Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Artificial Intelligence Center

Meta-prompts for LLM Prompt Optimization

abcd

Vojtěch Klouda

Supervisor: Ing. Jan Drchal PhD.
Field of study: Artificial Intelligence
Subfield: Natural Language Processing
May 2025

Acknowledgements

=)))

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 10. May 2025

Abstract

TODO

Keywords: language model, optimization

Supervisor: Ing. Jan Drchal PhD.
Resslova 307/9 Praha, E-322

Abstrakt

TODO

Klíčová slova: jazykový model, optimalizace

Překlad názvu: abcd — abcd

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Background

In recent years, Large Language Models (LLMs) have permeated the Natural Language Processing research landscape as well as into the general public. Already achieving human-like performance at a wide variety of tasks[2], they are bound by scaling laws[3] which predict performance gained with adding compute, fueling massive investments into computation capacity by industry players.

With costs of training new state-of-the-art foundational LLMs rising rapidly, research has turned to inference-time scaling[4], based on post-training[5][6] utilizing reinforcement learning and supervised fine-tuning, and prompting techniqes[7]. Another research branch gaining substantial attention recently is compile-time scaling[8] represented by prompt optimization[9]. Optimization using LLMs and particularly prompt optimization presents an exciting intersection between traditional algorithms and deep learning.

1.2 Problem Formulation

Let $\mathcal{T}$ be the space of text sequences. Then we will define an LLM as a stochastic mapping

$$\mathscr{M} : \mathcal{T} \to \mathcal{L}(\mathcal{T}), \tag{1.1}$$

where $\mathcal{L}(\mathcal{T})$ is a probabilistic language distribution learned during the LLM's training. This distribution is governed by the LLM's hyperparameters H, which affext its behaviour. Of particular interest is the sampling temperature $t \in H$ which interpolates between greedy decoding and uniform sampling. In theory, $\mathscr{M}$ is deterministic for $t = 0$, but in practice numerical errors still introduce variance.

A prompt $P \in \mathcal{T}$ is a text sequence that, when input into an LLM, produces an output

$$y \sim \mathscr{M}(P). \tag{1.2}$$

In contexts where P serves as a template for an additional query q, we will write

$$y \sim \mathscr{M}(P \,|\, q), \tag{1.3}$$

where $P \,|\, q$ denotes the result of inserting query q into a designated placeholder in P.

We can use LLMs to solve a general task

$$t = (q, g) \in \mathcal{D}, \tag{1.4}$$

where $\mathcal{D} \subseteq \mathcal{Q} \times \mathcal{G}$ is a dataset of query-answer pairs, $\mathcal{Q} \subseteq \mathcal{T}$ is the set of queries q and $\mathcal{G} \subseteq \mathcal{T}$ is the set of gold labels g. We consider each dataset to have one or more assigned evaluation metrics $\mathscr{F}_\mathcal{D} : \mathcal{T} \times \mathcal{G} \to \mathbb{R}$, which scores the LLM output using the corresponding gold label.

For open-ended tasks, the gold label does not exist, $G = \varnothing$. To achieve effective evaluation even for such tasks, we formulate a metric based on pairwise comparisons and generalize the definition as

$$\mathscr{F}_\mathcal{D} : \mathcal{T} \times \mathcal{G} \times 2^\mathcal{T} \to \mathbb{R}, \tag{1.5}$$

where $2^\mathcal{T}$ denotes the powerset of the set of text sequences. Using this definition, we define the mean perfomance $\mathcal{E}$ of a prompt P on a dataset $\mathcal{D}$ as

$$\mathcal{E}_\mathcal{D}(P) = \frac{1}{\|\mathcal{D}\|} \sum_{(q,g)\in\mathcal{D}} \mathscr{F}(\mathscr{M}(P \,|\, q), g, \mathcal{C}_q), \tag{1.6}$$

where $\mathcal{C}_q$ is a set of reference outputs. This set consists of past generations using the same query. Let $\mathcal{P} \subseteq \mathcal{T}$ be a population of prompts. Then

$$\mathcal{C}_q = \{\mathscr{M}(P \,|\, q) \,|\, P \in \mathcal{P}\}. \tag{1.7}$$

For convenience, we will define the scores of the population $\mathcal{P}$ on dataset $\mathcal{D}$ as

$$\mathcal{E}_\mathcal{D}(\mathcal{P}) = \{\mathcal{E}_\mathcal{D}(P) \,|\, P \in \mathcal{P}\}. \tag{1.8}$$

We can then formally define the problem of prompt optimization for a task dataset $\mathcal{D}$ as finding the optimal prompt

$$P^\star = \operatorname*{argmax}_{P\in\mathcal{T}} \mathbb{E}_{(q,g)\sim\mathcal{D}} \left[\mathscr{F}(\mathscr{M}(P \,|\, q), g, \mathcal{C}_q)\right]. \tag{1.9}$$

In Algorithm 1 we can see the general outline of a population-based optimization method. The initialization operator $\mathscr{O}_I$ creates an initial population of individuals $\mathcal{P}$. In each iteration, a selection operator $\mathscr{O}_S$ first selects a portion of the population according to some criteria. These selected individuals are then used by the expansion operator $\mathscr{O}_E$ to create new individuals. This process continues until a termination condition Φ_{stop} is reached.

We apply this technique to the problem of prompt optimization by defining the aforementioned operators. Of particular interest are the initialization

Algorithm 1: General optimization loop

Input: Initialization Operator $\mathscr{O}_I$, Selection Operator $\mathscr{O}_S$, Expansion Operator $\mathscr{O}_E$, Termination Condition Φ_{stop}
Output: Optimized Population $\mathcal{P}$
Data: $\mathcal{P} \leftarrow \mathscr{O}_I$
`// Initialize the population`
while $\neg\Phi_{stop}(\mathcal{P})$ **do**
  `// Selection and Expansion Steps`
  $\mathcal{P}_{\text{selected}} \leftarrow \mathscr{O}_S(\mathcal{P})$
  $\mathcal{P}_{\text{expanded}} \leftarrow \mathscr{O}_E(\mathcal{P}_{\text{selected}})$
  $\mathcal{P} \leftarrow \mathcal{P}_{\text{expanded}}$ `// Update the population`
return $\mathcal{P}$ `// Return the optimized population`

operator $\mathscr{O}_I$ and the expansion operator $\mathscr{O}_E$, which both need to produce new prompts. For this purpose, we utilize an LLM instance $\mathscr{M}_{\text{optim}}$ and leverage its text generation and reasoning capability. By using the lower index, we specify the purpose of the LLM and differentiate from another instances, which might use different hyperparameters.

Let $M \in \mathcal{T}$ be a *Meta-prompt*, or a prompt-generation prompt. We can now formulate generating a prompt

$$P = \mathscr{M}_{\text{optim}}(M \,|\, \mathcal{R})), \tag{1.10}$$

where $\mathcal{R} = \mathcal{R}(\mathcal{P}, \mathcal{C}, \mathcal{D}, \mathcal{E}_{\mathcal{D}}(\mathcal{P}))$ is a retrieval function that selects data from the current population, past generations, dataset samples and population scores. By changing the *Meta-prompt* and the retrieval function $\mathcal{R}$, a variety of possible operators $\mathscr{O}_I$ and $\mathscr{O}_E$ can be defined, thus shaping the prompt optimization process.

Chapter 2

Literature

2.1 Inference-time scaling

Inference-time scaling or test-time scaling is a paradigm that has gained traction in recent years with the advent of dedicated reasoning models[5][6]. As opposed to training-time scaling, where the performance of models scales with training times, model parameter counts and dataset sizes[3], inference-time scaling aims to improve performance by dedicating more resources to each inference call.

At their heart LLMs are probabilistic models over sequences and to generate a sequence they employ generation algorithms. Welleck et al.[4] provide an overview of these generation algorithms and then frame more advanced inference-time techniques as meta-generations, or strategies that employ sub-generators. Most generation algorithms attempt to find either highly probable sequences (MAP algorithms) or sample from the model's distribution. The simplest MAP algorithm is greedy decoding, which recursively finds the next token with the highest probability in the distribution. An example of algorithms that sample from the model's distribution is the ancestral sampling algorithm.

A generalization of greedy decoding is the beam search algorithm which maintains a structure of possible prefixes and each step expands them and scores them. An example[10] of a beam search algorithm can identify decoding branches where the model employs a reasoning chain to solve a given task. Authors of this algorithm found that answer tokens found in the decoding paths with a reasoning chain have greater token probabilities. This means that the model shows greater confidence in its answer having reasoned about it beforehand. In general beam search improves on simple greedy decoding but at a high computational cost[4].

Another class of generation algorithms are those which interpolate between more categories of sampling algorithms. Temperature sampling, which outperforms other adapters in input-output tasks like code generation and translation, is an interpolation between greedy sampling and uniform sampling. Interpolating between ancestral sampling and simple greedy sampling gave rise to decoding algorithms such as nucleus, top-k and η- and ϵ-sampling. When we require a structured output, for example a JSON data structure following a JSON schema, we can utilize parser-based decoding, which enforces a structural requirement. This can however come at worsened performance when using inflexible templates.

These low-level generator can be interconnected into more complex technique, which Welleck et al. call meta-generators[4]. We will stick to their terminology and discuss different sequence-level meta-generation algorithms. We will omit further discussion of token-level methods as they are irrelevant to the main topic of this thesis. These strategies can be divided into the categories of chained, parallel, step-level, and refinement-based meta-generators.

2.1.1 Chained meta-generation

Chained meta-generation is the composition of several subgenerators in sequence. These can be LLM calls or other functions that use previous inputs, such as a code execution function[11] or a tool for interaction with and arbitrary environment or a data source[12]. The subgenerators can be implemented as several LLM calls or with a single call given sufficient instructions in the prompt. [13] Some examples include Program of Thoughts[11], ReAct[12] and Chain-of-Thought[14][15] techniques.

In its essence, the model is a left-to-right text completion engine. We can make the analogy with human thinking modes, where it is said that humans have a fast automatic "System 1" mode and a slow and deliberate "System 2" mode[16]. In direct-QA mode, the LLM can underestimate the difficulty of the task[10] and stay in the "System 1" thinking mode. Simple greedy decoding paths mostly do not contain a reasoning chain[10], which means the model tends to make a guess, staying in "System 1". By crafting a good prompt that instructs the model to reason we can shift the model from "System 1" to "System 2" thinking. Another reason for the effectiveness of chained generation is that in LLM training some concepts and variables are observed more frequently than others[17]. This discrepancy hurts performance in direct-QA scenarios where the relevant variables are rarely seen together in training. With CoT, models can incrementally chain known dependencies and bridge conceptional gaps.

Chain-of-thought

Chain-of-Thought[15] (CoT) is a LLM prompting technique that works by inducing a coherent series of intermediate reasoning steps that lead to the final answer for a problem, thereby increasing computation time. Upon its discovery, it brought a dramatic performance increase on arithmetic tasks, where models previously struggled. This enhanced capability comes with the cost of longer and more computationally expensive outputs[18] and is more noticeable for more complicated problems[15].

CoT can been elicited by prompting techniques - few-shot with steps demonstrations or zero-shot with specific instructions[10] First CoT methods[15] involved one/few-shot prompting, Although effective, this requires human engineering of multi-step reasoning prompts. This method is also highly sensitive to prompt design with performance deteriorating for mismatched prompt example and task question types[14]. For this method, authors found that CoT is an emergent capability of model scale and did not observe benefits for small models[15]. where the prompt included examples of CoT reasoning in the prompt in facilitate a reasoning chain response.

On the other hand, zero-shot prompting induces a reasoning chain with a simple prompt like "Let's think step-by-step", making it versatile and task-agnostic[14]. Similar prompts also improve reasoning performance and some research[1] has been done on finding the optimal CoT prefix prompt.

Apart from prompting, CoT can be elicited and improved by model training or tuning. This method, requiring a significant amount of reasoning data[10], has gained traction with the development of dedicated reasoning models like OpenAI's o1[5] or Deepseek-R1[6]. Using methods such as supervised fine-tuning (SFT) or reinforcement learning (RL), the model is trained to automatically produce longer reasoning chains, often bound in dedicated "thought" tags or tokens. These models have shown significant performance boosts on reasoning benchmarks[5][6].

Models similar to o1 all primarily extend solution length by self-revision[19]. After finishing a thought process, the model tries to self-revise, which is marked by words such as "Wait" or "Alternatively". The model then tries to spot mistakes or inconsistencies in its reasoning or propose an alternative solution. Self-revision ability is thus a key factor in the effectiveness of sequential scaling for reasoning models[19].

Longer reasoning chains mean more computing power spent at inference. How far can we take this sequential scaling? In their study, Zeng et al.[19] argue that longer CoTs do not consistently improve accuracy of reasoning models. Furthermore, they find that the average length of correct solutions is shorter than that of incorrect ones.

Because self-revision accounts for most of the CoT length, the effectiveness of the method relies on the model's ability to self-revise. Authors of this paper argue that the self-revision ability of models is insufficient as they demonstrate limited capacity to correct their answers during self-revision. Some models on some tasks are even more likely to change a correct answer to an incorrect one than vice-versa.

Further research by Liu et al.[20] suggests that for some tasks CoT can be detrimental. Their experiments proved their hypothesis that CoT hurts performance on tasks where humans do better without deliberation and where the nature of LLM, like the much greater context memory, does not provide an advantage over human thinking. This phenomenon was observed on tasks like facial recognition, implicit statistical learning or pattern recognition.

2.1.2 Parallel meta-generation

Parallel meta-generation involves multiple generations concurrently. The final answer can then be chosen - with a reward model or with voting - or constructed from the ensemble of generations[4].

Parallel meta-generation allows weaker models to outperform bigger and more expensive models[18]. This can sometimes reduce cost as multiple samples with a smaller model are cheaper than a single sample with a more capable model. This is helped by the fact that parallel sampling can make use of batching and other system throughput optimization available for parallel inference[18].

One of the simplest such techniques is self-consistency[21] (SC), a method which builds upon CoT to aggregate answers from diverse reasoning chains and selects the best one based on majority voting. It significantly improves accuracy in a range of arithmetic and commonsense reasoning tasks at the cost of increased computation expenditure[21]. The effectiveness of SC with majority-voting comes from the fact that, for tasks with objective answers, there are often more ways to be wrong than to be right.

For our next discussion of SC and related methods we will compare the terms *coverage* $C_{\mathbb{D}}$ and *accuracy* $A_{\mathbb{D}}$ for a dataset $\mathbb{D}$. Given a language model $\mathcal{M}$, a task query $q_k \in \mathbb{D}$ and a task instruction $\mathbf{i}$, we can define the generation collection of length n as

$$Y_k = \{y_{jk} \mid j \in 1, ..., n\}, \tag{2.1}$$

$$y_{jk} \sim \mathcal{M}(\mathbf{i}(q_k)). \tag{2.2}$$

For objective tasks we can check the correctness with a metric $\mathcal{G}$

$$\mathcal{G}_k(y_{jk}, q_k) = \begin{cases} 1.0 & y_{jk} \text{ is the correct answer for } q_k \\ 0.0 & y_{jk} \text{ is an incorrect answer for } q_k. \end{cases} \tag{2.3}$$

To choose the final answer, we will define an answer selection function $\mathcal{S}(Y)$. This can be a majority vote selection function or some reward-based method. We can now define *coverage* $C_{\mathbb{D}}$ and *accuracy* $A_{\mathbb{D}}$ as

$$C_{\mathbb{D}} = \frac{1}{|\mathbb{D}|} \sum_{q_k \in \mathbb{D}} \max_{j=1,...,n} \mathcal{G}_k(y_{jk}, q_k) \tag{2.4}$$

$$A_{\mathbb{D}} = \frac{1}{|\mathbb{D}|} \sum_{q_k \in \mathbb{D}} \mathcal{G}_k\left(\mathcal{S}(Y_k), q_k\right). \tag{2.5}$$

In plain language, coverage is the fraction of the tasks where at least one sample results in a correct answer, whereas accuracy is the fraction of the tasks where a correct answer is selected by the algorithm as a final answer.

It is easy to see why coverage might rise as we increase the amount of samples n in SC generation. One could imagine that as letting students answer with their top n guesses for each question on a test. Indeed research[18] has found that the relationship of coverage and the number of samples can be modeled by an exponentiated power law, suggesting a scaling law for inference similar to the training scaling laws[3].

However coverage alone is not enough to paint the complete picture. What good is it to have a large collection which contains a correct answer if we cannot verify which one is correct. Parallel scaling with large sample collections is only useful if the correct samples in a collection can be identified[18][19]. The accuracy gain of SC tends to saturate quickly as we increase the number of paths[21]. Although coverage rises, it diverges[18] from accuracy as the algorithm is unable to select the correct answer from the collection. This highlight the necessity to develop better answer selection mechanisms than simple majority voting and automatic answer verification methods.

Zeng et al.[19] make use of the fact that correct solutions have shorter CoT on average and develop a length-weighted majority vote that outperforms simple majority voting on the challenging math benchmarks. GLaPE[22] is a method for gold label-agnostic evaluation which makes use of the fact that incorrect answers tend to be inconsistent.

2.1.3 Step-level meta-generation

Maintaining the terminology of Welleck et al.[4], step-level meta-generation algorithms implement search on the generation state-space. This can be done on the token level or on the level of longer sequences, but in this section we will focus on the latter.

Previously discussed inference-time scaling techniques all relied on sequential or parallel linear thought processes. They do not explore different continuations within a thought process and do not make use of planning, lookahead, or backtracking[16]. These methods also do not allow combining the flow of reasoning upon discovering new insights, something humans utilize when solving problems[23].

By generalizing CoT[15][14] into a tree structure, Yao et al.[16] present Tree of Thoughts (ToT), a technique which maintains a tree of thought. In this tree, each node is a thought in a form of a coherent language sequence, serving as an intermediate step in the reasoning process. For traversing the tree, a general tree-search algorithm, such as breadth-first or depth-first search, can be employed.

An important parameter in ToT is the branching factor. Unlike the standard tasks typically tackled by tree search algorithms where the number of possible actions at each node is finite, each call to LLM can yield a new output even for the same input, making each node's branching factor theoretically infinite[24]. Misaki et al.[24] argue that fixed-width multi-turn methods exhibit diminishing gains and develop a tree search method with an adaptive branching factor, leading to a more balanced exploration and exploitation capability.

Although ToT allows for planning and backtracking from unpromising thought chains, its structure is still too rigid[23]. For example, it is not possible to combine thoughts from independent branches from the tree. Graph of Thoughts[23] (GoT) is a framework that models the reasoning process as a heterogenous directed graph where each vertex is a thought containing a (partial) solution and edges are dependencies between these thoughts[23].

In both chain- and tree-based inference-time scaling methods, a substantial amount of compute power is allocated to processing historical information that is not beneficial to the reasoning process. To alleviate this, Atom of Thoughts[25] (AoT) iteratively decomposes the current question into a directed acyclic graph. The graph consists of subquestions which, depending on whether they have dependencies, are dependent or independent. All the independent questions can be answered directly and their answers added combined as context with the remaining subquestions to be contracted into a new current question.

2.1.4 Refinement meta-generation

The last category of meta-generation algorithms are refinement algorithms. Refinement algorithms work by alternating between generation and refinement. The refiner generates a revised version of the output based on past versions and additional information such as intrinsic or extrinsic feedback or environment observations[4]. Intrinsic refinement comes from the model inspecting its own answers. As we discussed in 2.1.1, models struggle with self-revision and rarely modify their answers in long reasoning chains. Feedback from general models is ineffective compared to dedicated feedback models or other quality feedback sources[26].

For extrinsic refinement, the model can utilize external information which can lead to a potential gain with refinement[4]. One example of a refinement-based framework, Reflexion[27], converts binary or scalar feedback from the environment into verbal feedback in the form of a textual summary. This feedback is then added as additional context for the LLM agent, e.g. CoT or ReAct module, in the next episode. Reflexion improves performance over strong baselines on sequential decision making, reasoning and programming tasks[27].

2.2 Prompt engineering

By prompt engineering we mean crafting a instruction set which transforms the query into a result according to our task requirements. Our task requirements can for example be

- obtaining the correct answer for a mathematical problem
- fixing a bug in a code base
- explaining the contents of an image.

Each of these tasks needs a separate instruction set **i** which can then be used with multiple queries, representing specific task instances. This signifies a shift from the training and fine-tuning paradigm, where a base model is first trained on a large corpus of data and then adapted for a specific task with supervised fine-tuning. This process requires a substantial amount of training data and computation power, making specialized LLMs unsuitable for many users and for applications, where extensive data collection is infeasible.

Since the inception of modern LLMs, prompt engineering has evolved into a field of its own. Current LLM systems, often containing multiple chained and interlinked models, require robust and well thought-out prompts at each step. Indeed, in many modern LLM applications, prompts have become programs themselves[8], marking a huge leap from the basic text messages of the early LLM days.

In this section, we will briefly cover the most notable prompt engineering techniques, which we will then be able to utilize in our study of automatic prompt optimization.

2.2.1 Components of a prompt

We can dissect a prompt into several components[7].

- **Directive:** The main task of the prompt, e.g., *"Write an email to a coworker."*
- **Context:** Everything necessary or beneficial to completing the directive, e.g., *"I was supposed to send a report to my boss, but I forgot."*
- **Examples:** How you would have solved a similar task, e.g. a past email on a similar topic.
- **Output specifications:** Style and format instructions, e.g., *"Respond with three paragraphs in formal style with tasteful emojis."*

Although flexible and sometimes blended together, high-performing prompts often follow this structure and order. In more technical applications, it is beneficial to use tags or delimiters to explicitly separate components. Models often reward prompts with a more code-like structure[28].

We now discuss each component in detail.

Directive

The directive should be a clear and objective description of the task, assuming that the model already knows how to solve it[29]. Specific requirements that narrow the scope should be avoided and left for other components.

In cases where the prompt serves as a prompt template, meaning it can be reused with various data points, the directive can include a placeholder. For example consider this directive

2.2.1 Directive with a placeholder

Write a limerick about {`topic`}.

This directive, and the prompt to which it belongs, could be reused for multiple values of `topic`.

Context

Context should provide all the background information relevant to the task at hand. The user can include more information about why they are using the model for the task, define the target audience or attach relevant documents. For example, a prompt that asks the LLM to summarize an article:

2.2.2 Using context to personalize output

Directive:

> Summarize this article on climate change for a high-school debate. `{article}`

Context:

> I already understand the basic causes of climate change, but I struggle with the economic side. Focus on how it affects economies and give arguments I could use in a policy debate.

In this example, the initial instruction serves as the directive, while the second component provides contextual information about the user's prior knowledge and objectives. Without this context, the model may produce a generic summary, overlooking the user's interest in economic impacts and failing to surface arguments relevant for a policy-oriented debate.

The context can become the endpoint for retrieval pipelines, which search a data source for relevant documents, or memory mechanisms, which gather personal information about the user from other conversations.

Another possible feature of the context component is the use of a memetic proxy[29], like role-assignment. Instead of writing a long instruction covering all the requirements and assumptions behind someone being an "experienced business analyst", we can just say "You are an experienced business analyst". In this way, we can instruct the model to adopt an identity or an expertise level. This primes the model to use a more technical language in its response.

Examples

By providing examples of solutions to similar tasks, we can condition the LLM to generate further examples from that distribution[30], increasing the chance of a suitable completion. Furthermore, examples make the decoding more robust and decrease prompt sensitivity[31]. Table 2.1 shows the agreed-upon

Prompting Type	**Description**
Zero-shot Prompting	Prompt has no examples. Model relies on instructions and pre-trained knowledge.
One-shot Prompting	Prompt has one example to guide the model.
Few-shot Prompting	Prompt includes a few examples.

Table 2.1: Comparison of Zero-shot, One-shot, and Few-shot Prompting

terminology for prompts with examples. The concept of adding examples to the prompt is also called In-Context Learning (ICL). In some cases, Few-shot prompting can be effective even without the use of other instructions[32]. When adding examples to a prompt, we have to pay attention to several aspects[7].

- **Exemplar quantity**: More is better with diminishing returns.
- **Exemplar ordering**: Models tend to pay more attention to the last examples.
- **Exemplar label distribution**: Unbalanced labels in examples skew model generation.
- **Exemplar label quality**: It is unclear whether incorrect examples hurt performance.
- **Exemplar format**: Optimal format may vary across tasks.
- **Exemplar similarity**: Effect of exemplar similarity depends on the situation.

Contrary to Brown et al.[32] who interpret the effectiveness of few-shot prompting as the model learning the task by observing examples, later research[29] suggestes that examples merely allow the LLM to locate the task more precisely in its learned task space. Still, the authors argued for the use of examples for redundancy enforcing the desired behavior[29].

Output specifications

With this component, we guide the structure, tone, style and formatting of the LLM's answer. A common technique, discussed in 2.1.1 is inducing reasoning with a CoT prompt, like "Let's think step-by-step", or instructing the LLM to first plan the solution and then execute it.

We can influence the length of the answer, ask the model to be formal or humorous, request specific formatting, like the use of LaTeX equations, or have it answer in a JSON format for a machine-readable response. Users should test multiple configurations as changing the output format can influence the final prediction[33].

2.2.2 Meta-prompting

Before we proceed we first need to overview the terminology discrepencies in contemporary research. Meta-prompts (also meta prompts or metaprompts) were first coined by Reynolds and McDonell[29]. Since then, this term was used in multiple contexts.

1. **Task-agnostic zero-shot prompt:** In contrast to task-sample specific few-shot prompting, meta-prompts were used to mean a 'task-agnostic zero-shot prompt', or 'seeds encapsulating a more general intention that will unfold into a specific prompt when combined with the task question[29].
2. **Natural language metaprocedure:** Extending 1, Zhang et al.[34] define meta-prompts as example-agnostic prompts designed to capture the reasoning structure of a specific category of tasks. They do this by employing a typed and structured prompt, resembling control flow templated and often expressed in JSON-like structures. This is similar to how DSPy[13] implements LLM calls with function signatures.

3. **Soft prompt optimization method:** In a parallel branch of research, MetaPrompting[35] was used as a name of a soft token prompt optimization method.
4. **Prompt generator:** In automatic prompt engineering and prompt optimization literature, meta-prompt is a prompt that generates prompts[36]. This can be seen as application of 2 to the task of prompt generation[34].

While both 1 and 2 are of interest to us, in this thesis, we will treat meta-prompts as prompt generators.

Meta-prompt as a prompt generator

Prompt optimization literature[9] treats meta-prompts just a prompt which generates prompts. As prompt generation is a complex reasoning-intensive language generation task[37], all the principles regarding prompt design for other hard tasks apply to meta-prompts as well.

Meta-prompts are usually templates for other data, such as examples of the task for Instruction Induction[38], past prompt generations along with their scores[1] and critiques of prompt outputs[39] or text description of the task[37]. Some reseach also utilizes professional task advice[9] or a prompt engineering tutorial[37]. Other methods used seed phrases, like a thinking style[40], to steer the generation.

In one of the few theoretically rigorous works on the subject, de Wynter et al.[36] formalize meta-prompts using category theory. Showing the possibility of creating a general purpose meta-prompt as well as suggesting that such meta-prompt will perform better than task-specific prompts in a wide range of applications.

2.3 Prompt optimization

In a previous section on inference-time algorithms, we outlined several methods for improving the performance of LLM generation and contrasted it to the traditional training-time scaling paradigm. In this section, we will discuss prompt optimization. This technique fits in neither training- or inference-time scaling techniques, as the computation allocated on optimizing our prompts can then be amortized over multiple uses of the prompt. We will refer to this as compile-time[8] scaling.

We will first discuss the motivation for automatizing the prompt engineering process. Prompt engineering is a largely empirical field without rigorous foundations. Although research agrees on some best practices, creating effective prompts often requires substantial trial-and-error experimentation and deep task-specific knowledge[41].

As LLM use permeates into the general public, research[28] notes difficulties of the average user with prompt design and evaluation as users bring expectations from human-to-human interactions to prompt design. These include the expectation that semantically equivalent instructions should produce equivalent results. This however does not hold and even minute alterations to the prompt, like adding a single space to the end, can dramatically influence the answer[31][33]. As another example in 2.2, zero-shot CoT[14] prefixes have vastly different performance on mathematical tasks.

Prompt	**GSM8K accuracy (%)**
“Let’s think step by step.”	71.8
“Let’s solve the problem together.”	60.5
“Let’s work together to solve this problem step by step.”	49.4

Table 2.2: Vastly different performance of CoT prefixes on GSM8K as per Yang et al.[1]

The average user is not used to the meticulous process of systematic program crafting and debugging. It seems that some coding proficiency is, in a way, a prerequisite for prompt engineering. With these limitations barring the general public from utilizing LLMs to their full potential, automatic prompt engineering and prompt optimization (PO) have become an exceptionally active research field in the recent years.

We can divide the PO research into two distinct branches depending on whether they treat prompts as sequences of discrete tokens or as soft embedding vectors. Both approaches have their pros and cons, summarized in table 2.3.

Feature	**Soft Optimization**	**Discrete Optimization**
Gradient use	✓	×
Interpretability	↓	↑
Transferability	↓	↑
Cost	↓	↑
Usable with APIs	×	✓

Table 2.3: Comparison of Soft vs. Discrete Prompt Optimization

Although soft PO is more effective as it allows for the use of continuous optimization methods such as gradient descent, with the increasing size of models it gets more expensive akin to fine-tuning. Furthermore, many of the most capable models are only accessible through proprietary APIs, which rarely allow access to the inner states of the LLM, rendering soft PO unusable. Optimizing prompts for an LLM hidden behind an API is inherently a black-box problem. In this thesis, we will study methods of discrete PO.

Table 2.4 offers an overview of discrete PO literature. Note that only only selected articles are included based on

1. being somewhat comparable,
2. being relevant to the implementation part of this thesis.

Also note that, due to space-constraints, only the most notable aspects of the methods are included. For a more comprehensive overview of PO methods, we refer the reader to a survey[9] by Ramnath et al.

Method	Initialization	Selection	Expansion	Notes
APE[42]	instruction induction	filter top-k prompts	paraphrasing	iterating does not help
ProTeGi[43]	manual initial prompt	UCB Bandits, Succesive Rejects	critique of a prompt as a "gradient"	state-of-the-art according to[44]
OPRO[1]	baseline prompt, instruction induction	filter top-k prompts	meta-prompt with scored prompts and exemplars	high sampling temperature for diversity
Promptbreeder[40]	seeded mutation of problem description	binary tournament	random operator out of 9 total	mutates meta-prompt, ICL examples
Evoprompt[45]	manual prompts and instruction induction	roulette and filter based on score	crossover and mutation	two operator variants
PE2[37]	manual or instruction induction	filter top-k prompts	meta-prompt with step-by-step instructions	optional task examples in meta-prompt
PhaseEvo[46]	manual or instruction induction	no filtering, success vector Hamming distance	instruction induction, EDA, crossover, feedback, paraphrase	alternates local, global search
PromptWizard[47]	variations from problem description	selects best prompt	critique + synthesis	improves examples after instructions
CriSPO[39]	manual initial prompt	filter top-k prompts	meta-prompt with critique history	self-generates critique aspects
SPRIG[48]	blank prompt	filters tens of thousands of candidates with UCB	edit-based enumeration of sentence-level operations	focus is on task-agnostic system prompts
SPO[41]	basic prompt template (e.g. CoT)	winner of pairwise evaluation	metaprompt utilizing feedback from evaluation	focus is on reference-free optimization

Table 2.4: Survey of Discrete PO Methods.

We can further divide the discrete PO research into two branches, depending on whether the method optimizes instructions (IO) or examples (EO). While some methods jointly optimize both, most research focuses on optimizing either one or optimizes them independently.

2.3.1 Exemplar optimization

Exemplar optimization (EO) focuses on selecting the most effective demonstrations for ICL, a powerful technique we discussed in 2.2.1, where the LLM learns the task implicitly from labeled input-output pairs. Although underrepresented in literature[44], recent work[49][44] shows that intelligently selected exemplars often outperform optimized instructions alone and even simple optimization methods like random search can lead to significant gains across diverse tasks. This effect is further amplified when EO is used together with IO, suggesting that the two should be co-optimized rather than treated separately[44]

EO is useful even for tasks without available examples with techniques like Bootstrapping[13], which utilizes inputs solved during optimization. By indentifying useful and informative solutions, it allows for EO without handcrafted input-output pairs.

2.3.2 Instruction optimization

The problem of finding the optimal instruction set can be formulated as a natural language program synthesis problem[42]. Although natural language program formulation is favorable as it represents a natural interface for humans to communicate with machines, it brings its own set of problems. Compared to automatic program synthesis methods, the search space of natural language is even larger. This makes finding the right instructions extremely difficult.

With continuous optimization, minor perturbations of e.g. network weights generates predictably small changes in functionality. Discrete changes on the other hand often dramatically change functionality[50] and are not amenable to gradient-based optimization[51].

Enumeration-based approaches

First discrete PO research such as APE[42] relied on Monte Carlo search, based on the idea of Instruction Induction[38]. This concept works by reverse-engineering instructions from a few task samples with a meta-prompt template like in 2.3.1.

2.3.1 Instruction Induction

LLM Input:

Below are a few examples of a task.
Write a set of instructions that would help me solve other examples of the task.

Q: Jim earns 10 dollars per hour as a waiter. How much does he earn for 5 hours and 45 minutes of work?
A: Jim earns $5*10 = 50$ dollars for the full 5 hours and $10*\frac{3}{4} = 7.5$ dollars for the 45 minutes. That makes $50 + 7.5 = 57.5$ dollars in total. #### 57.5
Q: ... **A:** ...

LLM Output:

Use step-by-step logical thinking to solve this mathematical word problem. Show your work and write your numerical answer separated by '####'.

By repeating this process with a non-zero sampling temperature while varying the example set, the LLM produces a diverse set of prompts. APE then scores them on a validation set and selects the instructions with the best performance. Extending this method, iterative APE explores local space around promising prompts by paraphrasing them. This however brought marginal performance gains in comparison to the basic Instruction Induction sampling[42]. Deng et al.[51] argue that this is because "paraphrasing-then-selection" methods do not explore the prompt space systematically.

Although the research has moved onto more advanced iterative optimization methods, Instruction Induction remains one of the most common ways of initialization in literature. The alternatives are manual initialization, which either consists of a basic prompt, or expertly prompt-engineering prompts. The former poses a problem, as starting the search with high-quality instructions is essential due to the intractably large search space[49]. The latter is beneficial as it allows the optimization to leverage human knowledge[45], but to an extent defeats the purpose of automatic PO. Also, it is impossible to have pre-defined expert prompts for tasks that are not known yet.

"Gradient"-based approaches

Although gradient-based optimization is unavailable in the discrete text space, many prompt optimization methods try to approximate with, as Tang et al.[52] put it, analogical gradient forms. Gradient forms provide a clear optimization direction in a hill-climber setting and allow the use gradient descent strategies such momentum, step size, warm-up and decay.

The simplest gradient form is a series of prompts and their scores in a meta-prompt instructing the optimizer LLM to create a new prompt in the sequence. The hope is for the model to extrapolate beyond the sequence of prompts and apply the observed pattern to acquire a better performing prompt.

2.3.2 Prompt+Score Meta-prompt

LLM Input:

Your task is to create a new prompt for a language model.
Below is a sequence of past prompts and their scores.
Design a new prompt in the sequence so that it achieves a better score.

Prompt 1: "Do the math." **Score**: 34.5
...
Prompt n**:** "Let's think step-by-step." **Score**: 70.3

Optional: Examples of the task like in 2.3.1

LLM Output:

Plan and solve the challenge while showing your work.

Ablations in OPRO[1] show that both the ascending order of the prompts as well showing scores in beneficial to the optimization process. Furthermore, another crucial part of the meta-prompt according to [1] and [52] are examples of the task similar to the Instruction Induction meta-prompt. This helps the LLM better understand the task at hand. An alternative would be to include a description of the task[39].

Another branch of research, e.g. ProTeGi[43] and CriSPO[39] focus on using model feedback as a part of the optimization signal. Outputs of LLMs contain rich quality information that directly reflects prompt effectiveness[41]. Utilizing this information makes for a stronger optimization signal than just a numerical score. This is pronounced particularly for text-generation tasks, where applying PO methods based on prompt+score pairs is challenging due to the lack of effective optimization signals[39]. A single number does not capture the nuances of text and opportunities for improvement.

Pryzant et al.[43] compare the prompt critique to a gradient in the text space. The LLM can in principle, thanks to its human-like task comprehension[41], perform a sort of gradient descent by fixing the issues found by the critique. The LLM outputs on a task can be viewed as an environment observation - an extrinsic information source. This is, as we discussed in 2.1, crucial for the LLM's ability to self-refine. Feedback-enriched meta-prompts, such as 2.3.3, may include prompt scores as well.

2.3.3 Prompt+Feedback+Score Meta-prompt

LLM Input:

Your task is to create a new prompt for a language model.
Below is a sequence of past prompts and their critiques and scores.
Design a new prompt in the sequence so that it achieves a better score.

Prompt 1**:** "Do the math." **Score**: 34.5
Critique: Does not encourage step-by-step reasoning.
...
Prompt n**:** "Let's think step-by-step." **Score**: 70.3
Critique: Too general, could be more enthusiastic.

LLM Output:

Yay! Let's solve this math problem with logical thinking!

The instructions in 2.3.2 and 2.3.3 are rather basic and can be endlessly improved via prompt engineering. For example Ye et al.[37] attempt to improve on APE and ProTeGi by designing a more complete meta-prompt.

By extending the gradient analogy, Tang et al.[52] introduce other concepts known from traditional machine learning. First, the learning rate can be implemented by limiting the number of token, word or sentence edits the model can make. Next, we can encorporate variable learning rate with strategies such as warm-up (learning rate grows in the beggining) or decay (learning rate gets smaller towards the end).

Another way of balancing exploration and exploitation, besides edit limit-based learning rate analogies, is tuning the LLM sampling temperature . Lower temperature encourages exploitation in the local solution space and higher temperature allows more aggressive exploration of different solutions[1].

By encorporate a history of past prompts in the metaprompt, we implement an analogy of momentum, a concept from machine learning which utilizes past gradients. Including the optimization trajectory is useful but poses the problem of inflating the meta-prompt. This means higher costs per prompt generation, but also risks of surpassing the LLM's context length limit. To fit into the context limit, trajectory can be summarized or retrieved based on recency, relevance or importance[52].

Evolution-based approaches

Evolutionary Algorithms (EAs) are a time-proven and versatile optimization method. They have been shown to be effective in search spaces with millions and billions of variables[30] by emulating the processes observed in natural evolution using crossover (sexual reproduction) and mutation (asexual reproduction) operators. Although widely successful, EAs have been limited by the challenging nature of operator design. Developing them requires extensive manual crafting with domain knowledge[53].

With the advent on LLMs and few-shot prompting, their ability to complete patterns can be leveraged to create a form of intelligent evolutionary crossover[30], which can be in theory truly general.

In principle, LLMs can mutate and combine any text representation that has moderate support in its training dataset and perform any genetic operator through fine-tuning or prompt engineering. The intersection between LLMs and EAs, among other traditional algorithms, emerges as an exciting branch of research.

It lends itself to leverage LLM-powered EAs to improve prompts for the LLM, extending previously discussed PO methods. The seminal work in this space was EvoPrompt[45], which repurposed two EAs: Genetic Algorithm and Differential Evolution for PO. Treating the text sequences in prompts as gene sequences, EvoPrompt performes crossover and mutation on text prompts.

EA-based PO seems to demostrate lower sensitivity to initial prompts. Although EvoPrompt utilizes some manual prompts, it achieves similar results with randomly sampled initial population as when using the best prompts[45]. On the other hand, they suffer from extremely high computational cost and slow convergence speed[46].

Methods like PhaseEvo[46] and Promptbreeder[40] strive to improve the efficiency and convergence of EA-based PO methods by supercharging them with more operators to provide balance between exploration and exploitation. Cui et al.[46] argue that the standard EA operators prioritize exploration and categorize them as global operators. To supplement them, they introduce local operators based on feedback, optimization trajectory and paraphrasing. By alternating between global and local search, PhaseEvo demostrates better cost-efficiency compared to other EA-based PO methods[46].

In Promptbreeder[40], authors also include multiple different operators. Besides variations of previously discussed operators, they also include shuffling ICL examples and, most notably, a "hyper-prompt" which mutates the optimizer meta-prompt in hope to make the method self-referential. A possible flaw in this method is the fact that it selects the operators randomly at each step, leading to diminished efficiency compared to a more organized approach[46] and hundreds of evaluations necessary before convergence[44].

All three discussed EA-based methods utilize Instruction Induction[38] to some extent, usually using calling it "Lamarckian mutation". Besides initialization, Lamarckian mutation can be useful for adding task-relevant prompts to the population in case the optimization diverges[40].

2.3.4 Mutation Meta-prompt

LLM Input:

> Rewrite the prompt below in a semantically equivalent but novel way.
>
> **Prompt:** "Let's think step-by-step."

LLM Output:

> We will solve this in logical increments.

2.3.5 Crossover/EDA Meta-prompt

LLM Input:

> Combine the following prompts into a novel prompt.
>
> **Prompt 1:** "Do the math."
> ...
> **Prompt n:** "Let's think step-by-step."

LLM Output:

> Do the step-by-step thinking and solve the math problem.

In 2.3.4 and 2.3.5 the reader can find possible meta-prompts for a mutation and a crossover operater, respectively. By changing the meta-instructions in 2.3.5 and increasing n, we can shift the crossover operator into a Estimation-of-distribution (EDA) operator. Whereas crossover aims to combine 2-3 prompts into 1, EDA infers the distribution of a larger number of prompts and tries to create a new prompt from that distribution.

An important factor in the implementation of the crossover and EDA operators is the way of selecting prompt specimens. In EvoPrompt[45], a roulette selection method is employed. Other methods utilize more advanced selection methods to encourage diversity. Promptbreeder[40] filters inputs for its EDA based on their BERT embedding and PhaseEvo[46] selects parent prompts based on the Hamming distance of their "performance vectors", which hold the prompts' performance on task samples. This way ensures that the prompts that get paired with each other do not make the same mistakes.

Another important factor in the design of the operator is the ordering. Both Promptbreeder and PhaseEvo sort the prompts in ascending order of fitness. To prevent the model from relying too much on the last prompts, the authors lie to the model by telling it the prompt are in descending order of fitness. This somewhat curbs the bias toward the later examples[46].

EAs are but one of many metaheuristics historically used for optimization and the interactions of LLMs and metaheuristics is poised to be a fruitful research area in the near future. In their research Pan et al.[54] used and compared several metaheuristics including Hill-Climbing, Simulated Annealing, Genetic Algorithm, Tabu Search and Harmony Search for PO.

Note on evaluation cost reduction

In many PO methods, prompt candidate evalutation is the most computation-intensive part of the process. Especially with score-based evaluation, it has to be performed multi times to ensure scoring stability[41]. To lower the evaluation costs, research adopts two approaches.

First branch, represented by ProTeGI[43] and SPRIG[48] use strategies such as UCB and Succesive Rejects to allocate evaluations only to promising candidates. This way, the total compute budget gets reduced. The other branch relies on ditching the numerical scoring and comparing outputs directly in a pairwise manner with a LLM judge[41].

2.3.3 Multi-stage and reference-free methods

Common critique of the methods is that they are unpractical and not applicable to real-world LLM use cases. Most methods depend heavily on external references for evaluation which are often unavailable or unpractical to define especially for open-ended tasks [41]. Xiang et al.[41] tackle this problem by using pairwise LLM-based evaluation and Zhang et al.[22] develop a gold label-free method method on evaluation based on self-consistencies of different answers.

Also, as the complexity of prompt structure increases, many prompt optimization techniques are no longer applicable[8]. Schnabel et al.[8] define Symbolic Prompt Programs, representable as directed acyclic graphs. In these graphs, nodes are functions, such as `RenderText`, `GenerateResponse` or `ParseOutput` and edged are dependencies between these nodes. This representation allows for effective search with node mutators and a multitude of search algorithms.

Furthermore, modern LLM workflows interconnect various LLM and prompt instances into complex prompt programs. Most prompt optimizer approaches do not apply to these multi-stage LLM programs[49] as a whole. Optimizing each component separately is possible but this approach ignores their mutual influence.

DSPy[13] is a Python library aiming to simplify LLM program composition with an intuitive interface. Most importantly, it allows for optimizing the whole pipeline, including EO, IO and weight fine-tuning with the promise of declarative LLM program design without extensive prompt engineering. DSPy offers several pipeline optimization methods, like MiPROv2[49] and BetterTogether[55]. MiPROv2 generalizes OPRO[1] to multi-stage joint example and instruction optimization utilizing a surrogate Bayesian model to find the optimal configuration. With BetterTogether, Soylu et al. show that PO and fine-tuning complement each other, achieving superior performance.

Chapter 3

Implementation

3.1 Inference framework

Taking inspiration from DSPy[13], we first implement a simple LLM-calling framework capable of invoking several selected inference strategies. Motivations for this are twofold:

1. DSPy is a young and ambitious project aiming at simplifying LLM pipeline design and optimization. As we focus on single-stage prompt program optimization, this capability is not useful for our work. Furthermore, due to the framework's infancy, it lacks proper documentation and sometimes exhibits unexpected behavior.
2. Implementing the prompting techniques discussed in 2.1 provides further insight into their workings and performance.

3.1.1 Structured generation

Following current research trends[34], we build our inference framework around a structured JSON template, or a `Signature`. The `Signature` structure consists of input and output fields and additional instructions. These fields are populated by a `Field` data structure. Of particular interest are the output fields, which hold the output name, desired type and optional description.

When employing good naming practices the model can often deduce the task only by looking at output names and types. Consider the following `Signature`:

3.1.1 Simple Signature

Word: `str` → Antonymum: `str`

For more complex tasks, filling the output descriptions or even adding explicit instructions is necessary. In 3.1.2 notice that it is possible to specify multiple inputs and outputs, which are then generated in the order given.

3.1.2 Complex Signature

Text: `str`, Grading guide: `str` → Evaluation: `str`, Grade: `int`

Instructions:

> Grade the text.
> You are an expert text evaluator.
> Use the grading guide to evaluate the test and give a final grade.
> Use formal language and markdown formatting in the evaluation and output a 1-10 integer for the grade.

Sufficiently large instruction-tuned LLMs are usually good at reliably producing JSON output. For smaller models or more complex output structures, it might be necessary to use some form of constrained generation as discussed in 2.1. A JSON schema could be constructed automatically from the `Signature` and passed into a parser-based sampler. However this is not necessary for our use-case.

3.1.2 Predict method

To facilitate `Signature`-powered generation, we implement a `predict` method that involves

1. Prepending a developer prompt to the messages
2. Parsing of `Signature` outputs
3. Repeated generation in case of parsing failure.

3.1.3 Predict method developer prompt

You are an intelligent function that returns structured JSON outputs matching a given schema.

You will receive a JSON object containing:
- ‘inputs‘: a dictionary of named inputs
- ‘outputs‘: a dictionary specifying the expected output fields with their types and descriptions
- ‘instructions‘: a task or question to answer (optional)

Your job is to:
1. Understand the task from ‘instructions‘ or infer it from ‘inputs‘ and ‘outputs‘
2. Use the ‘inputs‘ to compute or generate the answer
3. Respond **only** with keys from the ‘outputs‘ dictionary and values matching the described types

Only return a flat JSON object like:
{
"field1": <value matching type and description>,
"field2": <...>
}

Do not add metadata, explanations, or wrap outputs in additional structures.
Do not include type names or field descriptions in the output.
Your output must be strictly valid JSON and fill **all** requested output fields.

The developer prompt has to clearly explain to the LLM how to work with the JSON-based `Signature`. In 3.1.3 notice the sections of the prompt following principles outline in 2.2. First, the directive states the task, then a further context is added about the `Signature` data structure and the task. Next, notice the example showing the proper output, and finally few more clarifying instructions about the output format. In experiments, this prompt is successful in incentivizing parseable outputs adhering to the specifications.

Parsing the output presents some challenges as the LLM sometimes wraps the JSON output into a markdown code block or uses inconsistent escape sequences. We implement a simple parses based on regular expressions that is able to parse a majority of outputs. In case of model failure, such as getting stuck in a generation loop, we add a repeated generation feature.

3.1.3 Inference techniques implementation

Leveraging the `predict` method and the modular `Signature`-based interface, we implement a suite of inference-time prompting techniques. Each technique is realized through systematic modifications of the `Signature` fields, changing the developer prompt and the chaining of multiple generation steps and function calls. This design allows for composability and reuse while preserving transparency. We implement the following methods.

1. **Chain-of-thought**[14]: Prepends a reasoning field to the `Signature` outputs which forms a scratchpad for the LLM.
2. **Chain-of-thought with Self-consistency**[21]: Multiple CoT generations with majority-voting.
3. **ReAct**[12]: Adding tools allows the LLM to interleave thoughts and action steps.
4. **Program-of-thought**[11]: Two-stage CoT with Python-code execution
5. **Reflexion**[27]: After an initial generation, the model is prompted to self-critique and revise its output.
6. **Tree of Thoughts**[16]: The problem is first decomposed and each step is expanded, forming a thought tree, which is then traversed with BFS or DFS.

3.2 Optimization Framework

Although our first implementation attempt utilized an evolutionary algorithm, we will use a basic population-based hill-climber algorithm. This design decision has several reasons.

1. Most PO research uses a hill-climber architecture.
2. EAs suffer from slow convergence compared to state-of-the-art hill-climber PO.
3. PO is complex as it is and more complicated architectures only introduce more hyperparameters.

Therefore, we follow the algorithm outlined in 1 and focus on designing the operators, which define the optimization process.

3.2.1 Operator Design

A common issue observed is the loss of generality. The model might misunderstand the metaprompt and, instead of generating a general instruction, attempt to make a tutorial for solving just the example problems, or even try to directly solve them.

Failure case: To solve this type of problem, first calculate how much money Tobias had before purchasing the shoes. Use the given amounts for his allowance and mowing lawns to determine part of his savings. Subtract these earnings from his total savings to find how much he earned from shoveling driveways. Finally, divide the remaining amount by the price per driveway to find the number of driveways shoveled. Success case: 1. Read the problem carefully and identify the key data points. 2. Determine what the problem is asking you to find. 3. Break down the problem into smaller parts if necessary. 4. Use appropriate mathematical operations to solve each part. 5. Combine the results to find the final answer. 6. Double-check your calculations and ensure the solution satisfies all conditions stated in the problem.

3.3 Datasets

Chapter 4

Experiments

Appendix A

Bibliography

[1] C. Yang, X. Wang, Y. Lu, H. Liu, Q. V. Le, D. Zhou, and X. Chen, "Large language models as optimizers," 2024.

[2] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro, and Y. Zhang, "Sparks of artificial general intelligence: Early experiments with gpt-4," 2023.

[3] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," 2020.

[4] S. Welleck, A. Bertsch, M. Finlayson, H. Schoelkopf, A. Xie, G. Neubig, I. Kulikov, and Z. Harchaoui, "From decoding to meta-generation: Inference-time algorithms for large language models," 2024.

[5] OpenAI, "Openai o1 system card," 2024.

[6] DeepSeek-AI, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025.

[7] S. Schulhoff, M. Ilie, N. Balepur, K. Kahadze, A. Liu, C. Si, Y. Li, A. Gupta, H. Han, S. Schulhoff, P. S. Dulepet, S. Vidyadhara, D. Ki, S. Agrawal, C. Pham, G. Kroiz, F. Li, H. Tao, A. Srivastava, H. D. Costa, S. Gupta, M. L. Rogers, I. Goncearenco, G. Sarli, I. Galynker, D. Peskoff, M. Carpuat, J. White, S. Anadkat, A. Hoyle, and P. Resnik, "The prompt report: A systematic survey of prompting techniques," 2024.

[8] T. Schnabel and J. Neville, "Symbolic prompt program search: A structure-aware approach to efficient compile-time prompt optimization," 2024.

[9] K. Ramnath, K. Zhou, S. Guan, S. S. Mishra, X. Qi, Z. Shen, S. Wang, S. Woo, S. Jeoung, Y. Wang, H. Wang, H. Ding, Y. Lu, Z. Xu, Y. Zhou, B. Srinivasan, Q. Yan, Y. Chen, H. Ding, P. Xu, and L. L. Cheong, "A systematic survey of automatic prompt optimization techniques," 2025.

[10] X. Wang and D. Zhou, "Chain-of-thought reasoning without prompting," 2024.

[11] W. Chen, X. Ma, X. Wang, and W. W. Cohen, "Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks," 2023.

[12] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," 2023.

[13] O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Vardhamanan, S. Haq, A. Sharma, T. T. Joshi, H. Moazam, H. Miller, M. Zaharia, and C. Potts, "Dspy: Compiling declarative language model calls into self-improving pipelines," 2023.

[14] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in *Advances in Neural Information Processing Systems* (S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, eds.), vol. 35, pp. 22199–22213, Curran Associates, Inc., 2022.

[15] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2023.

[16] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," 2023.

[17] B. Prystawski, M. Y. Li, and N. D. Goodman, "Why think step by step? reasoning emerges from the locality of experience," 2023.

[18] B. Brown, J. Juravsky, R. Ehrlich, R. Clark, Q. V. Le, C. Ré, and A. Mirhoseini, "Large language monkeys: Scaling inference compute with repeated sampling," 2024.

[19] Z. Zeng, Q. Cheng, Z. Yin, Y. Zhou, and X. Qiu, "Revisiting the test-time scaling of o1-like models: Do they truly possess test-time scaling capabilities?," 2025.

[20] R. Liu, J. Geng, A. J. Wu, I. Sucholutsky, T. Lombrozo, and T. L. Griffiths, "Mind your step (by step): Chain-of-thought can reduce performance on tasks where thinking makes humans worse," 2024.

[21] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," 2023.

[22] X. Zhang, Z. Zhang, and H. Zhao, "Glape: Gold label-agnostic prompt evaluation and optimization for large language model," 2024.

[23] M. Besta, N. Blach, A. Kubicek, R. Gerstenberger, M. Podstawski, L. Gianinazzi, J. Gajda, T. Lehmann, H. Niewiadomski, P. Nyczyk, and T. Hoefler, "Graph of thoughts: Solving elaborate problems with large language models," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, p. 17682–17690, Mar. 2024.

[24] K. Misaki, Y. Inoue, Y. Imajuku, S. Kuroki, T. Nakamura, and T. Akiba, "Wider or deeper? scaling llm inference-time compute with adaptive branching tree search," 2025.

[25] F. Teng, Z. Yu, Q. Shi, J. Zhang, C. Wu, and Y. Luo, "Atom of thoughts for markov llm test-time scaling," 2025.

[26] Z. Wang, J. Zeng, O. Delalleau, D. Egert, E. Evans, H.-C. Shin, F. Soares, Y. Dong, and O. Kuchaiev, "Dedicated feedback and edit models empower inference-time scaling for open-ended general-domain tasks," 2025.

[27] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," 2023.

[28] J. Zamfirescu-Pereira, R. Y. Wong, B. Hartmann, and Q. Yang, "Why johnny can't prompt: How non-ai experts try (and fail) to design llm prompts," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, (New York, NY, USA), Association for Computing Machinery, 2023.

[29] L. Reynolds and K. McDonell, "Prompt programming for large language models: Beyond the few-shot paradigm," 2021.

[30] E. Meyerson, M. J. Nelson, H. Bradley, A. Gaier, A. Moradi, A. K. Hoover, and J. Lehman, "Language model crossover: Variation through few-shot prompting," 2024.

[31] J. Zhuo, S. Zhang, X. Fang, H. Duan, D. Lin, and K. Chen, "Prosa: Assessing and understanding the prompt sensitivity of llms," 2024.

[32] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[33] A. Salinas and F. Morstatter, “The butterfly effect of altering prompts: How small changes and jailbreaks affect large language model performance,” 2024.

[34] Y. Zhang, Y. Yuan, and A. C.-C. Yao, “Meta prompting for ai systems,” 2025.

[35] Y. Hou, H. Dong, X. Wang, B. Li, and W. Che, “Metaprompting: Learning to learn better prompts,” 2023.

[36] A. de Wynter, X. Wang, Q. Gu, and S.-Q. Chen, “On meta-prompting,” 2024.

[37] Q. Ye, M. Axmed, R. Pryzant, and F. Khani, “Prompt engineering a prompt engineer,” 2024.

[38] O. Honovich, U. Shaham, S. R. Bowman, and O. Levy, “Instruction induction: From few examples to natural language task descriptions,” 2022.

[39] H. He, Q. Liu, L. Xu, C. Shivade, Y. Zhang, S. Srinivasan, and K. Kirchhoff, “Crispo: Multi-aspect critique-suggestion-guided automatic prompt optimization for text generation,” 2024.

[40] C. Fernando, D. Banarse, H. Michalewski, S. Osindero, and T. Rocktäschel, “Promptbreeder: Self-referential self-improvement via prompt evolution,” 2023.

[41] J. Xiang, J. Zhang, Z. Yu, F. Teng, J. Tu, X. Liang, S. Hong, C. Wu, and Y. Luo, “Self-supervised prompt optimization,” 2025.

[42] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, and J. Ba, “Large language models are human-level prompt engineers,” 2023.

[43] R. Pryzant, D. Iter, J. Li, Y. T. Lee, C. Zhu, and M. Zeng, “Automatic prompt optimization with "gradient descent" and beam search,” 2023.

[44] X. Wan, R. Sun, H. Nakhost, and S. O. Arik, “Teach better or show smarter? on instructions and exemplars in automatic prompt optimization,” 2024.

[45] Q. Guo, R. Wang, J. Guo, B. Li, K. Song, X. Tan, G. Liu, J. Bian, and Y. Yang, “Connecting large language models with evolutionary algorithms yields powerful prompt optimizers,” 2024.

[46] W. Cui, J. Zhang, Z. Li, H. Sun, D. Lopez, K. Das, B. Malin, and S. Kumar, “Phaseevo: Towards unified in-context prompt optimization for large language models,” 2024.

[47] E. Agarwal, J. Singh, V. Dani, R. Magazine, T. Ganu, and A. Nambi, “Promptwizard: Task-aware prompt optimization framework,” 2024.

[48] L. Zhang, T. Ergen, L. Logeswaran, M. Lee, and D. Jurgens, "Sprig: Improving large language model performance by system prompt optimization," 2024.

[49] K. Opsahl-Ong, M. J. Ryan, J. Purtell, D. Broman, C. Potts, M. Zaharia, and O. Khattab, "Optimizing instructions and demonstrations for multistage language model programs," 2024.

[50] J. Lehman, J. Gordon, S. Jain, K. Ndousse, C. Yeh, and K. O. Stanley, "Evolution through large models," 2022.

[51] M. Deng, J. Wang, C.-P. Hsieh, Y. Wang, H. Guo, T. Shu, M. Song, E. P. Xing, and Z. Hu, "Rlprompt: Optimizing discrete text prompts with reinforcement learning," 2022.

[52] X. Tang, X. Wang, W. X. Zhao, S. Lu, Y. Li, and J.-R. Wen, "Unleashing the potential of large language models as prompt optimizers: An analogical analysis with gradient-based model optimizers," 2024.

[53] S. Liu, C. Chen, X. Qu, K. Tang, and Y.-S. Ong, "Large language models as evolutionary optimizers," 2024.

[54] R. Pan, S. Xing, S. Diao, W. Sun, X. Liu, K. Shum, R. Pi, J. Zhang, and T. Zhang, "Plum: Prompt learning using metaheuristic," 2024.

[55] D. Soylu, C. Potts, and O. Khattab, "Fine-tuning and prompt optimization: Two great steps that work better together," 2024.