this tractable, we factorize our problem into optimizing the free-form instructions and few-shot demonstrations of every module and introduce several strategies to craft task-grounded instructions and navigate credit assignment across modules. Our strategies include (i) program-and-data-aware techniques for proposing effective instructions, (ii) a stochastic mini-batch evaluation function for learning a surrogate model of our objective, and (iii) a meta-optimization procedure in which we refine how LMs construct proposals over time. Using these insights we develop MIPRO, a novel optimizer that outperforms baselines on five of seven diverse LM programs using a best-in-class open-source model (Llama3-8B), by as much as 13% accuracy. We have released our new optimizers and benchmark in DSPy at `http://dspy.ai`.

1 Introduction

Solving complex tasks with Language Models (LMs) often requires applying sophisticated prompting techniques (Wei et al., 2022; Chen et al., 2022) and chaining them together into multi-stage pipelines (Wu et al., 2022; Dohan et al., 2022; Khattab et al., 2022; Beurer-Kellner et al., 2023; Yao et al., 2023; Schlag et al., 2023). Such *Language Model (LM) Programs* continue to advance NLP tasks (Pourreza and Rafiei, 2023; Khattab et al., 2024; Ridnik et al., 2024) through systematic composition (Khattab et al., 2021; Creswell and Shanahan, 2022; Pan et al., 2024) and tool use (Qin et al., 2023). However, LM programs today are commonly designed via "prompt engineering": crafting prompts via manual trial and error to coerce a specific LM to operate each step in a specific pipeline. Recent work such as APE (Zhou et al., 2023), OPRO (Yang et al., 2024), and EvoPrompt (Guo et al., 2024) presents *prompt optimizers*, i.e., algorithms that search over strings to identify high-performing prompts. Unfortunately, the majority of this work does not directly apply to multi-stage LM programs in which we lack gold labels or evaluation metrics for the individual LM calls. Khattab et al. (2024) study how to express *arbitrary LM pipelines* such that the instructions, demonstrations (input/output examples), and LM weights of each LM call are treated as parameters that can be optimized toward any metric. While the authors present optimizers that can create demonstrations of multi-stage pipelines and use them to optimize prompts, weights, or even both together

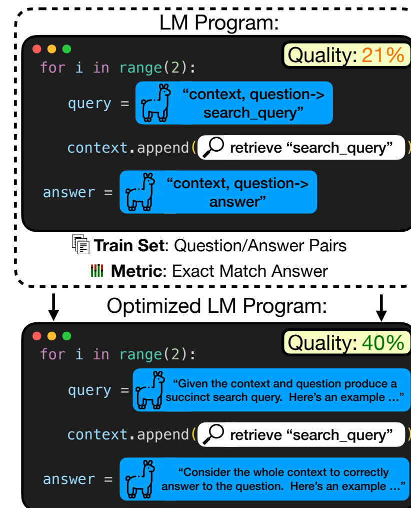

Figure 1: An example of the optimization problem we explore, shown for a multi-hop retrieval LM program. Given some question–answer pairs and a metric, the optimizer proposes new instructions and bootstraps new demonstrations (not pictured) for each stage.

*Equal contribution.

Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pages 9340–9366
November 12-16, 2024 ©2024 Association for Computational Linguistics

(Soylu et al., 2024), their proposed optimizers cannot tune instructions for multi-prompt pipelines.

We seek to efficiently optimize prompts in arbitrary LM programs, especially those with multiple stages (Figure 1) and explore approaches that hold under weak assumptions, consistent with the abstractions from the DSPy programming model (Khattab et al., 2024). In particular, we assume no access to LM weights, log-probabilites, or handwritten metrics or labels for intermediate stages in a chain of LM calls. We require only the LM program itself, a metric to optimize, and a training set of inputs (and, depending on the metric, final outputs).

We formally define the problem of prompt optimization for LM programs and outline the design space by identifying two key challenges. First, the *proposal challenge*: the space of possible prompts is intractably large, and this is exacerbated as the number of modules increase. Proposing a few high-quality instructions is thus essential. Second, the *credit assignment challenge*: our problem requires jointly optimizing over many distinct variables that parameterize the prompts of all modules. To allocate search effort, we must infer the impact of our configurations for each variable effectively.

We define several strategies to tackle each of these challenges and systematically explore the tradeoffs they present. We find that optimizing bootstrapped few-shot examples is often essential for realizing the greatest performance gains, but that optimizing instructions becomes more essential for tasks with conditional rules. We also find that optimizing both instructions and few-shot examples together generally leads to the best results.

We make three contributions. First, we present a formalization of the problem of optimizing language model programs (§2) and propose an algorithm design space with three strategies to address the challenge of prompt proposal and three strategies to resolve the issue of credit assignment (§3). Second, we release a benchmark suite for LM program optimizers spanning seven tasks (§5). Third, we construct and evaluate a rich subset of possible algorithms for prompt optimization (§4). Highlighted amongst these algorithms is MIPRO (Multi-prompt Instruction PRoposal Optimizer) which outperforms baseline optimizers on five of seven tasks in our benchmark, by as much as 13% accuracy improvement. Using our algorithms, we derive five key lessons for practitioners looking to optimize LM programs (§6).

2 Problem Statement

Algorithm 1 Optimize Φ with optimizer M

1: **Input:** Optimizer M, Initial Program Φ, Metric μ
2: **Input:** Max Iterations I, Training Data $\mathcal{D}$
3: **Input:** Minibatch size B, Proposer Hyperparameters θ
4: **Output:** Optimized version of Φ
5:
6: M.Initialize($\mathcal{D}$, θ) ▷ Initialize optimizer using the data
7: **for** $k \leftarrow 1$ **to** I **do**
8: $(\mathbf{V} \mapsto S_k) \leftarrow M$.Propose($\theta$) ▷ Generate proposal
9: $\mathcal{D}_k \leftarrow \{(x_j, x'_j) \sim \mathcal{D}\}_{j=1}^{B}$ ▷ Sample size-B batch
10: $\sigma \leftarrow \frac{1}{B}\sum_{(x,x')\in\mathcal{D}_k} \mu(\Phi_{\mathbf{V}\mapsto S_k}(x), x')$ ▷ Validate updated program
11: M.Update($\mathbf{V} \mapsto S_k, \sigma$) ▷ Update optimizer based on the observed validation score
12: **end for**
13: $(\mathbf{V} \mapsto S_k) \leftarrow M$.ExtractOptimizedSets()
14: **return** $\Phi_{\mathbf{V}\mapsto S}$

Consider an LM program Φ consisting of m modules, each using some LM. Each module i is defined by a prompt template p_i that contains a set of variables (open slots) $\mathbf{v}$. For example, a prompt template for few-shot QA might have variables for instructions, demonstrations, and the target question.

Let $\mathbf{V}$ be the set of all variables used by prompt templates for Φ, and let $\mathbf{V} \mapsto S$ be a total assignment of variables $\mathbf{V}$ to strings S. We use $\Phi_{\mathbf{V}\mapsto S}$ to specify the program Φ run under such an assignment. Our high-level goal is to find a total assignment that optimizes Φ's performance with respect to metric μ on a trainset $\mathcal{D}$ that has inputs X and optional metadata X' (such as labels):

$$\Phi^* = \underset{\mathbf{V}\mapsto S}{\arg\max} \frac{1}{|\mathcal{D}|} \sum_{(x,x')\in\mathcal{D}} \mu(\Phi_{\mathbf{V}\mapsto S}(x), x')$$

This is the problem faced by people designing LM programs. It is intractable, as (i) each string $s \in S$ can take on any value, (ii) the metric μ provides supervision only at the level of the entire task, so every variable in $\mathbf{V}$ is latent, and (iii) we assume no access to the gradients or embeddings of the LMs involved, which rules out many RL and prompt-tuning algorithms (Zhang et al., 2022; Li and Liang, 2021; Shin et al., 2020). In addition, (iv) system designers generally have small datasets $\mathcal{D}$ and (v) small budgets of LM calls for evaluating Φ.

In many cases, we want to optimize just a subset of the variables used by Φ. In the present work, for example, we assume each prompt p has a variable i over free-form instructions and a set of K variables $\{d_{i1}, \ldots, d_{ik}\}$ over demonstrations. In these set-

tings, we assume that all the other variables for Φ are set to constant values.

To find approximate solutions to (2), we work within the general optimization framework defined by Algorithm 1. This framework generalizes prior approaches such as OPRO (Yang et al., 2023) and APE (Zhou et al., 2022) to optimizing LM programs. The main parameters to this method are the optimizer M and the unoptimized program Φ. We assume that each optimizer has methods Initialize, Propose, Update, and ExtractOptimizedSets and that it has some internal state which informs proposals, and updates on calls to update.

3 Designing LM Program Optimizers

Algorithm 1 defines a general optimization framework for LM programs. We seek efficient instantiations of this algorithm in which we minimize the number of times the program Φ is invoked (Line 10) and, as a result, the number of times we must sample proposals (Line 8). To this end, we are especially interested in building LM program optimizers with strategies that handle the proposal and credit assignment challenges discussed in Section 1. In this section, we present several novel or improved strategies for this. Section 4 then defines a few effective compositions of these strategies that allow us to empirically study their properties in practice. We showcase how these components come together to make LM Program optimizers in Figure 4.

3.1 The Proposal Problem

To make the approximate optimization tractable, we must be able to efficiently sample candidate prompts that are well suited to the nature of our task, program, data, and metric. To do this, we leverage another LM as a 'proposer' LM, and consider (i) bootstrapping few-shot examples that demonstrate how to conduct the task, (ii) collecting and summarizing important factors that could inform the construction of high-quality instructions, and/or (iii) meta-optimizing how the proposer LM is used to create high-performing instructions.

Bootstrapping Demonstrations Khattab et al. (2022, 2024) study a simple yet surprisingly effective rejection-sampling strategy for optimizing the prompts of LM programs. Inputs x are sampled from the training set, and run through $\Phi(x)$ to generate input/output traces τ for each module in the program. If the output scores as measured by metric μ are successful, i.e. $\mu(\Phi(x), x') \geq \lambda$ for some threshold λ and $(x, x') \in \mathcal{D}$, they treat all values in the trace as a potential labeled demonstrations (i.e. valid input/output examples) for the respective module in Φ. Given these potential demonstrations, the optimization problem is reduced to selecting combinations of demonstrations (within and across modules) that serve as effective few-shot examples for prompting. Khattab et al. (2024) find this can often outperform hand-written demonstrations for multi-stage programs.

Grounding How can we guide our proposal LM to craft performant instructions for a given module? We hypothesize that providing the proposal LM with relevant context, such as properties of the data, the program, and examples of successful task completions, will allow it to create instructions better suited to the task. Hence, we build a zero-shot LM program for (i) characterizing patterns in the raw dataset $\mathcal{D}$, (ii) summarizing the program's control flow, (iii) bootstrapping program demonstrations, and (iv) collecting per-stage prompts that were previously evaluated with their evaluation scores on the train set. We consider supplying each of these pieces as context to 'ground' the LM proposing our instructions. Details on constructing the dataset and program summaries are included in Appendix C.

Learning To Propose Every proposal strategy has several hyperparameters, e.g. the temperature used for instruction generation and whether to ground the proposer with a data summary, program control flow, etc. Optimal configurations of these hyperparameters may depend in practice on the task, program, and proposer LM. For example, the dataset summary may be essential to a logical reasoning task but may distract a small proposer LM for highly familiar tasks like factoid question answering. Motivated by this, in learning to propose, we parameterize proposal hyperparameters, and learn a Bayesian model over several trials to find what proposal strategy works for a given task, program, and LM setup.

3.2 Credit Assignment

Proposed assignments may be combined in many configurations. To search this space, we must identify the contribution of specific choices to LM program performance. We propose and explore three solutions for this credit-assignment problem: **greedy**, **surrogate**, and **history-based**.

Greedy As one technique, we consider proposing and evaluating single-stage changes to the LM

program separately. This method limits the misattribution of errors to incorrect stages, but is inefficient as (i) changes must be applied one at a time, and (ii) changes to some stages may not change program-level performance until other stages are improved first. Our preliminary experimentation with greedy credit assignment demonstrated that it was no more effective than other approaches but it imposed considerably worse time complexity.

Surrogate To achieve more efficient credit assignment, we propose the use of Bayesian learning, which is known for it's ability to efficiently optimize functions with multiple latent variables. In this setup, a surrogate model learns to predict the quality of different parameter combinations from previous evaluations, allowing us to focus future exploration on the promising regions of the search space. In practice, we use Optuna's implementation of the Tree Structured Parzen Optimizer to build a Bayesian model over the quality of parameter combinations for the LM program (Akiba et al., 2019; Bergstra et al., 2011). This multivariate variation of TPE models joint contributions between parameter choices, allowing us to jointly optimize our program's variables (Falkner et al., 2018). In short, surrogate-based optimization allows us to optimize efficiently over a discrete set of existing parameter proposals. However, a shortcoming of this is that it only allows for optimization over a fixed set of proposals. Learnings from past evaluations cannot be used to improve the proposals themselves.

History-Based Here we make the assumption that given a history of past evaluations, a sufficiently strong LM could perform credit assignment, removing the need for an explicit surrogate model. This would allow us to perform credit assignment *and* propose improved instructions simultaneously, as the proposer LM could in theory do both at once. Following the strategy outlined in OPRO (Yang et al., 2023), we rely on the proposer LM to learn assigned credit from a history of evaluated instructions and their scores by including these in the context. The proposer LM then outputs a new instruction based on this information. In Section 4, we cover a few ways in which the LM can be instructed to conduct this credit assignment process more or less explicitly.

4 Optimizers

We now motivate specific instantiations of optimizer algorithms composed of the algorithmic strategies outlined in Section 3. We describe the algorithms that are the focus of our experimental investigation, and leave full descriptions for the remaining optimizers to Appendix F.

4.1 Bootstrap Random Search

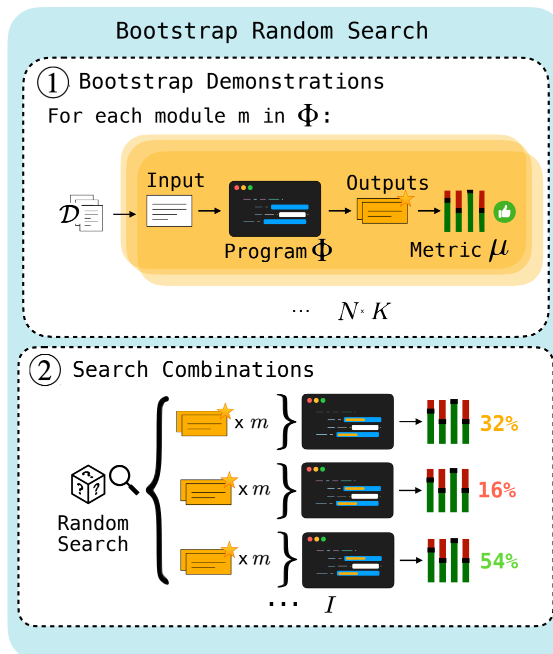

Figure 2: Bootstrap Random Search. In Step 1, demonstrations are bootstrapped by running training inputs through the program Φ and keeping traces that produce sufficiently high scoring outputs, as judged by metric μ. In Step 2, these bootstrapped demonstration sets are searched over using random search, and the most performant set is returned.

Khattab et al. (2024) achieve strong results by generating and selecting task demonstrations for each module using random search. This approach fits into our general framework and serves as a strong baseline in our experiments. This optimization procedure (highlighted in Figure 2) works as follows:

The hyperparameters include: K, the number of demonstrations to use for each module in Φ, and N, the number of total sets to bootstrap and evaluate. To **Initialize**, N sets of few-shot examples are bootstrapped using the Bootstrapping Demonstrations procedure described in 3.1. An input-output pair $(x, x') \in \mathcal{D}$ is randomly sampled from $\mathcal{D}$, where x is an input to the program and x' contains metadata (e.g. empty or final answers). Then, $\Phi(x)$ is run, which generates a full trace τ of the steps that Φ used for example x. If the output scores highly, i.e. $\mu(\Phi(x), x') \geq \lambda$ for some threshold λ and given metadata or final label x', we assume the trace to be

correct, and use the inputs / outputs for each module in the trace as candidate few-shot examples. This process is repeated until N sets of K few-shot examples for each module have been bootstrapped. To **Propose**, we sample a set of few-shot examples and use them to parameterize each module in Φ. To **Update**, the parameterized Φ is added to a global store along with its evaluation score on the training set (or a dedicated validation split thereof). To **ExtractOptimizedSets**, the top-scoring assignment is returned.

4.2 Module-Level OPRO

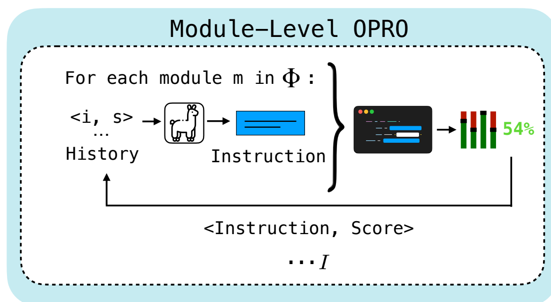

Figure 3: The Module-Level OPRO optimizer. A history of module-level instructions and program score pairs are given as input to the proposer LM to generate a new instruction for each module. These are then evaluated in the program, and the resulting score is added back with each module's instruction to the module's history. The process repeats for I iterations.

We seek to extend the OPRO algorithm (Yang et al., 2023) to an arbitrary LM program Φ, e.g. with $m \geq 1$ stages embedded in a larger program. In OPRO, the proposer LM is provided with a history of proposed instructions and their scores, allowing it to learn to propose better instructions over time. To extend OPRO, we first consider an approach we refer to as Module-Level OPRO, in which we assume that the program score is a good enough proxy for an individual instruction's quality. In other words, even though the program is parameterized with m instructions, we will assume that the score is reflective enough of each. Module-level OPRO (Figure 3) works as follows:

To **Initialize**, a seed instruction is used to parameterize each module in Φ, which is evaluated. To **Propose**, the resulting score and the i_{th} module's instruction are inputted into the proposer LM to create a new instruction for module i. This is done for all m modules. The set of m generated instructions are then used to parameterize Φ, and the parameterized program is evaluated. To **Update**, each instruction and the score is added to the history for each module. The OPRO sub-routine is run again with the updated histories to create a new set of instructions. We note that optimizers for each stage are provided with histories of module-level trajectories and proxy scores only. This process continues until the maximum number of iterations is reached. To **ExtractOptimizedSets**, the parameterization of Φ that scored highest is returned.

4.3 MIPRO

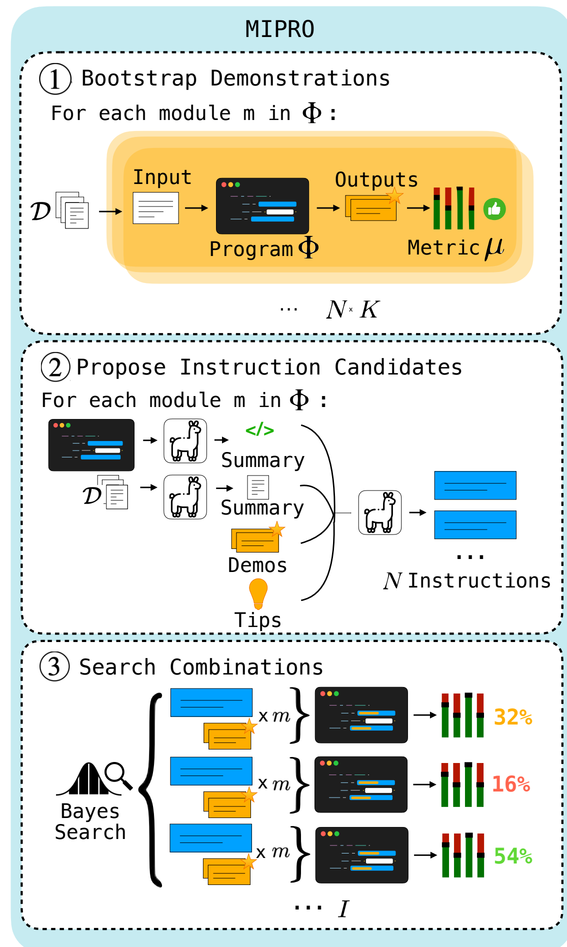

Figure 4: The MIPRO optimizer. In Step 1, demonstrations are bootstrapped using the same process from Step 1 of Bootstrap Random Search. In Step 2, instructions are proposed using the grounding strategy described in 3.1. In Step 3, Bayesian optimization is used to find the best performing combination of instruction and demonstration candidates.

To relax OPRO's strong assumptions, we propose the use of a Bayesian surrogate model to explicitly learn the sensitivity of task-level scores to module-level parameters such as instructions and demonstrations throughout optimization. We refer to

this approach as MIPRO (Multi-prompt Instruction PRoposal Optimizer). By separating the task of credit assignment from prompt proposal, we allow for the proposal LM to focus on the task of proposal only. Credit assignment and final selection is then done post-hoc using our surrogate model.

Bayesian optimization is known for its robustness to noise, as it effectively incorporates uncertainty into the optimization process (Snoek et al., 2012). We therefore propose evaluating over mini-batches of our training data, rather than the full set with each iteration. This allows us to explore and exploit parameter configurations more efficiently. The MIPRO algorithm (Figure 4) is as follows:

To **Initialize**, MIPRO bootstraps a set of N few-shot example sets and instructions per module using the Bootstrap Demonstration and Grounding strategies respectively, found in 3.1. Latent categorical variables representing the choice of few-shot examples and instructions for each module are initialized with a uniform prior. To **Propose**, we use the sampling rule from the Tree-structured Parzen Estimator (Bergstra et al., 2011) to select the instructions and few-shot examples used to parameterize Φ. To **Update**, this parameterized Φ is scored on a randomly selected mini-batch of B samples, and the scores are used to update the Estimator's priors over parameter quality. To **ExtractOptimizedSets**, every S steps, the candidate parameterizations of Φ with the highest mean score over trials is evaluated on the full train-set. At the end, the highest scoring fully evaluated parameterization is returned as the optimal assignment.

4.4 Other MIPRO variants

0-Shot MIPRO is a straightforward extension of MIPRO that simply optimizes over instructions only, rather than both jointly. This could be desirable for cost or context-window constraints.

Bayesian Bootstrap is the restricted version of MIPRO to optimizing over bootstrapped demonstrations. This may be ideal when few-shot examples are essential to the task or when we have already identified a good instruction and want to use our budget to optimize demos alone.

MIPRO++ applies a Bayesian surrogate model to optimize *proposal* hyperparameters, rather than the choice of LM program parameters themselves. This follows directly from the Learning to Propose strategy discussed in Section 3.1. In the full form of this approach, a surrogate model is used to learn optimized parameters for proposing instructions, as well as for bootstrapping demonstrations. However, in the context of this work, we focus on evaluating this approach for optimizing our instruction proposal strategy specifically (described next).

0-Shot MIPRO++ tunes instructions by meta-optimizing how or whether our Grounded instruction proposal strategy uses the dataset summary (`boolean`), uses the program summary (`boolean`), adjusts the proposer LM temperature (`float`), provides the proposer LM a plain-text tip on prompt engineering (`categorical`; see Appendix C), and selects a specific set of bootstrapped demos to show to the proposer LM (`categorical`). A Bayesian model with the same mini-batching strategy employed in MIPRO is then used to optimize over these hyperparameters. A program with the newly proposed instruction is evaluated on each trial, using the mini-batching approach described earlier. The best fully-evaluated program is returned.

4.5 Other OPRO variants

Program-Level OPRO provides the proposer LM with a history of *full, multi-stage* trajectories and relies on it to assign credit for task-level scores to the program stages. While Module-level OPRO embeds the assumptions that there is no inter-assignment dependency and that credit assignment across modules is equal, Program-level OPRO assumes that an LM will successfully complete credit-assignment when provided with long trajectory histories. These are all very strong assumptions. In particular, information contained in histories is likely to be lost as history length grows (Liu et al., 2023). In our experiments, we opt for using Module-level OPRO because Program-Level OPRO is more complex and did not appear to provide additional performance gains.

CA-OPRO "Coordinate-Ascent" OPRO (CA-OPRO) employs a greedy credit assignment approach to extend OPRO to multi-stage settings. It iterates through each module m in the program, proposes a new set of N instructions for m using Module-Level OPRO's proposal function, and evaluates each proposal by keeping all other parameters in the program fixed. It then updates module m with the best evaluated instruction so far, and repeats with the next module. This entire process is then repeated D times. From initial experiments, we found CA-OPRO's performance did not justify its inefficiency, so we focus our experiments on evaluating other methods more thoroughly.

Benchmark	Task Type	Program	Modules	LM Calls	Metric
HotPotQA	Multi-Hop QA	Multi-Hop Retrieval	2	3	Exact Match
HotPotQA Conditional	Multi-Hop QA	Multi-Hop Retrieval	2	3	Custom
Iris	Classification	Chain of Thought	1	1	Accuracy
Iris-Typo	Classification	Chain of Thought	1	1	Accuracy
Heart Disease	Classification	Answer Ensemble	2	4	Accuracy
ScoNe	Natural Language Inference	Chain of Thought	1	1	Exact Match
HoVer	Multi-Hop Claim Verify	Multi-Hop Retrieval	4	4	Recall@21

Table 1: DSPy Optimizer Benchmark and associated programs. We benchmark our optimizers on seven diverse programs. Additional details are in Appendices A and E.

5 Experimental Setup

5.1 Benchmark

We develop seven tasks (i.e. seven groups of dataset, metric, and LM program) to evaluate LM program optimizers. Table 1 presents our tasks, whose full descriptions, splits, and DSPy program pseudocode are presented in Appendices A, B.1 and E, respectively. We use 500 examples for training, 500 for our development, and a test set of 2k examples (or the full test set if smaller).

As Table 1 shows, we include four multi-stage and two single-stage programs. **HotPotQA** (Yang et al., 2018), in the "fullwiki" setting, requires systems to answer factoid questions by retrieving two relevant articles from Wikipedia. We build a program with a module for generating search queries (invoked twice) and another for generating the final answer. This is a canonical test of LM program optimizers, based on Khattab et al. 2024.

We hypothesize that optimizing *free-form instructions* can have the most impact on tasks with *subtle* rules that *cannot be properly inferred through a few examples*. We devise **HotPotQA Conditional** to test this: we change the answer format accepted from the program depending on whether the answer is a person, a date, a place, etc.

We also include two classical classification tasks: **Iris** (Fisher 1936; flower classification given six real-numbered features) and **Heart Disease** (Detrano et al. 1989; binary classification given 13 categorical and continuous features). We test Iris in two settings, one with a misspelling in the prompt that may confuse the LM and one corrected.[1] Iris can be nearly solved with a simple set of rules (not provided to the LM), so we seek to test if LM program optimizers can automatically teach LMs a Chain-of-Thought behavior to also perform well on such tasks. In contrast, it may be harder to find a small number of crucial patterns in Heart Disease, and we thus test a program that generates three clinical opinions using Chain-of-Thought LM calls and then generates a final judgment accordingly.

To assess whether optimizers can express data-specific nuances that are not evident from the program itself, we use **ScoNe** (She et al., 2023), an entailment task in which LMs must reason about logical puzzles with nested negation.

Lastly, we evaluate three-hop retrieval over unchecked claims using **HoVer** (Jiang et al., 2020). Our LM program alternates three times between generating queries, using them for retrieval from Wikipedia, and using the results to inform future queries. We use HoVer's gold labels for the documents to be retrieved for each input claim to report Retrieval@21 with all top-10 retrieved documents across three hops.

5.2 Methods & Models

We evaluate the optimizers discussed above for (i) instruction-only optimization, (ii) few-shot optimization, and (iii) joint instruction & few-shot optimization. For instruction-only optimization, we compare Module-Level OPRO, 0-Shot MIPRO, and 0-Shot MIPRO++. For optimizing few-shot demonstrations only, we compare Bootstrap Random Search (RS) with Bayesian Bootstrap. For optimizing both instructions and few-shot demonstrations together, we use MIPRO. We use an un-optimized LM Program as a baseline. In order to evaluate the utility of Grounding, we compare Module-Level OPRO with a version without Grounding, which we refer to as Module-Level OPRO −G. In these experiments, we use only the components described in the original OPRO paper

[1] The misspelling was initially accidental (asking to classify a flower as "versicolour" rather than "versicolor"). However, it serves as a realistic test for optimization from a misspecified prompt, so we report on both settings. Future work should permit LM programming abstractions to detect such errors.

in our proposal prompt: few-shot examples, and a history of previously proposed instructions and their scores. Optimizers are run for a budget of 20–50 trials with full evaluation on the trainset, depending on the task. Note that this translates to a larger number of actual optimization trials for optimizers using minibatching. More information on the exact budgets used for each task, as well as experiment hyperparameters, are detailed in Appendix B. We conduct 5 runs for each method on each task. We use Wilcoxon signed-rank tests between the averages of all runs for each example in the test set to help assess the statistical significance of performance differences between two methods.

In the majority of experiments, we use GPT-3.5 as our proposer LM (the model that crafts instructions) with a default temperature of 0.7, and Llama-3-8B as our task model (the LM used inside the LM programs). We note that the instruction proposal temperature is updated as a hyperparameter in our Learning to Learn experiments. For bootstrapping few-shot demonstrations, we use Llama-3-8B as a default teacher-model, but switch to GPT-4o for more challenging tasks (ScoNe and HoVer).

6 Results & Discussion

Table 2 summarizes our main results, from which we derive five overarching lessons.

Lesson 1: Optimizing bootstrapped demonstrations as few-shot examples is key to achieving the best performance. For the majority of tasks, we find that optimizing boostrapped demonstrations alone yields significantly better performance than optimizing instructions alone. We confirm this with a Wilcoxon signed-rank statistical test, which shows that even simple Bootstrap Random Search beats the best instruction-only optimizer for a given task in all but one case. The exception to this is HotPotQA Conditional, which supports our hypothesis and findings discussed in Lesson 3. We finally note that creating the *right* set of bootstrapped demonstrations is important. Our optimization runs indicate that there is high variation in the performance resulting from different few-shot sets (see Appendix G). We thus infer that strong bootstrapped examples provide information pertaining to successful reasoning behavior more than just, say, teaching task format.

Lesson 2: Optimizing both instructions and few-shot examples with MIPRO generally yields the best overall performance. We support this with a statistical test comparing MIPRO with the second highest averaging optimizer for each task. The exceptions to this are HotPotQA, Heart Disease, and Iris without a typo. We hypothesize that instructions are less valuable for tasks like HotPotQA, whose final module is a relatively straightforward Q&A task that is likely in-distribution for many models. For Heart Disease, we hypothesize that this is due to initializing our optimizers with a simple seed instruction that does not convey any classification criteria, which current instruction optimizers have a limited ability to infer.

Lesson 3: Instruction optimization is most important for tasks with conditional rules that are (i) not immediately obvious to the LM and (ii) not expressible via a limited number of few-shot examples. This hypothesis is supported by our Iris-Typo experiment—and primarily by our HotPotQA Conditional experiments, where we optimize over a seed instruction stating the rules of the task. For this task, we find that even 0-shot instruction optimization outperforms demonstration-only optimization. In these cases, especially if the task is complex, it's important that we optimize over a seed prompt, as our optimizers are not yet able to infer all task rules. (This lesson is also reflected in the Heart Disease results, as discussed above.) In the Iris-Typo setting, our instruction optimizer even helps correct mistakes in the seed prompt.

Lesson 4: Grounding is helpful for instruction proposal overall, but the best proposal strategy varies by task. In our Module-Level OPRO Grounding ablations, we find that Grounding is essential for performance improvements for HotPotQA and HoVer, but seems to hurt performance for ScoNe. This motivates approaches like MIPRO++, which are able to learn custom proposal strategies for a given task. Indeed, we see that 0-Shot MIPRO++ is able to learn a proposal strategy that recovers this performance for ScoNe. One additional benefit of 0-Shot MIPRO++ is that the learned importance scores from the Bayesian model used to optimize proposal hyperparameters can provide us insight into the utility of each proposal component. Studying these importance scores (found in Appendix D) reveals, across tasks, the highest importance scores go to the choice of bootstrapped demonstrations in the meta-prompt and the tip. We observe that the importance of many parameters varies between tasks: for example, the dataset summary has a high learned importance score for ScoNe, whereas it is one of the least

Optimizer	ScoNe			HotPotQA			HoVer			HotPotQA Cond.			Iris		Iris-Typo		Heart Disease	
	Train	Dev	Test	Train	Dev	Test	Train	Dev	Test	Train	Dev	Test	Train	Test	Train	Test	Train	Test
Instructions only (0-shot)																		
N/A	57.0	56.2	69.1	35.4	31.8	36.1	30.2	30.8	25.3	13.8	10.5	6	46.4	40.9	34.7	32	23.3	26.8
Module-Level OPRO −G	70.0	67.4	76.1	36.0	31.7	36.0	30.0	30.0	25.7	–	–	–	–	–	–	–	–	–
Module-Level OPRO	69.1	67.6	73.5	41.9	36.2	39.0	37.1	38.6	32.5	–	–	–	–	–	–	–	–	–
0-Shot MIPRO	66.3	65.2	71.5	40.2	34.2	36.8	37.7	38.4	33.1	22.6	20.3	14.6	40.8	36.4	56.8	56.7	26.8	25.8
0-Shot MIPRO++	69.0	66.9	75.7	41.5	36.2	39.3	37.1	37.3	32.6	–	–	–	–	–	–	–	–	–
Demonstrations only (Few-shot)																		
Bootstrap RS	74.9	69.6	75.4	48.6	44.0	**45.8**	42.0	42.0	37.2	16.4	15.0	10.4	95.2	**94.1**	58.9	58.7	78.4	**79.2**
Bayesian Bootstrap	75.4	67.4	77.4	49.2	44.8	**46.2**	44.6	44.7	37.6	–	–	–	–	–	–	–	–	–
Both (Few-shot)																		
MIPRO	74.6	69.8	**79.4**	49.0	43.9	**46.4**	44.7	46.7	**39.0**	28.4	28.1	**23.3**	98.4	88.6	69.1	**68.7**	75.2	74.2

Table 2: Results averaged across 5 runs, divided into optimizing instructions only (i.e., "0-shot" prompts), demonstrations only, and both. The best performing values in each column are highlighted in bold. These bold values represent the highest average scores compared to the second-highest, with significance supported by Wilcoxon signed-rank tests ($p < .05$) between the corresponding run averages. If significance is not confirmed, multiple results are bolded to denote their comparable performance.

important parameters for HotPotQA and HoVer.

Lesson 5: There is more to learn about LM program optimizers. When comparing the performance of Module-Level OPRO, 0-Shot MIPRO, and 0-Shot MIPRO++, we find that results are mixed. Bayesian Bootstrap outperforms Bootstrap Random Search for ScoNe, but this finding is not statistically significant for HotPotQA and HoVer. 0-shot MIPRO++ outperforms 0-shot MIPRO for ScoNe and HotPotQA, but is about equivalent in the case of HoVer. Future work may find more differentiated results when studying these optimizers at different optimization budgets: for example, it's possible that 0-shot MIPRO would perform best in very low budget settings, given that it's use of mini-batching allows it to explore many more parameter options with the same budget. Conversely, 0-shot MIPRO++ may shine in scenarios where budget is not an issue, and spending many trials to learn optimal proposal dynamics could lead to differentiated results. We save this exploration for future work.

7 Related Work

Recent work has explored optimizing string prompts, including gradient-guided search (Shin et al., 2020; Wen et al., 2023), reranking brute force search (Gao et al., 2021), evolutionary algorithms (Fernando et al., 2023), prompting other LMs (Yang et al., 2023; Zhou et al., 2023; Pryzant et al., 2023), and reinforcement learning (RL) (Deng et al., 2022; Zhang et al., 2022; Hao et al., 2022). Prior work on RL for prompts focuses on word level edits of only a few words (Deng et al., 2022), phrase level edits (Zhang et al., 2022), or text-to-image generation (Hao et al., 2022). Khattab et al. (2024) present DSPy, a programming model for expressing LM programs and optimizing their prompts and weights. Unlike our work, the authors only explore optimizers based on bootstrapping strong demonstrations. Sordoni et al. (2023) explore joint prompt optimization for stacked LLM calls, modeling this as variational inference and exploring this for two simple layers. Their approach inherently relies on having access to log probabilities for explicitly passed tokens to LMs, an increasingly restrictive assumption in practice (e.g. with commodified LM APIs that allow only text-in-text-out processing). In contrast, our optimizers work on an arbitrary number of modules for any LM program out-of-the-box.

8 Conclusion

We formalize the problem of optimizing prompts in LM programs (§2). We identify two key challenges of LM program prompt optimization: (1) proposal of a small set of high-quality prompts and (2) credit assignment during optimization. We address these challenges using three proposal generation and three credit assignment strategies (§3) and explore a representative subset of optimizer algorithms (§4) using a new benchmark of diverse tasks (§5.1). Our findings show that optimizing few-shot demonstrations is very powerful, but instruction optimization can be essential for complex task specifications with multiple conditional rules. Finally, we find that jointly optimizing both instructions and demonstrations using the MIPRO optimizer is the most effective approach in five out of seven settings.

Limitations

This work studies a set of optimizers under a fixed budget, but does not examine how optimization dynamics might differ across extremely low or high budget scenarios. As discussed in Section 6, doing so may reveal new insights about the trade-offs between different optimizers, such as those that learn to improve proposals overtime, versus those that optimize over existing proposals very efficiently. In our experiments, we also use a fixed proposer LM and task LM. Future research should assess whether the proposed methods demonstrate consistent performance when employing different models. Furthermore, a limitation of the optimizers introduced in this work is their restricted ability to infer the rules governing complex tasks without a handwritten seed prompt. While some information can be gleaned from grounding—such as dataset details or examples of the task—this may be insufficient for extrapolating a comprehensive set of rules. We encourage subsequent studies to investigate how optimizers could learn such task dynamics without relying on handwritten inputs. Finally, while we have made efforts to establish a benchmark that covers a diverse range of tasks and programs, there remains more to learn regarding the performance of optimizer methods on increasingly complex tasks and programs. Improving this benchmark presents a promising avenue for future research.

Acknowledgements

Krista Opsahl-Ong is supported by the NSF CS-Grad4US Graduate Fellowship. This work was partially supported by IBM as a founding member of the Stanford Institute for Human-Centered Artificial Intelligence (HAI), and by the HAI Hoffman–Yee Grant "Dendritic Computation for Knowledge Systems". Additional support was provided by the Wallenberg AI, Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation, as well as by Digital Futures at KTH. Finally, this research was also supported in part by affiliate members and other supporters of the Stanford DAWN project – Facebook, Google, and VMware.

References

AI@Meta. 2024. Llama 3 model card.

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631.

James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24.

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2023. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1946–1969.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Antonia Creswell and Murray Shanahan. 2022. Faithful reasoning using large language models. *Preprint*, arXiv:2208.14271.

Mingkai Deng, Jianyu Wang, Cheng-Ping Hsieh, Yihan Wang, Han Guo, Tianmin Shu, Meng Song, Eric Xing, and Zhiting Hu. 2022. RLPrompt: Optimizing discrete text prompts with reinforcement learning. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3369–3391, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Robert C. Detrano, András Jánosi, Walter Steinbrunn, Matthias Emil Pfisterer, Johann-Jakob Schmid, Sarbjit Sandhu, Kern Guppy, Stella Lee, and Victor Froelicher. 1989. International application of a new probability algorithm for the diagnosis of coronary artery disease. *The American journal of cardiology*, 64 5:304–10.

David Dohan, Winnie Xu, Aitor Lewkowycz, Jacob Austin, David Bieber, Raphael Gontijo Lopes, Yuhuai Wu, Henryk Michalewski, Rif A Saurous, Jascha Sohl-Dickstein, et al. 2022. Language model cascades. *arXiv preprint arXiv:2207.10342*.

Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and efficient hyperparameter optimization at scale. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1437–1446. PMLR.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2023. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*.

Rory A. Fisher. 1936. The use of multiple measurements in taxonomic problems. *Annals of Human Genetics*, 7:179–188.

Tianyu Gao, Adam Fisch, and Danqi Chen. 2021. 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)*, pages 3816–3830, Online. Association for Computational Linguistics.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. 2024. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. *Preprint*, arXiv:2309.08532.

Yaru Hao, Zewen Chi, Li Dong, and Furu Wei. 2022. Optimizing prompts for text-to-image generation. *arXiv preprint arXiv:2212.09611*.

Yichen Jiang, Shikha Bordia, Zheng Zhong, Charles Dognin, Maneesh Singh, and Mohit Bansal. 2020. HoVer: A dataset for many-hop fact extraction and claim verification. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 3441–3460, Online. Association for Computational Linguistics.

Omar Khattab, Christopher Potts, and Matei Zaharia. 2021. Baleen: Robust Multi-Hop Reasoning at Scale via Condensed Retrieval. In *Thirty-Fifth Conference on Neural Information Processing Systems*.

Omar Khattab, Keshav Santhanam, Xiang Lisa Li, David Hall, Percy Liang, Christopher Potts, and Matei Zaharia. 2022. Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive nlp. *arXiv preprint arXiv:2212.14024*.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan A, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2024. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*.

Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. 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)*, pages 4582–4597, Online. Association for Computational Linguistics.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*.

Jiayi Pan, Yichi Zhang, Nicholas Tomlin, Yifei Zhou, Sergey Levine, and Alane Suhr. 2024. Autonomous evaluation and refinement of digital agents. *Preprint*, arXiv:2404.06474.

Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *arXiv preprint arXiv:2304.11015*.

Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. 2023. Automatic prompt optimization with" gradient descent" and beam search. *arXiv preprint arXiv:2305.03495*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *Preprint*, arXiv:2307.16789.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. 2024. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*.

Imanol Schlag, Sainbayar Sukhbaatar, Asli Celikyilmaz, Wen-tau Yih, Jason Weston, Jürgen Schmidhuber, and Xian Li. 2023. Large language model programs. *arXiv preprint arXiv:2305.05364*.

Jingyuan S. She, Christopher Potts, Samuel R. Bowman, and Atticus Geiger. 2023. ScoNe: Benchmarking negation reasoning in language models with fine-tuning and in-context learning. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 1803–1821, Toronto, Canada. Association for Computational Linguistics.

Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. 2020. AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 4222–4235, Online. Association for Computational Linguistics.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical bayesian optimization of machine learning algorithms. *Preprint*, arXiv:1206.2944.

Alessandro Sordoni, Xingdi Yuan, Marc-Alexandre Côté, Matheus Pereira, Adam Trischler, Ziang Xiao, Arian Hosseini, Friederike Niedtner, and Nicolas Le Roux. 2023. Joint prompt optimization of stacked llms using variational inference. *Preprint*, arXiv:2306.12509.

Dilara Soylu, Christopher Potts, and Omar Khattab. 2024. Fine-tuning and prompt optimization: Two great steps that work better together. *Preprint*, arXiv:2407.10930.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*.

Yuxin Wen, Neel Jain, John Kirchenbauer, Micah Goldblum, Jonas Geiping, and Tom Goldstein. 2023. Hard prompts made easy: Gradient-based discrete optimization for prompt tuning and discovery. *arXiv preprint arXiv:2302.03668*.

Tongshuang Wu, Michael Terry, and Carrie Jun Cai. 2022. Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts. In *Proceedings of the 2022 CHI conference on human factors in computing systems*, pages 1–22.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. 2023. Large language models as optimizers. *arXiv preprint arXiv:2309.03409*.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. 2024. Large language models as optimizers. *Preprint*, arXiv:2309.03409.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2369–2380, Brussels, Belgium. Association for Computational Linguistics.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. *Preprint*, arXiv:2210.03629.

Tianjun Zhang, Xuezhi Wang, Denny Zhou, Dale Schuurmans, and Joseph E Gonzalez. 2022. Tempera: Test-time prompt editing via reinforcement learning. In *The Eleventh International Conference on Learning Representations*.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. Sglang: Efficient execution of structured language model programs. *Preprint*, arXiv:2312.07104.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910*.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2023. Large language models are human-level prompt engineers. *Preprint*, arXiv:2211.01910.

A Detailed Task Descriptions

HotPotQA (Yang et al., 2018) is a dataset of questions that require reasoning over multiple Wikipedia articles to answer. We adopt the "full-wiki" setting in which the system must retrieve the right articles from all 5M Wikipedia page abstracts. Our LM program, derived from Khattab et al. 2024, involves three stages, given a question: (1) generating a search query that a retrieval model uses to find Wikipedia passages, (2) reading those passages to generate a second "hop" query for the retrieval model, and finally (3) reading all of the retrieved passages to answer the question.

HotPotQA Conditional We hypothesize that optimizing instructions can have the most impact on tasks with rules that (1) are not immediately obvious to the task LM and (2) cannot be fully defined through a few examples. To test this, we devise a task that applies additional conditional rules to HotPotQA: depending on the category of the answer, the LM must respond in a specific format:

- When the answer is a person, the response must be in lowercase. When it's a place, the response should contain no punctuation.
- When the it's a date, the response must end with "Peace!", but in no other circumstances.
- If answer falls into another category, the response must be in all caps.

We use GPT-4 to annotate the answer types as "person", "place", or "date" with author supervision for applying special conditional rules based on answer type. We use the same multi-hop LM program described above to solve this task, and initialize the program with an instruction describing the rules of the task. This program is then evaluated using exact match along with regex parsing to determine if all other conditions are being followed.

Iris is a classic classification dataset over floating-point features (Fisher, 1936). It involves classifying a flower as one of three iris species given the flower's sepal and petal width and length. Our LM program for this task is a simple Chain-of-Thought program, initialized the instruction: "Given the petal and sepal dimensions in cm, predict the iris species." Crucially, we know that Iris can be nearly solved with a simple set of rules (not provided to the LM) and seek to test if LM program optimizers can automatically teach LMs to also perform well on such tasks.

Heart Disease (Detrano et al., 1989) is a classification task where it may be harder to find a small number of crucial patterns. Given a set of 13 features including a patient's age, biological sex, and cholesterol, we must predict whether the patient has heart disease. Our LM program generates three separate clinical opinions using Chain-of-Thought LM calls and then generates a final judgment based on the opinions given.

ScoNe (She et al., 2023) evaluates logical reasoning with negation. We define a simple Chain-of-Thought program to reason over ScoNe entailment questions and produce a binary answer. ScoNe allows us to represent rich single-stage tasks in our evaluation. We hypothesize that ScoNe's focus on NLI and logical deduction with nested negations evaluates whether optimizers can express task nuances via instructions or demonstrations.

HoVer (Jiang et al., 2020) contains claims that require many-hop search steps over Wikipedia to be fact-checked. We consider the sub-task of retrieving all required evidence. Our LM program performs a search on the claim, summarizes the results (LM call 1), adds that to context and proposes the next search query (LM call 2), performs a search, summarizes the results (LM call 3), proposes a search query based on both summaries (LM call 4), and finally searches once more. HoVer has gold labels for the documents to be retrieved. We measure Retrieval@21 with all top-10 retrieved documents across three hops. We restrict HoVer to examples with 3 supporting facts in order to study optimization in a more challenging setting. We note that different supporting facts can originate from the same document, so finding all required documents can sometimes be done in <3 hops. Approximately 67% (83%) of the training (test) sets are 3-hop queries, and the rest are 2-hop queries. In follow-up studies, we recommend filtering by examples that require 3-hops to study retrieval in a more uniform setting.

B Experiment Setup Details

B.1 Data Splits

Our data splits are described in Table 3. We generally use 500 examples for training, 500 for our development, and a test set of 2k examples (or the full test set for tasks with test sets <2k samples).

Benchmark	Train	Dev	Test
HotPotQA	500	500	2000
HotPotQA Conditional	500	200	200
Iris	75	N/A	75
Heart Disease	120	N/A	183
ScoNe	500	500	1200
HoVer	500	500	1520

Table 3: Train, Dev, Test splits for each of our datasets. Training data was used for training our Optimizers. Dev data was used as a development set to internally iterate on methods. Test was used for final evaluation and reporting. The datset for HotPotQA Conditional is smaller because the labels were created by hand. We do not create a devset for Iris or Heart Disease due to the fact that these were (1) small datasets and (2) we did not use them for method iteration.

B.2 Optimizer Budget

We optimize HotPotQA and ScoNe with a budget of 50 full evaluation trials (which translates to about 300 minibatch trials). Iris, Heart Disease, and HotPotQA Conditional are run with a budget of 30 full evaluation trials, and HoVer 20 full evaluation trials. We use less trials for HoVer experiments given that it is the most expensive program to run. We use less trials for Iris, Heart Disease, and HotPotQA Conditional given that these experiments were focused on understanding the value of instruction versus few-shot optimization rather than evaluating specific methods, which may need more trials to see differentiated results.

B.3 Optimizer Hyperparameters

Table 4 shows the number of instruction and / or few-shot demonstration candidates for each module (N) that 0-Shot MIPRO, Bayesian Bootstrap, and MIPRO were used to optimize over in our experiments. We note that these hyperparameters were not chosen with extensive sweeps, but were instead chosen built on intuition built over running past experiments. Our general rule of thumb was to set N to be < T/v, where T is the trial optimization budget, and v is the total number of variables we are optimizing over.

B.4 Language Model Hyperparameters

We perform most of our experiments with the LLama 3 8B model (AI@Meta, 2024) which we serve using SGLang (Zheng et al., 2024) on A100 GPUs. We parallelize inference calls across 8 A100 GPUs for many of our experiments. However, only a single GPU capable of running Llama 3 8B or a cloud inference provider would be necessary to replicate all results. The temperature of our Llama model is optimized as a part of the MIPRO++ experiments, but we otherwise use temperature 0.7. We also always use top_p=1.0 sampling for Llama. We generate until the model reaches the max tokens for a given task or the stop tokens

$$"\backslash n\backslash n", "\backslash n---", "assistant"$$

. For ScoNe this is 200 tokens, for HoVer this is 600 tokens, and for all other tasks this is 150 tokens. Importantly we run Llama 3 without a chat template as we find this behaves better given DSPy's autocomplete prompt style. For our proposer model, we use GPT-3.5 with temperature 0.7 and top_p=1.0 or GPT-4 with the same settings for ScoNe, HoVer, and Iris.

Optimizer	Benchmark	N
0-Shot MIPRO	HotPotQA	60
	HotPotQA Conditional	35
	Iris	50
	Heart Disease	30
	ScoNe	70
	HoVer	15
Bayesian Bootstrap	HotPotQA	60
	HotPotQA Conditional	N/A
	Iris	N/A
	Heart Disease	N/A
	ScoNe	70
	HoVer	15
MIPRO	HotPotQA	30
	HotPotQA Conditional	30
	Iris	30
	Heart Disease	15
	ScoNe	70
	HoVer	10

Table 4: Number of candidates per module (N) used for optimization. Note that this hyperparameter only applies to our 0-Shot MIPRO, MIPRO, and Bayesian Bootstraping optimizers, because for other optimizers the # of candidates explored equals the number of trials.

C Grounding Details

The following section includes details regarding the grounded prompting strategy used in our methods.

C.1 Instruction Proposal Program

Below is the signature for an LM program we use to generate instruction candidates. Note that a separate module is used to generate the dataset description and the program description.

```
class GenerateSingleModuleInstruction(dspy.Signature):
    (
```

```
        """Use the information below to learn about a
    task that we are trying to solve using calls to an LM,
    then generate a new instruction that will be used to
    prompt a Language Model to better solve the task."""
      )
      dataset_description = dspy.InputField(desc="A
    description of the dataset that we are using.",)
      program_code = dspy.InputField(desc="Language model
    program designed to solve a particular task.",)
      program_description = dspy.InputField(desc="Summary
    of the task the program is designed to solve, and how
    it goes about solving it.")
      module = dspy.InputField(desc="The module to create
    an instruction for.")
      task_demos = dspy.InputField(desc="Example inputs/
    outputs of our module.")
      previous_instructions = dspy.InputField(desc="
    Previous instructions we've attempted, along with their
     associated scores.")
      basic_instruction = dspy.InputField(desc="Basic
    instruction.")
      tip = dspy.InputField(desc="A suggestion for how to
    go about generating the new instruction.")
      proposed_instruction = dspy.OutputField(desc="
    Propose an instruction that will be used to prompt a
    Language Model to perform this task.")
```

Listing 1: Instruction Proposal Program

C.2 Tips

List of instruction generation tips, used to encourage diversity of features in the instructions generated.

```
tips = {
    "none": "",
    "creative": "Don't be afraid to be creative!",
    "simple": "Keep the instruction clear and concise.",
    "description": "Make sure your instruction is very
     informative and descriptive.",
    "high_stakes": "The instruction should include a high
     stakes scenario in which the LM must solve the task!",
    "persona": "Provide the LM with a persona that is
     relevant to the task (ie. \"You are a ...\")"
}
```

Listing 2: Instruction Proposal Program

C.3 Dataset Summary Generation Process

In order to write our dataset summaries we looped over the training set in batches and ask the proposer LM to write a set of observations, given a previous set of observations. If the LM has nothing to add then we ask it to output "COMPLETE". If the LM outputs "COMPLETE" 5 times, then we stop looping through the training set. Next we ask the LM to summarize the observations, which produces the dataset summaries included below.

Dataset Descriptor Prompt `Given several examples from a dataset please write observations about trends that hold for most or all of the samples. I will also provide you with a few observations I have already made. Please add your own observations or if you feel the observations are comprehensive say 'COMPLETE'. Some areas you may consider in your observations: topics, content, syntax, conciceness, etc. It will be useful to make an educated guess as to the nature of the task this dataset will enable. Don't be afraid to be creative`

Dataset Summarizer Prompt `Given a series of observations I have made about my dataset, please summarize them into a brief 2-3 sentence summary which highlights only the most important details.`

C.4 Example Dataset Summaries

We include the generated dataset summaries from a few of our tasks below in order to provide examples of how these look.

ScoNe. "The dataset consists of logical reasoning tasks involving negations and double negations, challenging individuals to make deductions based on provided scenarios and questions. The focus is on testing the ability to draw logical inferences accurately while paying attention to details, with a variety of categories indicating diverse reasoning challenges. Overall, the dataset aims to evaluate models' performance in logical reasoning tasks by emphasizing the implications of negations on drawing conclusive deductions."

HotpotQA. "The dataset contains trivia-style questions from a wide range of topics like music, film, history, and literature. Questions are well-structured and require specific information as answers, suggesting a focus on testing knowledge. The dataset's consistent format and emphasis on accuracy make it suitable for developing a trivia quiz application or knowledge testing platform."

HoVeR. "The dataset consists of structured claims supported by specific facts, focusing on comparisons between entities, relationships, and specific characteristics. It prioritizes accuracy, specificity, and detailed information retrieval, enabling diverse fact-checking tasks across various topics such as music, sports, literature, and film. The consistent emphasis on validation and accuracy through supporting facts suggests a strong foundation for verifying claims within the dataset."

HotPotQA Conditional. "Many of the questions in this dataset are concise and to the point, indicating that the questions are well-structured and aimed at eliciting specific information. The dataset covers a wide range of topics, including sports, music, history, professions, and notable figures. The categories provided for each question are also well-organized and help to identify the specific type of information being requested. The emphasis on

factual information rather than opinions or interpretations is also notable, as it suggests that the dataset is intended for use in objective and verifiable knowledge assessments. Furthermore, the inclusion of temporal elements, such as specific years and durations, and the focus on prominent figures and events, indicate that the dataset is designed to test knowledge of specific events, people, and eras"

Iris. "The dataset appears to have three classes: setosa, versicolor, and virginica. Each of the classes has a distinct distribution on the petal_length and petal_width variables. However, the sepal_length and sepal_width variables appear to have less clear cut trends. Additionally, the species variable does not provide much additional information since it is a duplicate of the answer variable. — Is this correct? Can I improve this? Please let me know if so."

Heart Disease. "The dataset shows a distribution of ages from 30s to 60s, with a higher proportion of male subjects. Chest pain types include asymptomatic, non-anginal pain, atypical angina, and typical angina. Blood pressure and cholesterol levels vary, but most values fall within typical ranges. Exercise-induced angina is uncommon, and ST depression induced by exercise is generally mild. The dataset appears to model a binary outcome predicting the presence or absence of heart disease.2. **Sex**: There is a higher occurrence of male subjects compared to female subjects in this dataset. 3. **Chest Pain Types (cp)**: Most common chest pain type observed is asymptomatic, followed by non-anginal pain, atypical ang"

C.5 Program Summarization Process

We also ask the proposer LM to summarize the LM program to include as context when grounding the proposer. To do this we reflexively include the DSPy code directly into a summarizer for the program. We ask it to highlight two details in its summary (1) The task this program is intended to solve and (2) How it appears to work.

Program Summarizer Prompt `Below is some pseudo-code for a pipeline that solves tasks with calls to language models. Please describe what type of task this program appears to be designed to solve, and how it appears to work.`

D Learned Feature Importances

D.1 ScoNe

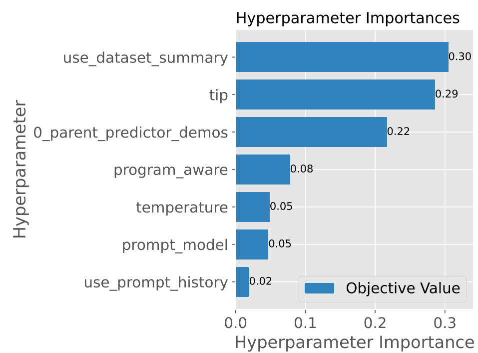

Figure 5: Learned hyperparameter importances for ScoNe. Here we see that the Bayesian model learned the dataset summary, the tip, and the task demos in the prompt to be important to proposal quality.

D.2 HotpotQA

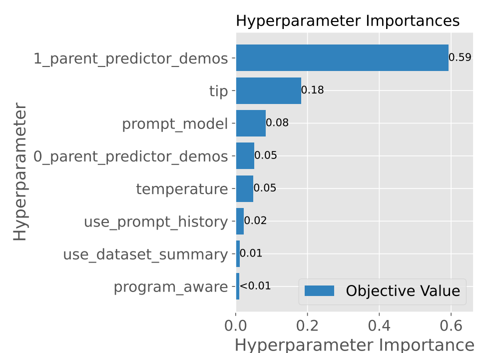

Figure 6: Learned hyperparameter importances for HotpotQA. Here we see that the set of demonstrations chosen for the meta-prompt were most important for proposal quality.

D.3 HoVeR

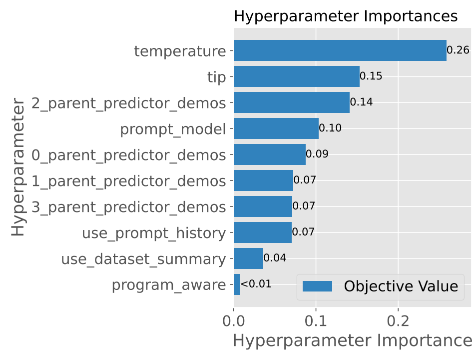

Figure 7: Learned hyperparameter importances for HoVeR. Again, we see that the task demonstrations chosen (ie. the parent predictor demos, as labeled here for each module in the program) are learned to be important, as well as the tip. In this case, we also see that the proposal temperature is learned to be important to the model.

E DSPy LM Programs

We provide pseudocode for all LM programs. We will also publicly release the code for this benchmark on publication.

E.1 ScoNe

```
class ScoNeSignature(dspy.Signature):
  ("""context, question -> answer""")

  context = dspy.InputField()
  question = dspy.InputField()
  answer = dspy.OutputField(desc="Yes or No")

class ScoNeCoT(dspy.Module):
  def __init__(self):
    self.generate_answer = dspy.ChainOfThought(
      ScoNeSignature)

  def forward(self, context, question):
    return self.generate_answer(context, question)
```

Listing 3: ScoNe Program

E.2 HotpotQA

```
class MultiHop(dspy.Module):
  def __init__(self):
    self.retrieve = dspy.Retrieve(k=3)
    self.generate_query = dspy.ChainOfThought("context,
      question->search_query")
    self.generate_answer = dspy.ChainOfThought("context,
      question->answer")

  def forward (self, question) :
    context = []
    for hop in range(2):
      query = self.generate_query(context, question).
      search_query
      context += self.retrieve(query).passages
    return self.generate_answer(context, question).answer
```

Listing 4: HotPotQA Program

E.3 HoVeR

```
class RetrieveMultiHop(dspy.Module):
  def __init__(self):
    super().__init__()
    self.k = 7
    self.create_query_hop2 = dspy.ChainOfThought("claim,
      summary_1->query")
    self.create_query_hop3 = dspy.ChainOfThought("claim,
      summary_1,summary_2->query")
    self.retrieve_k = dspy.Retrieve(k=self.k)
    self.summarize1 = dspy.ChainOfThought("claim,passages->
      summary")
    self.summarize2 = dspy.ChainOfThought("claim,context,
      passages->summary")

  def forward(self,claim):
    # HOP 1
    hop1_docs = self.retrieve_k(claim).passages
    summary_1 = self.summarize1(claim=claim, passages=
      hop1_docs).summary # Summarize top k docs

    # HOP 2
    hop2_query = self.create_query_hop2(claim=claim,
      summary_1=summary_1).query
    hop2_docs = self.retrieve_k(hop2_query).passages
    summary_2 = self.summarize2(claim=claim, context=
      summary_1, passages=hop2_docs).summary

    # HOP 3
    hop3_query = self.create_query_hop3(claim=claim,
      summary_1=summary_1, summary_2=summary_2).query
    hop3_docs = self.retrieve_k(hop3_query).passages

    return dspy.Prediction(retrieved_docs = hop1_docs +
      hop2_docs + hop3_docs)")
```

Listing 5: HoVeR Retrieval Program

E.4 HotPotQA Conditional With Handwritten Seed Instructions

```
class GenerateAnswerInstruction(dspy.Signature):
  """When the answer is a person, respond entirely in
      lowercase.  When the answer is a place, ensure your
      response contains no punctuation.  When the answer is a
       date, end your response with "Peace!".  Never end your
       response with "Peace!" under other circumstances.
      When the answer is none of the above categories respond
       in all caps."""

  context = dspy.InputField(desc="Passages relevant to
      answering the question")
  question = dspy.InputField(desc="Question we want an
      answer to")
  answer = dspy.OutputField(desc="Answer to the question")

class MultiHopHandwritten(dspy.Module):
  def __init__(self, passages_per_hop):
    super().__init__()
    self.retrieve = dspy.Retrieve(k=passages_per_hop)
    self.generate_query = dspy.ChainOfThought("context ,
      question->search_query")
    self.generate_answer = dspy.ChainOfThought(
      GenerateAnswerInstruction)

  def forward(self, question):
    context = []
    for hop in range(2):
      query = self.generate_query(context=context, question=
      question).search_query
      context += self.retrieve(query).passages
    return dspy.Prediction(
      context=context,
      answer=self.generate_answer(context=context, question=
      question).answer,
    )
```

Listing 6: HotPotQA Conditional Program with Handwritten Instructions. We use the same HotPotQA program above when starting from no seed instructions.

E.5 Iris

```
class IrisSig(dspy.Signature):
  "Given the petal and sepal dimensions in cm, predict the
      iris species."

  petal_length = dspy.InputField()
  petal_width = dspy.InputField()
```

```
  sepal_length = dspy.InputField()
  sepal_width = dspy.InputField()
  answer = dspy.OutputField(desc='setosa, versicolor, or
      virginica')

class Classify(dspy.Module):
  def __init__(self):
    self.pred = dspy.ChainOfThought(IrisSig)

  def forward(self, petal_length, petal_width, sepal_length,
       sepal_width):
    return self.pred(petal_length=petal_length, petal_width=
      petal_width, sepal_length=sepal_length, sepal_width=
      sepal_width)
```

Listing 7: Iris Program for predicting flower species given plant properties

E.6 Iris-Typo

```
class IrisSig(dspy.Signature):
  "Given the petal and sepal dimensions in cm, predict the
      iris species."

  petal_length = dspy.InputField()
  petal_width = dspy.InputField()
  sepal_length = dspy.InputField()
  sepal_width = dspy.InputField()
  answer = dspy.OutputField(desc='setosa, versicolour, or
      virginica')

class Classify(dspy.Module):
  def __init__(self):
    self.pred = dspy.ChainOfThought(IrisSig)

  def forward(self, petal_length, petal_width, sepal_length,
       sepal_width):
    return self.pred(petal_length=petal_length, petal_width=
      petal_width, sepal_length=sepal_length, sepal_width=
      sepal_width)
```

Listing 8: Iris Program for predicting flower species given plant properties. This program contains a typo that tells the LM to classify a flower species as "versicolour" instead of "versicolor"

E.7 Heart Disease

```
class HeartDiseaseInput(dspy.Signature):
  age = dspy.InputField(desc="Age in years")
  sex = dspy.InputField(desc="Sex (male or female)")
  cp = dspy.InputField(desc="Chest pain type (typical angina
      , atypical angina, non-anginal pain, asymptomatic)")
  trestbps = dspy.InputField(desc="Resting blood pressure (
      in mm Hg on admission to the hospital)")
  chol = dspy.InputField(desc="Serum cholestoral in mg/dl")
  fbs = dspy.InputField(desc="Fasting blood sugar > 120 mg/
      dl (true or false)")
  restecg = dspy.InputField(desc="Resting
      electrocardiographic results (normal, ST-T wave
      abnormality, left ventricular hypertrophy)")
  thalach = dspy.InputField(desc="Maximum heart rate
      achieved")
  exang = dspy.InputField(desc="Exercise induced angina (yes
       or no)")
  oldpeak = dspy.InputField(desc="ST depression induced by
      exercise relative to rest")
  slope = dspy.InputField(desc="The slope of the peak
      exercise ST segment (upsloping, flat, downsloping)")
  ca = dspy.InputField(desc="Number of major vessels (0-3)
      colored by flourosopy")
  thal = dspy.InputField(desc="Thalassemia (normal, fixed
      defect, reversible defect)")

class HeartDiseaseSignature(HeartDiseaseInput):
  """Given patient information, predict the presence of
      heart disease."""

  answer = dspy.OutputField(desc="Does this patient have
      heart disease? Just yes or no.")

class HeartDiseaseVote(HeartDiseaseInput):
  """Given patient information, predict the presence of
      heart disease. I can critically assess the provided
      trainee opinions."""

  context = dspy.InputField(desc="A list of opinions from
      trainee doctors.")
  answer = dspy.OutputField(desc="Does this patient have
      heart disease? Just yes or no.")
```

```
class Classify(dspy.Module):
  def __init__(self):
    self.classify = [dspy.ChainOfThought(
      HeartDiseaseSignature, temperature=0.7 + i*0.01) for i
      in range(3)]
    self.vote = dspy.ChainOfThought(HeartDiseaseVote)

  def forward(self, age, sex, cp, trestbps, chol, fbs,
      restecg, thalach, exang, oldpeak, slope, ca, thal):
    kwargs = dict(age=age, sex=sex, cp=cp, trestbps=trestbps
      , chol=chol, fbs=fbs, restecg=restecg,
           thalach=thalach, exang=exang, oldpeak=oldpeak,
      slope=slope, ca=ca, thal=thal)

    opinions = [c(**kwargs) for c in self.classify]
    opinions = [(opinion.rationale.replace('\n', ' ').strip(
      '.'), opinion.answer.strip('.')) for opinion in
      opinions]
    opinions = [f"I'm a trainee doctor, trying to {reason}.
      Hence, my answer is {answer}." for reason, answer in
      opinions]
    return self.vote(context=opinions, **kwargs)
```

Listing 9: Heart Disease Program for classifying patients as possessing heart disease. We have LMs simulate doctor opinions and then give them to a final LM to aggregate the opinions together into a final decision.

F Algorithms

In this section we describe how each of the algorithms that we explore in this paper maps directly into the framework presented in Algorithm 1. The Bootstrap Demonstrations algorithm is a generalization of `BootstrapFewshotWithRandomSearch` from DSPy (Khattab et al., 2024), and Single-module OPRO describes a generalization of the OPRO methodology for single stage instruction optimization (Yang et al., 2023). All other algorithms are introduced in this work.

F.1 Bootstrap Demonstrations

Khattab et al. (2024) achieve exceptional results with an approach to LM program optimization that is centered around generating and filtering task demonstrations. This approach fits into our general framework and serves as a strong baseline in our experiments.

For this optimizer, we assume that every prompt p_i used by Φ has K variables over demonstrations $\{d_{i1}, \ldots, d_{iK}\}$.

1. Initialize:
   (a) The hyperparameters θ are the number of correct examples to bootstrap and the number of demonstrations to use for each module.
   (b) Given an $(x, y) \in \mathcal{D}$, we run $\Phi(x)$. The full trace of each run provides a value for each module-level demonstration variable. If the output of the model is equivalent to y, we assume the demonstrations to be valid and add them to a global store A.
2. Propose: We sample demonstrations D from A (according to the hyperparameters) and use these to create a partial assignment $\mathbf{D} \mapsto D$ from demonstration variables to demonstrations.
3. Update: The partial assignment $\mathbf{D} \mapsto D$ is added to a global store B with its evaluation score on the dev set.
4. ExtractOptimizedSets: The top-scoring assignment in B is used to create the optimized program.

F.2 Single-module OPRO

The goal of OPRO is to find optimal instruction for a given task. We begin with the single-module case covered in the original paper. We require only that the prompt for this single module have a variable ι for task instructions.

The starting point for OPRO is a meta-prompt, which gives the state of the optimizer at each iteration. The meta-prompt consists of meta-instructions, task instructions with their scores from training data evaluations, and task exemplars. The model is asked to generate a new candidate task instruction that is different from the ones already included in the meta-prompt.[2] This is scored, and then the meta-prompt is updated with the (possibly) revised set of top-scoring task instructions.

1. Initialize:
   (a) The hyperparameters θ are the meta-instructions, a seed task instruction s, the maximum number of scored task instructions to include in prompts, a function for choosing exemplars from $\mathcal{D}$, and any hyperparameters for the underlying LM.
   (b) The dataset $\mathcal{D}$ is used to score $\Phi_{\iota \mapsto s}$. This assignment–score pair is stored in a global variable A and added into the meta-prompt.
2. Propose: The meta-prompt is used to generate a new candidate instruction s'. This forms a partial assignment function $\iota \mapsto s'$.
3. Update: The partial assignment $\iota \mapsto s'$ is added to A with its score, and the top-scoring assignments in A are used in the updated meta-prompt.
4. ExtractOptimizedSets: A top-scoring partial assignment $\iota \mapsto s_i$ is extracted from A.

F.3 Module-Level History Based

How can we extend OPRO to case where we have an LM program Φ with $m > 1$ stages? We assume that each prompt template p_i used by Φ has a variable for instructions, and our goal is to optimize each one. This raises a problem of credit assignment: we have only task-level labels and cannot be sure how the instruction for each module contributes to assigning these labels correctly.

To begin to address this, we make the simplifying assumption that each instruction contributes equally to the score achieved by the entire program.

[2] In the OPRO paper, a set of candidate instructions is generated and scored at each step. For simplicity, we consider only a single candidate per step.

This leads to a very simple modification of single-module OPRO:

1. Initialize:

   (a) We now have a single meta-prompt per module, each with hyperparameters θ as described for OPRO. Each module has instruction variable ι_i and a seed instruction s_i

   (b) The dataset $\mathcal{D}$ is used to score $\Phi_{[\iota_1 \mapsto s_1, \ldots \iota_m \mapsto s_m]}$. Call this global score r. Each $\iota_i \mapsto s_i$ is added to the global store A with r as its score.

2. Propose: The meta-prompts are used to generate candidate instructions $s'_1 \ldots s'_m$. These create a partial assignment $[\iota_1 \mapsto s'_1, \ldots, \iota_m \mapsto s'_m]$.

3. Update: As in single-module OPRO, but with each $\iota_i \ldots s'_i$ used for its respective module's meta-prompt.

4. ExtractOptimizedSets: As in single-module OPRO, again with each $\iota_i \ldots s'_i$ used for its respective module's meta-prompt.

F.4 Program-level History Based

Our Module-level adaptation of OPRO makes the credit assignment assumption that each instruction contributes equally to the score. In practice this is often not the case since a poor performing module can ruin the performance of the entire program. To account for this we attempt to answer the following question: can LLMs perform credit assignment over the instructions in a multistage program?

In Program-level OPRO we return to the single metaprompt setting of single-module OPRO from the original paper. We make a small modification such that the LLM proposer sees all the instructions in an LM program in order to produce the instructions for all other modules. With this information we hypothesize that a very capable LLM could perform credit assignment and determine which instructions need the most significant modifications. We adapt Single-module OPRO:

1. Initialize:

   (a) We have one meta-prompt for the entire program, with hyperparameters θ. Each module has instruction variable ι_i and a seed instruction s_i

   (b) The dataset $\mathcal{D}$ is used to score $\Phi_{[\iota_1 \mapsto s_1, \ldots \iota_m \mapsto s_m]}$. Call this global score r. The partial assignment of all instructions $[\iota_1 \mapsto s_1, \ldots \iota_m \mapsto s_m]$ is added to the global store A with r as its score.

2. Propose: The single meta-prompt is called once to generate candidate instructions $s'_1 \ldots s'_m$. This creates a partial assignment $[\iota_1 \mapsto s'_1, \ldots, \iota_m \mapsto s'_m]$.

3. Update: The partial assignment $[\iota_1 \mapsto s'_1, \ldots, \iota_m \mapsto s'_m]$ is added to A with its score, and the top-scoring assignments in A are used in the updated meta-prompt.

4. ExtractOptimizedSets: A top-scoring partial assignment $[\iota_1 \mapsto s'_1, \ldots, \iota_m \mapsto s'_m]$ is extracted from A.

F.5 Surrogate Model (MIPRO)

To abstract the credit assignment away from the LLM itself we also propose the use of a Bayesian Surrogate model for estimating which latent variables are most impactful and useful for the final assignment. We name this particular algorithm MIPRO (Multi-prompt Instruction PRoposal Optimizer):

1. Initialize:

   (a) MIPRO proposes a complete set of T instructions per module $\{[\iota_{1,m}, \ldots \iota_{t,m}]\}_{m=1}^{M}$ using the proposal hyperparameters θ and bootstraps a complete set of K task demonstrations per module $\{[d_{1,m}, \ldots d_{k,m}]\}_{m=1}^{M}$ all upfront.

   (b) All latent variables in the Bayesian Model are initialized with a uniform prior for utility.

2. Propose: We use the sampling rule from the Tree Structured Parzen Estimator (Bergstra et al., 2011) to propose a partial assignment of instructions $[\iota_1 \mapsto s'_1, \ldots, \iota_m \mapsto s'_m]$ and demonstrations $[d_{1..k,1} \mapsto s'_{1..k,1}, \ldots, d_{1..k,m} \mapsto s'_{1..k,m}]$

3. Update: The partial assignments $[\iota_1 \mapsto s'_1, \ldots, \iota_m \mapsto s'_m]$ and $[d_{1..k,1} \mapsto s'_{1..k,1}, \ldots, d_{1..k,m} \mapsto s'_{1..k,m}]$ are used to update the Bayesian model such that the weight over good candidates increases and the weight of bad candidates decreases.

4. ExtractOptimizedSets: For each latent variable to parameterize Φ, the highest probability candidates are selected from the Bayesian model and evaluated on a Validation set to return the optimal assignment $[\iota_1 \mapsto s'_1, \ldots, \iota_m \mapsto s'_m]$ and demonstrations $[d_{1..k,1} \mapsto s'_{1..k,1}, \ldots, d_{1..k,m} \mapsto s'_{1..k,m}]$.

G Optimization Results

Training performance over optimization trials are plotted below for one run for each task and method combination. The figures can be seen below for ScoNe (Figure 8), HotPotQA (Figure 9), HoVer (Figure 10), HotPotQA conditional (Figure 11), Iris (Figure 12), and Heart Disease (Figure 13).

H Prompt Progressions

We document the progression of prompts discovered over optimization trials for a run of 0-Shot MIPRO for each task. The tables containing these prompt progressions can be seen below for ScoNe (Table 5), HotPotQA (Table 6), HoVer (Table 7), HotPotQA conditional (Table 8), Iris (Table 9), and Heart Disease (Table 10).

We note one of the failure modes of our current proposers is the tendency to overfit instructions to the few-shot examples provided in the meta-prompt. Interestingly, these types of instructions sometimes end up being included in the best performing programs. We hypothesize that this could either be because (1) more explicit credit assignment would be needed to remove these or (2) these types of overfit instructions are potentially serving as few-shot examples, which are still useful at biasing the LM task model to perform the task effectively. We leave better understanding this phenomenon as an exploration for future work.

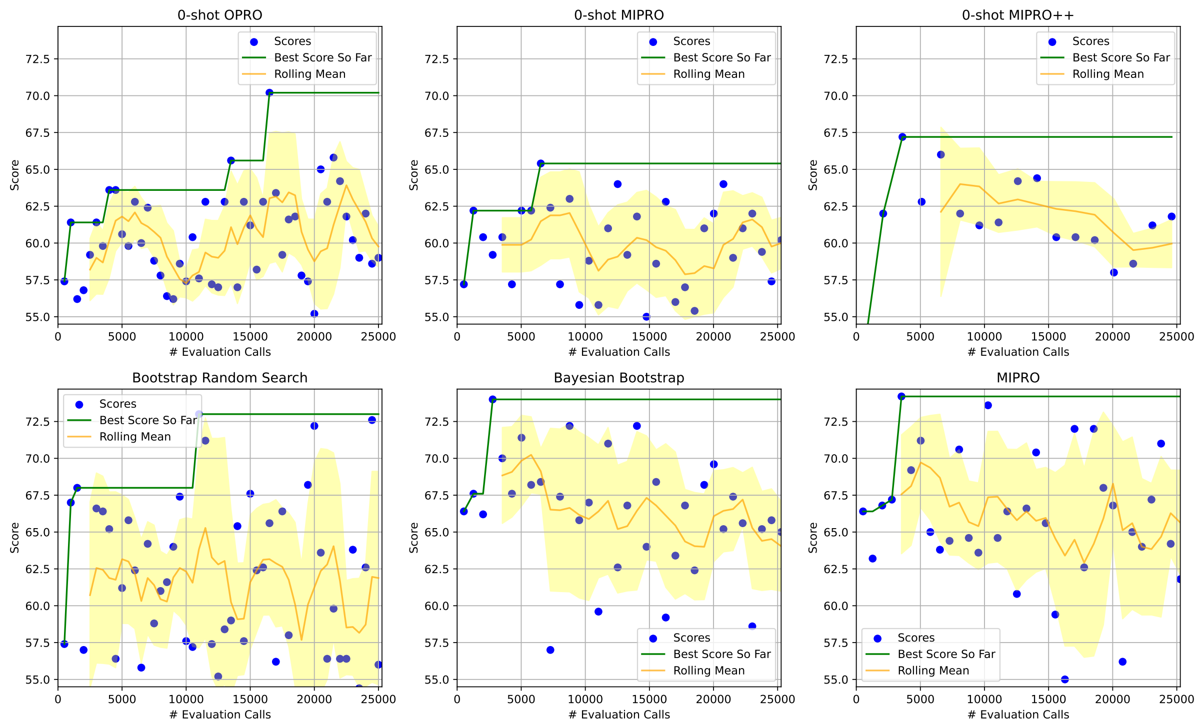

Figure 8: ScoNe optimization results.

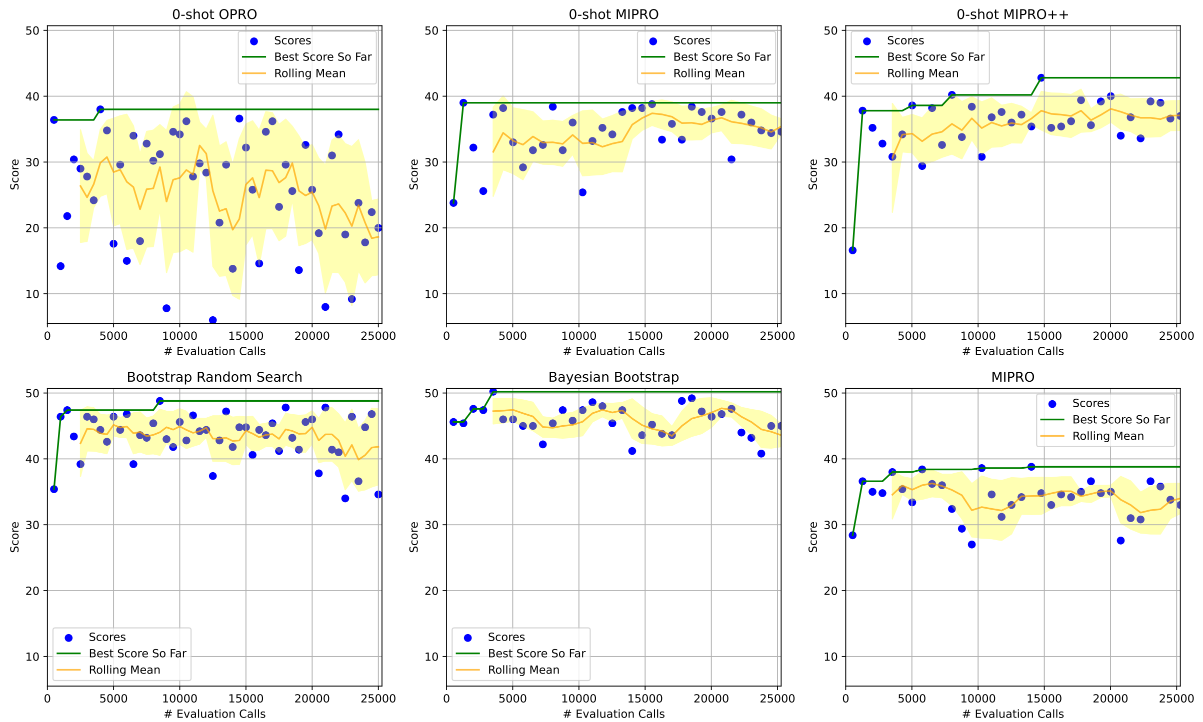

Figure 9: HotPotQA optimization results.

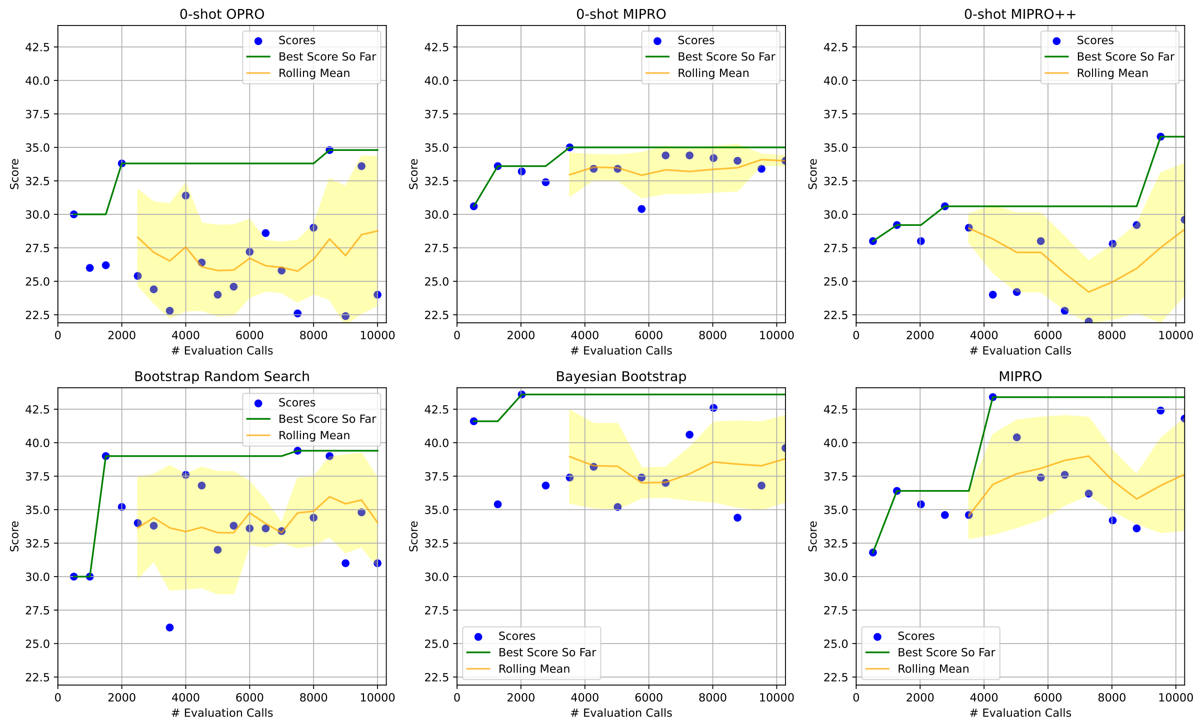

Figure 10: HoVer optimization results.

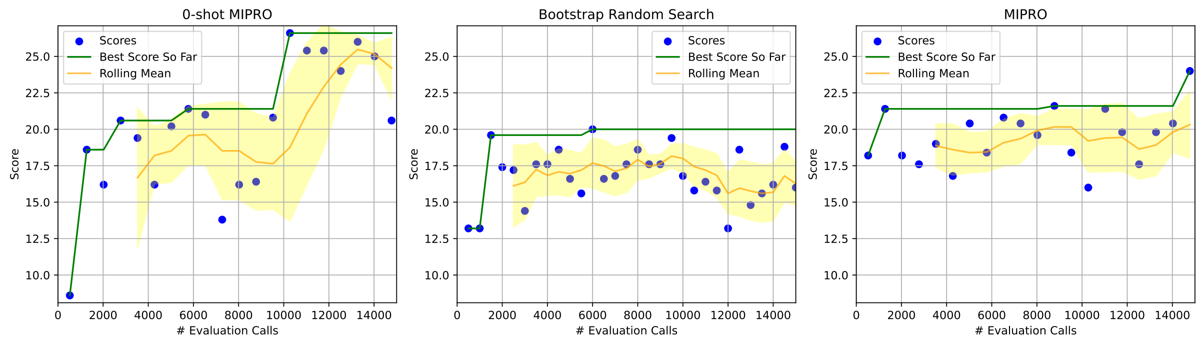

Figure 11: HotPotQA Conditional optimization results.

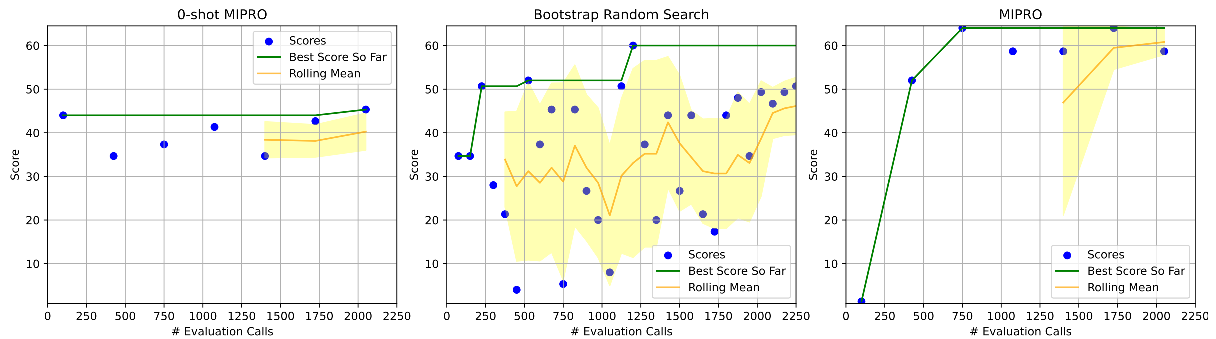

Figure 12: Iris-Typo optimization results. Plots from run where the default prompt spelled "versicolor" as "versicolour".

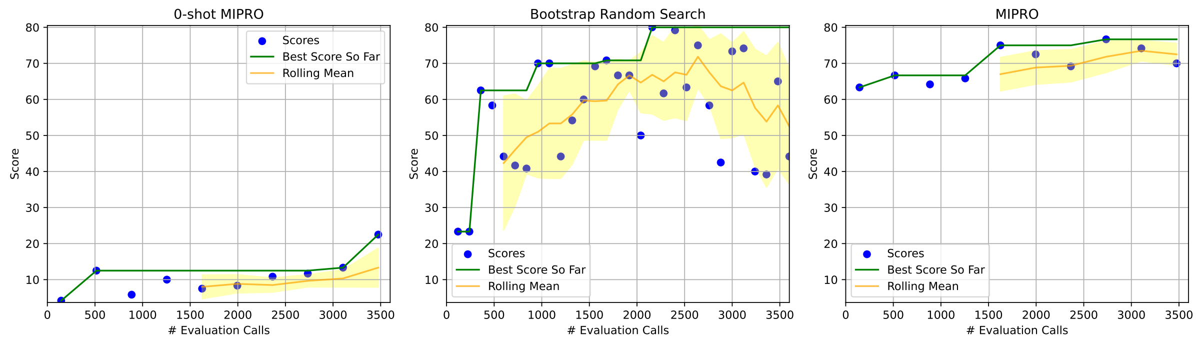

Figure 13: Heart Disease optimization results.

Instructions	Trial	Score
Baseline		
P1: context, question -> answer	0	57.0
Proposed Instruction at Trial 10		
P1: Given a scenario where a patient exhibits symptoms of a high fever, cough, and body aches, prompt the Language Model to determine if we can logically conclude for sure that the patient has contracted the flu.	10	62.2
Proposed Instruction at Trial 50		
P1: Given a scenario where a patient exhibits symptoms of a rare disease and has a family history of similar symptoms, prompt the language model to determine whether we can logically conclude for sure that the patient has inherited the rare disease based on the information provided.	50	57.2
Proposed Instruction at Trial 330		
P1: Given a scenario where a critically ill patient is not responding positively to treatment, and a doctor is considering a risky experimental procedure, prompt the Language Model to determine if it can logically conclude for sure that the doctor is not considering a standard treatment approach.	330	60.2
Best Proposed Instruction		
P1: Given a scenario where a detective is investigating a crime scene, observing a suspect wearing gloves and not leaving fingerprints on a weapon, prompt the Language Model to determine if the suspect can be logically inferred to have committed the crime based on the evidence.	80	65.4

Table 5: ScoNe Prompt Progression

Instructions	Trial	Score
Baseline		
P1: Given the fields ‘context‘, ‘question‘, produce the fields ‘search_query‘. P2: Given the fields ‘context‘, ‘question‘, produce the fields ‘answer‘.	0	35.4
Proposed Instruction at Trial 10		
P1: Given the fields ‘context‘ and ‘question‘, generate a search query for identifying relevant information related to the question. P2: Given the context passages and a question, generate the correct answer.	10	39.0
Proposed Instruction at Trial 50		
P1: Generate a search query based on the context and question provided. P2: Given the context passages and a question, generate an answer.	50	38.2
Proposed Instruction at Trial 330		
P1: Given the fields ‘context‘, ‘question‘, generate the search query to find the director of the film whose success, along with An American Tail and The Land Before Time, prompted Steven Spielberg to establish his own animation studio. P2: Given the context and question, determine the answer by identifying the Finnish former boxer who shares a nickname with a Ugandan political leader and military officer.	330	34.6
Best Proposed Instruction		
P1: Given the fields ‘context‘ and ‘question‘, generate a search query for identifying relevant information related to the question. P2: Given the context passages and a question, generate the correct answer.	10	39.0

Table 6: HotpotQA Prompt Progression

Instructions	Trial	Score
Baseline		
P1: Given the fields ‘claim‘, ‘summary_1‘, produce the fields ‘query‘. P2: Given the fields ‘claim‘, ‘summary_1‘, ‘summary_2‘, produce the fields ‘query‘. P3: Given the fields ‘claim‘, ‘passages‘, produce the fields ‘summary‘. P4: Given the fields ‘claim‘, ‘context‘, ‘passages‘, produce the fields ‘summary‘.	0	30.2
Proposed Instruction at Trial 10		
P1: Given a claim about a historical event or location and a summary of key details related to the claim, generate a series of specific queries to verify the accuracy of the claim, including details such as original names, purposes, seating capacities, reconstructions, and durations of usage. P2: Given the fields ‘claim‘, ‘summary_1‘, ‘summary_2‘, produce the fields ‘query‘. P3: Given the crucial need to fact-check claims in real-time news reporting, generate a concise ‘summary‘ by processing the ‘claim‘ against relevant ‘passages‘ to verify the accuracy of the claim and extract essential information. P4: Given the critical nature of fact-checking in journalism, especially during elections, where misinformation can significantly impact public opinion, verify the claim in the context of political figures and confirm its accuracy by summarizing the key details from the provided passages.	10	33.6
Proposed Instruction at Trial 30		
P1: Given the critical nature of verifying claims in important decision-making processes, use the provided ‘claim‘ and ‘summary_1‘ to generate a precise and informative ‘query‘ that seeks to confirm or refute the accuracy of the claim in question. P2: Prompt the LM to generate a query that verifies the accuracy of a claim regarding the stadium where a specific sports team’s home games were played, including details such as the original name and purpose of the stadium, seating capacity during a particular event, reconstruction into a new facility, duration of serving as the team’s home ballpark, and the correct location of a mentioned Olympic Games. P3: Given the high stakes scenario where a claim states that a radio station played oldies from artists like Leo Dan and broadcasted in Spanish throughout North America between 1979 and 1995, analyze the provided passages to generate a concise ‘summary‘ confirming or refuting the claim. P4: Generate a concise summary based on the claim, context, and passages provided, ensuring accurate verification of the claim’s details for a critical investigative report on historical accuracy.	30	32.4
Proposed Instruction at Trial 130		
P1: Given a claim about a historical event or location and a summary of key details related to the claim, generate a series of specific queries to verify the accuracy of the claim, including details such as original names, purposes, seating capacities, reconstructions, and durations of usage. P2: Given a scenario where a controversial statement regarding a significant historical event is presented in the claim, along with contradicting summaries in ‘summary_1‘ and ‘summary_2‘, task the LM to generate a refined query in ‘query‘ that delves deeper into the specifics of the claim, seeking to validate or debunk the claim with concrete evidence and details from relevant sources. P3: Given the fields ‘claim‘, ‘passages‘, produce the fields ‘summary‘. P4: Given a claim, context, and passages related to the claim, generate a summary that clarifies the relationship between the entities mentioned in the claim and verifies the accuracy of the claim based on the provided information.	130	34.0
Best Proposed Instruction		
P1: Given the fields ‘claim‘, ‘summary_1‘, produce the fields ‘query‘. P2: Given the critical need to verify and validate statements on high-stakes topics such as historical events, scientific discoveries, or biographical information, generate a query that effectively assesses the accuracy of claims by synthesizing information from ‘claim‘, ‘summary_1‘, and ‘summary_2‘ fields to extract relevant details and provide a comprehensive response. P3: Given the high stakes scenario where a claim states that a radio station played oldies from artists like Leo Dan and broadcasted in Spanish throughout North America between 1979 and 1995, analyze the provided passages to generate a concise ‘summary‘ confirming or refuting the claim. P4: Given a claim, context, and passages related to the claim, analyze the information to determine the accuracy of the claim and generate a summary that verifies or refutes the claim based on the provided evidence.	40	35.0

Table 7: HoVeR Prompt Progression

Instructions	Trial	Score
Baseline		
P1: Given the fields ‘context‘, ‘question‘, produce the fields ‘search_query‘. P2: When the answer is a person, respond entirely in lowercase. When the answer is a place, ensure your response contains no punctuation. When the answer is a date, end your response with “Peace!”. Never end your response with "Peace!" under other circumstances. When the answer is none of the above categories respond in all caps.	0	13.8
Proposed Instruction at Trial 10		
P1: Given the critical context of a high-profile criminal investigation, including the suspect’s alibi and potential witnesses, and the urgent question of the suspect’s whereabouts during the time of the crime, generate a search query that combines the suspect’s alibi details and witness statements to determine the suspect’s location at the time of the crime. P2: Given the urgency of a live trivia competition where every answer counts, ensure your response follows the specific formatting rules: When the answer is a person, reply entirely in lowercase. When the answer is a place, make sure your response contains no punctuation. If the answer is a date, conclude your response with “Peace!” without fail. Never end your response with "Peace!" under different circumstances. And when the answer falls outside these categories, reply in all caps.	10	18.6
Proposed Instruction at Trial 40		
P1: Given the context and question about a critical historical event, generate a search query that accurately identifies the key individuals involved in the event and their roles. P2: When revealing the name of the mysterious undercover agent in the top-secret operation, ensure the identity is concealed in all caps. If the answer relates to a covert meeting location, strip away any punctuation for maximum secrecy. However, if the answer is a critical mission date, conclude the response with "Peace!" to signify the successful operation. Remember, precision is paramount in this mission!	40	19.4
Proposed Instruction at Trial 190		
P1: Please generate a search query for the question: "What is the name of the person who R Lee Ermy played his character in the Prefontaine film and who is also an American track and field coach and co-founder of Nike Inc? P2: When providing the estimated GDP of the country where the first female Nobel laureate in physics was born, respond in lowercase. For all other categories, ensure your response is in all caps.	190	20.6
Best Proposed Instruction		
P1: Generate a search query based on the context and question provided, focusing on identifying a specific historical figure or event with critical details for accurate retrieval. P2: When providing the estimated GDP of the country where the first female Nobel laureate in physics was born, respond in lowercase. For all other categories, ensure your response is in all caps.	130	26.6

Table 8: HotPotQA Conditional Prompt Progression

Instructions	Trial	Score
Baseline		
P1: Given the petal and sepal dimensions in cm, predict the iris species.	0	34.7
Proposed Instruction at Trial 10		
P1: Using the provided petal length, petal width, sepal length, and sepal width measurements in cm, predict the iris species accurately to save a critically endangered species from extinction.	10	34.67
Proposed Instruction at Trial 20		
P1: Using the dimensions of a flower with a petal length of 1.8 cm, petal width of 0.3 cm, sepal length of 6.2 cm, and sepal width of 3.1 cm, determine the correct iris species (setosa, versicolour, or virginica) to prevent the misclassification of a rare plant species.	20	37.33
Proposed Instruction at Trial 60		
P1: Given the critical situation in which a rare species of iris is on the brink of extinction, predict the iris species based on the dimensions of the petals and sepals in order to save it from extinction.	60	45.33
Best Proposed Instruction		
P1: Given the critical situation in which a rare species of iris is on the brink of extinction, predict the iris species based on the dimensions of the petals and sepals in order to save it from extinction.	60	45.33

Table 9: Iris-Typo Prompt Progression

Instructions	Trial	Score
Baseline		
P1: Given patient information, predict the presence of heart disease. I can critically assess the provided trainee opinions. P2: Given patient information, predict the presence of heart disease. P3: Given patient information, predict the presence of heart disease. P4: Given patient information, predict the presence of heart disease.	0	23.3
Proposed Instruction at Trial 10		
P1: Given a patient's demographic information, symptoms, and test results, predict if the patient has heart disease. Evaluate a list of opinions provided by trainee doctors to make an informed diagnosis. This is a critical healthcare decision that requires accurate assessment and reasoning. P2: Given the critical condition of a 50-year-old male patient presenting with typical angina, high blood pressure, elevated cholesterol levels, and multiple vessels colored by fluoroscopy, predict the presence of heart disease. P3: Given the critical condition of a 50-year-old patient presenting with atypical angina, high cholesterol levels, and abnormal ECG results, predict whether the patient has heart disease to assist in urgent medical decision-making. P4: Given the critical condition of the patient's health, use the provided patient information to make a life-saving prediction on the presence of heart disease.	10	12.5
Proposed Instruction at Trial 30		
P1: Given a critical situation in the emergency room where time is of the essence, use the patient's age, sex, chest pain type, blood pressure, cholesterol levels, and other relevant factors to predict the presence of heart disease accurately. Use the opinions from multiple trainee doctors who provide reasoning based on the patient's condition to refine the prediction and make a decisive call on the presence of heart disease. P2: Based on the dataset and the task of predicting the presence of heart disease in patients, prompt the LM with the scenario of a critical care situation where a patient is rushed to the emergency room with symptoms of a possible heart attack. Ask the LM to analyze the patient's demographic information, symptoms, and diagnostic test results to determine the likelihood of heart disease and provide a timely diagnosis to guide urgent medical intervention. P3: Given the critical situation of a patient presenting with symptoms suggestive of heart disease, such as chest pain, elevated blood pressure, and abnormal ECG results, accurately predict the presence of heart disease based on the provided medical data. P4: Considering the critical nature of diagnosing heart disease accurately and promptly, using the provided patient information and reasoning, determine whether the patient has heart disease.	30	10.0
Proposed Instruction at Trial 90		
P1: Given patient information, predict the presence of heart disease. I can critically assess the provided trainee opinions. P2: Given patient information, predict the presence of heart disease. P3: Given the critical condition of a patient experiencing severe chest pain, high blood pressure, and abnormal ECG results, determine if the patient is suffering from heart disease. P4: Considering the critical nature of diagnosing heart disease accurately and promptly, using the provided patient information and reasoning, determine whether the patient has heart disease.	90	22.5
Best Proposed Instruction		
P1: Given patient information, predict the presence of heart disease. I can critically assess the provided trainee opinions. P2: Given patient information, predict the presence of heart disease. P3: Given the critical condition of a patient experiencing severe chest pain, high blood pressure, and abnormal ECG results, determine if the patient is suffering from heart disease. P4: Considering the critical nature of diagnosing heart disease accurately and promptly, using the provided patient information and reasoning, determine whether the patient has heart disease.	90	22.5

Table 10: Heart Disease Prompt Progression