References

- LLMs are Human-Level Prompt Engineers (https://arxiv.org/pdf/2211.01910.pdf)
    - APE summary (https://sites.google.com/view/automatic-prompt-engineer)

Using an LLM to improve prompts to an LLM

The Automatic Prompt Engineer (APE) is a system to *improve* upon prompts

- given a prompt
- APE will create a prompt that is *more effective*

It uses an LLM for multiple purposes

- to create variations of the given prompt
- to evaluate which variation is best

Using APE to improve upon Instruction Following

Many prompts are of the form

- a textual *task description* (*instruction*) that describes a task to be performed
- zero or more exemplars: demonstrating the input/output relationship described by the instruction

Given just the exemplars

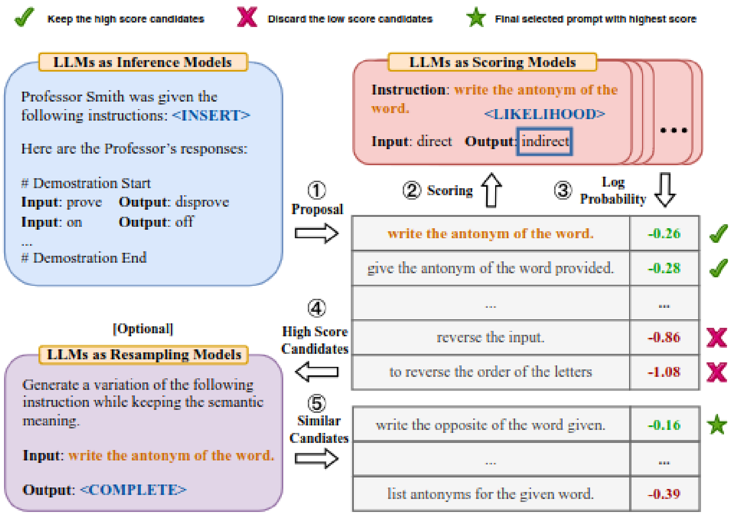
- APE can *create* the "best" instruction

The APE method for automatically generating "good" instructions is conceptually simple.

- Use an LLM to create a distribution of instruction, conditional on the exemplars
- Evaluate the consistency of the instruction with the exemplars
- Optionally
    - Choose the instructions with highest evaluation
    - Ask a LLM to generate variations of each instruction

Here is a picture of the workflow.

APE Workflow



(a) Automatic Prompt Engineer (APE) workflow

Attribution: https://arxiv.org/pdf/2211.01910.pdf#page=2

Step 1: LLM as Inference Model

Get the LLM to create an multiple *instruction* candidates, given the intended response

- *response* to a (yet to be created) instruction
    - `Input: prove, Output: disprove`

Given the context (response), the LLM is prompted to generated the *instruction* that could cause the given response

- prompt is a Masked Language Modeling task: fill in the mask (`<INSERT>`)

This is in the same spirit as Backtranslation (LLM_Instruction_Following_Synthetic_Data.ipynb#Instruction-Backtranslation)

- learns a new model to map from response to instruction
- rather than adapting an LLM to do the same

Steps 2 and 3: LLM as Scoring Model

Evaluate each candidate instruction created by the LLM in Step 1

The evaluation is performed by prompting the LLM to compute the likelihood

- that a given response

  ```
  Input: direct, Output: indirect
  ```

- is consistent with each candidate instruction

The candidate instruction with the highest likelihood is selected as the Instruction to use.

Note

Likelihood is expressed as the log probability

- is a negative number since probabilities are fractions
- less negative numbers are higher probabilities

Steps 4 and 4 (Optional): LLM as Re-sampling model

Given the candidate Instruction selected by the previous step

- generate a variation of the instruction
- by asking the LLM to create it via text completion

```
Generate a variation of the following instruction ...

Input: write the antonym of the word; Output: <COMPLETE>
```

Each step is implemented as an instance of the pre-trained LLM's ability to complete text (or fill in a mask).

No fine-tuning or adaptation of weights is involed.

Forward/Reverse mode generation of candidates

This is a minor technical point.

The prompt in Step 1 of the workflow above is not in the format consistent with text-continuation

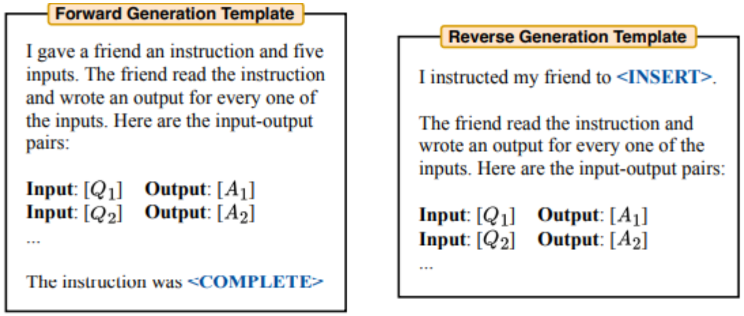
- format is called *forward generation*
- so must use an LLM that solves Masked Language task, rather than text-continuation

An alternate prompt can be used that is consistent with text-continuation

- format is called *reverse generation*

APE Forward/Reverse Generation templates

Attribution: https://arxiv.org/pdf/2211.01910.pdf#page=4



APE evaluation: super-human performance !

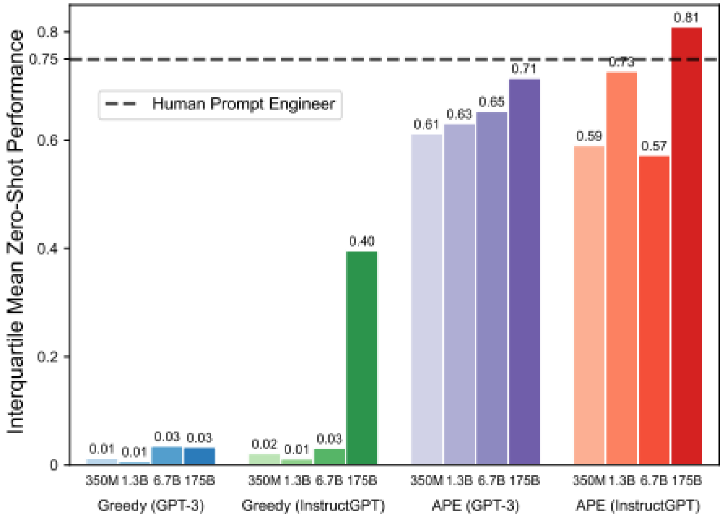
Here is a comparison of APE generated prompts

- versus
    - an alternate method (previously published), labeled "Greedy"
    - a human engineer (horizontal dotted line)
- evaluated on models of various sizes
    - GPT-3
    - Instruct GPT-3 (fine-tuned for instruction following)
- using 24 NLP tasks

Note

The reported statistic is *interquartile mean* (i.e., average after dropping the upper and lower 25% of results)

APE Workflow and Results



(b) Interquartile mean across 24 tasks

Attribution: https://arxiv.org/pdf/2211.01910.pdf#page=2

Zero-shot: Improving on ”Let’s think step by step”

[Chain of Thought (CoT) prompting (NLP_Beyond_LLM.ipynb#Chain-of-thought-prompting)](NLP_Beyond_LLM.ipynb#Chain-of-thought-prompting)

- is a simple technique
- for create prompts with better performance
- for multi-step reasoning problems

In the few-shot setting

- exemplars demonstrate step by step reasoning
- eliciting the LLM to produce text continuation that also exhibits step by step reasoning

In the zero-shot setting, it simply involves appending

> *Let's think step by step*

to the prompt

Chain of Thought Prompting

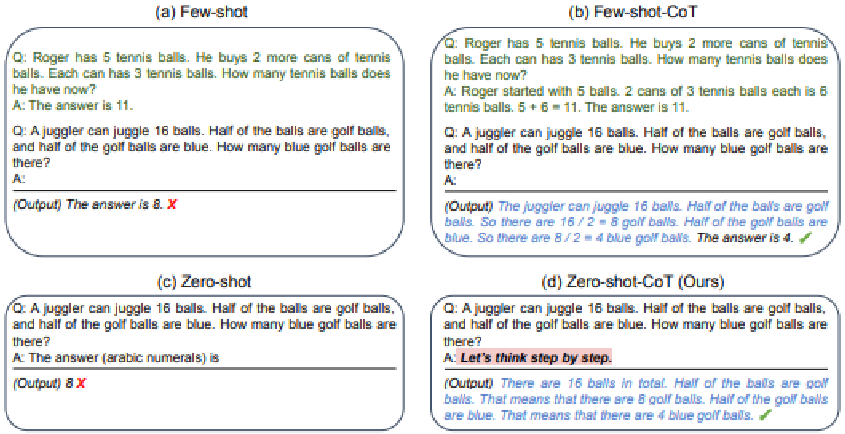


Figure 1: Example inputs and outputs of GPT-3 with (a) standard Few-shot ([Brown et al., 2020]), (b) Few-shot-CoT ([Wei et al., 2022]), (c) standard Zero-shot, and (d) ours (Zero-shot-CoT). Similar to Few-shot-CoT, Zero-shot-CoT facilitates multi-step reasoning (blue text) and reach correct answer where standard prompting fails. Unlike Few-shot-CoT using step-by-step reasoning examples **per task**, ours does not need any examples and just uses the same prompt "Let's think step by step" *across all tasks* (arithmetic, symbolic, commonsense, and other logical reasoning tasks).

Attribution: https://arxiv.org/pdf/2201.11903.pdf

Let's use APE (https://arxiv.org/pdf/2211.01910.pdf#page=19) to find a zero-shot prompt appendage that improves upon

> *Let's think step by step*

The authors use the following template (where `INPUT` and `OUTPUT` are place-holders for an actual question and answer pair).

```
Instruction: Answer the following question

Q: INPUT
A: Let's <INSERT> OUTPUT
```

We are using forward-mode generation to get APE

- to create a phrase that follows the `INPUT`
- that begins with the word "Let's"

APE creates

> *Let’s work this out in a step by step way to be sure we have the right answer.*

and the author's demonstrate improved performance on several benchmarks.

This is a nice demonstration of using an LLM to help craft better prompts to LLM's.

In [2]:

```
print("Done")
```

```
Done
```