

# idea

## Task: 用in-context learning来优化prompt

首先这里的prompt肯定是含有soft prompt的, soft prompt是learnable的而hard prompt是不可学习的

利用in-context learning来优化, 就是利用demonstration, 目前了解到demonstration的数目对效果有影响

*Influence Function对data的选取?*

一个可能的setting是要用LLM自己迭代优化prompt???

---

## 关于LLM as Optimizers的思考

### 现有的方法

1. Resampling-based methods (选择语义类似的prompt)
  - Iterative-APE/APE
2. Reflection-based methods (可能发生语义改变的prompt)
  - APO
  - PromptAgent
  - OPRO(implicit reflection-based method): 在有些task上表现相对较差, 一个是因为优化方法没有方向导致LLM不知道什么是"better prompts", 另一个可能是因为LLM并不一定在最优的prompt附近sample

实现上的对比:

APE / Iterative-APE:

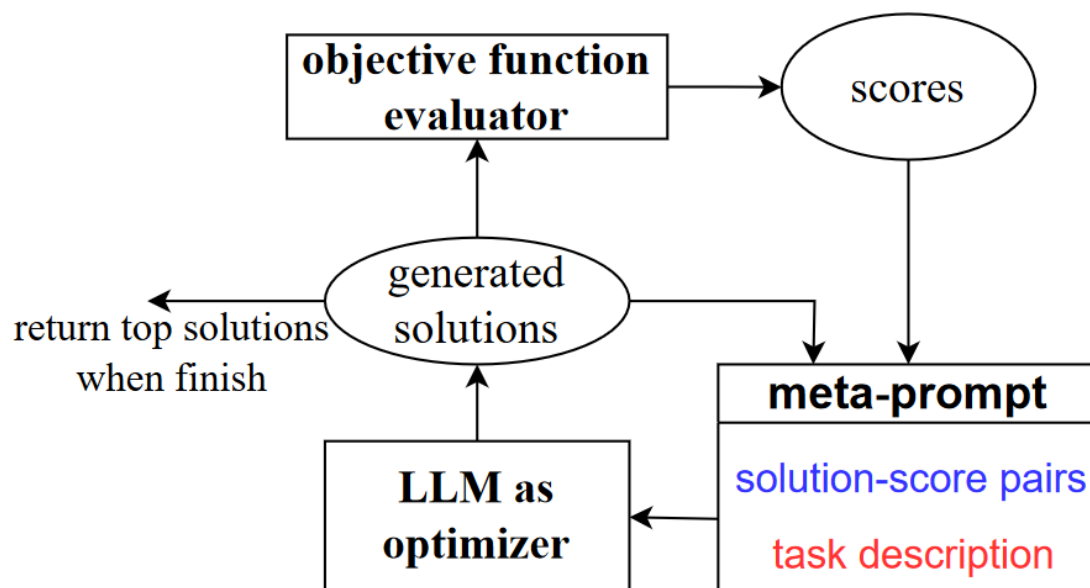
1. 首先根据给定的一些demonstrations(**这些demonstrations是随机的?**), LLM生成prompts"池"
2. 然后根据特定的score function给prompts打分, 并根据分数进行filter/refine

3. 更新prompts"池"(APE: 直接根据分数更新 Iterative-APE: 让 LLM生成语义类似的prompts进行更新)

APO: 第一个prompt  $\nabla$  根据初始的prompt  $p_0$  产生一系列的梯度  $g$  ; 第二个prompt  $\delta$  利用梯度  $g$  来edit当前的prompt  $p_0$

1. 初始的prompt为  $p_0$
2. 通过一个Expand算法, 根据当前的prompt, 先生成梯度  $g$  , 然后根据梯度  $g$  来修改当前的prompt, 最后运用一个**paraphrasing LLM**(called  $LLM_{mc}$ )在新的prompt candidates周围explore local monte carlo search space(要求LLM生成跟输入语义相似但是用词不同的新candidates)
3. 通过算法**UCB Bandits**或者算法**Successive Rejects**来挑选 prompts

OPRO:



该方法是按照optimization trajectory生成新的prompts, 同时对 prompt插入的位置也做了探索

## 现有的问题

1. LLM在self-correction上表现得并不是太好, 体现在LLM产生的 reflection重复/类似(无论error distribution是什么样), 因此可能导致LLM产生的prompt并不适用

2. LLM optimizer产生的新prompt并不一定会被target model follow, 因为target model的instruction-following的能力是不可控制的
3. LLM对prompt format比较sensitive, 语义相似的prompts可能表现得截然不同

## 可能的方案

1. Automatic Behavior Optimization(ABO):
  1. At each step, the LLM optimizer is instructed to generate step-by-step prompts.
  2. Next, we utilize the LLM optimizer to write an “Instruction-following Demonstration” for each prompt, i.e., an example illustrating how to strictly follow every detail of the given prompt. This practice aims to enhance the controllability of the prompt optimization process by ensuring any improvement made will be strictly followed by the target models.
  3. During the reflection and prompt refinement process, given error examples, the LLM optimizer is required to identify the failure step of the target model and refine the prompt by further breaking down the solution at the problematic step. This aims to avoid invalid feedback by utilizing the LLM optimizer to perform more objective tasks during reflection.
  4. For each refined prompt, the instruction-following demonstration is also updated to illustrate how to strictly follow the refined steps.

## Prompt Tuning vs Prompt Engineering

来自一个[Youtube视频](#)

Prompt Engineering是Hard prompt, 通常是人工构建的

Prompt Tuning是Soft prompt, 是AI生成的embedding, 可解释性差

## Hard prompt vs Soft prompt

Hard prompt是Discrete Prompt, prompt是一个实际的文本字符串

Hard prompts are manually handcrafted text prompts with discrete input tokens. ~ HuggingFace

Soft Prompt是Continuous prompt, 直接在底层语言模型的embedding中进行描述

Soft prompts are learnable tensors concatenated with the input embeddings that can be optimized to a dataset; the downside is that they aren't human readable because you aren't matching these "virtual tokens" to the embeddings of a real word. ~ HuggingFace

"soft" prompts designed by an AI that outperformed human-engineered "hard" prompts. ~ [Source](#)

## Prefix Tuning vs Prompt Tuning vs P-tuning

1. Prefix Tuning与Prompt Tuning的区别: The prefix parameters are inserted in **all** of the model layers, whereas prompt tuning only adds the prompt parameters to the model input embeddings. The prefix parameters are also optimized by a separate feed-forward network (FFN) instead of training directly on the soft prompts because it causes instability and hurts performance
2. P-tuning(与Prefix Tuning的区别): The prompt tokens can be inserted anywhere in the input sequence, and it isn't restricted to only the beginning. The prompt tokens are only added to the input instead of adding them to every layer of the model. Introducing *anchor* tokens can improve performance because

they indicate characteristics of a component in the input sequence

## Completion/Token Concept

The inputs are called *prompts* and outputs are referred to as *completions*.

LLMs take the input *prompts* and chunk them into smaller units called *tokens* to process and generate language. Tokens may include trailing spaces and even sub-words. This process is language dependent.