SELF-REFINE: Iterative Refinement with Self-Feedback

Aman Madaan[1], **Niket Tandon**[2], **Prakhar Gupta**[1], **Skyler Hallinan**[3], **Luyu Gao**[1], **Sarah Wiegreffe**[2], **Uri Alon**[1], **Nouha Dziri**[2], **Shrimai Prabhumoye**[4], **Yiming Yang**[1], **Shashank Gupta**[2], **Bodhisattwa Prasad Majumder**[5], **Katherine Hermann**[6], **Sean Welleck**[2,3], **Amir Yazdanbakhsh**[6], **Peter Clark**[2]
[1]Language Technologies Institute, Carnegie Mellon University
[2]Allen Institute for Artificial Intelligence
[3]University of Washington [4]NVIDIA [5]UC San Diego [6]Google Research, Brain Team
`amadaan@cs.cmu.edu, nikett@allenai.org`

Abstract

Like humans, large language models (LLMs) do not always generate the best output on their first try. Motivated by how humans refine their written text, we introduce SELF-REFINE, an approach for improving initial outputs from LLMs through iterative feedback and refinement. The main idea is to generate an initial output using an LLM; then, the same LLM provides *feedback* for its output and uses it to *refine* itself, iteratively. SELF-REFINE does not require any supervised training data, additional training, or reinforcement learning, and instead uses a single LLM as the generator, refiner and the feedback provider. We evaluate SELF-REFINE across 7 diverse tasks, ranging from dialog response generation to mathematical reasoning, using state-of-the-art (GPT-3.5 and GPT-4) LLMs. Across all evaluated tasks, outputs generated with SELF-REFINE are preferred by humans and automatic metrics over those generated with the same LLM using conventional one-step generation, improving by $\sim$20% absolute on average in task performance. Our work demonstrates that even state-of-the-art LLMs like GPT-4 can be further improved at test-time using our simple, standalone approach.[1].

1 Introduction

Although large language models (LLMs) can generate coherent outputs, they often fall short in addressing intricate requirements. This mostly includes tasks with multifaceted objectives, such as dialogue response generation, or tasks with hard-to-define goals, such as enhancing program readability. In these scenarios, modern LLMs may produce an intelligible initial output, yet may benefit from further iterative refinement—i.e., iteratively mapping a candidate output to an improved one—to ensure that the desired quality is achieved. Iterative refinement typically involves training a refinement model that relies on domain-specific data (e.g., Reid and Neubig (2022); Schick et al. (2022a); Welleck et al. (2022)). Other approaches that rely on external supervision or reward models require large training sets or expensive human annotations (Madaan et al., 2021; Ouyang et al., 2022), which may not always be feasible to obtain. These limitations underscore the need for an effective refinement approach that can be applied to various tasks without requiring extensive supervision.

Iterative *self*-refinement is a fundamental characteristic of human problem-solving (Simon, 1962; Flower and Hayes, 1981; Amabile, 1983). Iterative self-refinement is a process that involves creating an initial draft and subsequently refining it based on self-provided feedback. For example, when

[1]Code and data at `https://selfrefine.info/`

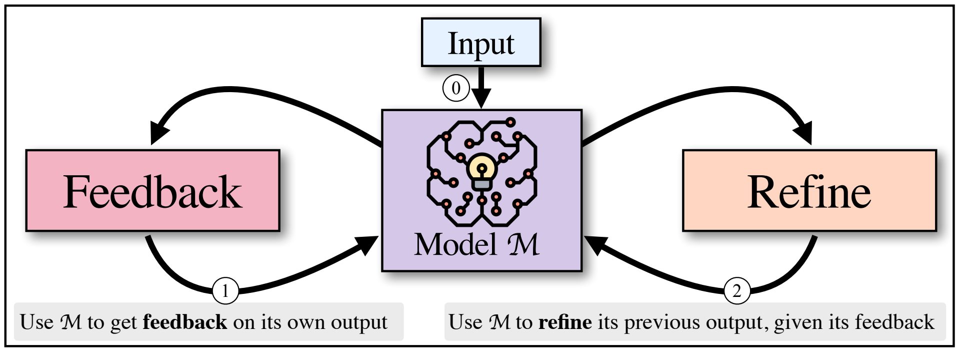

Input
0
Feedback
Model $\mathcal{M}$
Refine
1
2
Use $\mathcal{M}$ to get **feedback** on its own output
Use $\mathcal{M}$ to **refine** its previous output, given its feedback

Figure 1: Given an input (⓪), SELF-REFINE starts by generating an output and passing it back to the same model $\mathcal{M}$ to get feedback (①). The feedback is passed back to $\mathcal{M}$, which refines the previously generated output (②). Steps (①) and (②) iterate until a stopping condition is met. SELF-REFINE is instantiated with a language model such as GPT-3.5 and does not involve human assistance.

drafting an email to request a document from a colleague, an individual may initially write a direct request such as "*Send me the data ASAP*". Upon reflection, however, the writer recognizes the potential impoliteness of the phrasing and revises it to "*Hi Ashley, could you please send me the data at your earliest convenience?*". When writing code, a programmer may implement an initial "quick and dirty" implementation, and then, upon reflection, refactor their code to a solution that is more efficient and readable. In this paper, we demonstrate that LLMs can provide iterative self-refinement without additional training, leading to higher-quality outputs on a wide range of tasks.

We present SELF-REFINE: an iterative self-refinement algorithm that alternates between two generative steps–FEEDBACK and REFINE. These steps work in tandem to generate high-quality outputs. Given an initial output generated by a model $\mathcal{M}$, we pass it back to the same model $\mathcal{M}$ to get *feedback*. Then, the feedback is passed back to the same model to *refine* the previously-generated draft. This process is repeated either for a specified number of iterations or until $\mathcal{M}$ determines that no further refinement is necessary. We use few-shot prompting (Brown et al., 2020) to guide $\mathcal{M}$ to both generate feedback and incorporate the feedback into an improved draft. Figure 1 illustrates the high-level idea, that SELF-REFINE *uses the same underlying language model to generate feedback and refine its outputs*.

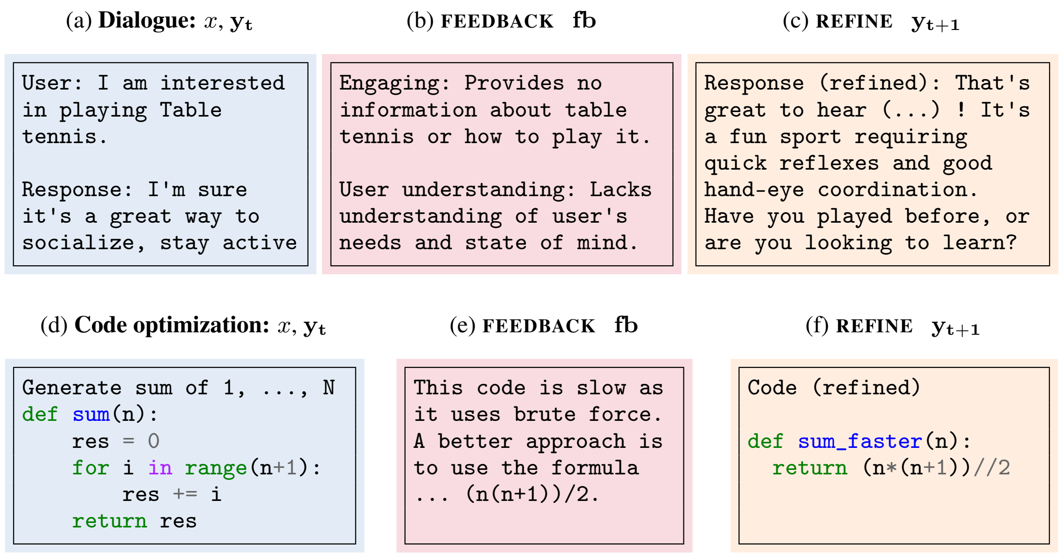
We evaluate SELF-REFINE on 7 generation tasks that span diverse domains, including natural language and source-code generation. We show that SELF-REFINE outperforms direct generation from strong LLMs like GPT-3.5 (`text-davinci-003` and `gpt-3.5-turbo`; OpenAI; Ouyang et al., 2022) and GPT-4 (OpenAI, 2023) by 5-40% absolute improvement. In code-generation tasks, SELF-REFINE improves the initial generation by up to absolute 13% when applied to strong code models such as Codex (`code-davinci-002`; Chen et al., 2021). We release all of our code, which is easily extensible to other LLMs. In essence, our results show that even when an LLM cannot generate an optimal output on its first try, the LLM can often provide useful feedback and improve its own output accordingly. In turn, SELF-REFINE provides an effective way to obtain better outputs from a single model without any additional training, via iterative (self-)feedback and refinement.

2 Iterative Refinement with SELF-REFINE

Given an input sequence, SELF-REFINE generates an initial output, provides feedback on the output, and refines the output according to the feedback. SELF-REFINE iterates between feedback and refinement until a desired condition is met. SELF-REFINE relies on a suitable language model and three prompts (for initial generation, feedback, and refinement), and does not require training. SELF-REFINE is shown in Figure 1 and Algorithm 1. Next, we describe SELF-REFINE in more detail.

Initial generation Given an input x, prompt p_{gen}, and model $\mathcal{M}$, SELF-REFINE generates an initial output y_0:

$$y_0 = \mathcal{M}\left(p_{\text{gen}} \| x\right). \tag{1}$$


(a) **Dialogue:** x, $\mathbf{y_t}$

```
User: I am interested
in playing Table
tennis.

Response: I'm sure
it's a great way to
socialize, stay active
```

(b) **FEEDBACK** $\mathbf{fb}$

```
Engaging: Provides no
information about table
tennis or how to play it.

User understanding: Lacks
understanding of user's
needs and state of mind.
```

(c) **REFINE** $\mathbf{y_{t+1}}$

```
Response (refined): That's
great to hear (...) ! It's
a fun sport requiring
quick reflexes and good
hand-eye coordination.
Have you played before, or
are you looking to learn?
```

(d) **Code optimization:** x, $\mathbf{y_t}$

```
Generate sum of 1, ..., N
def sum(n):
    res = 0
    for i in range(n+1):
        res += i
    return res
```

(e) **FEEDBACK** $\mathbf{fb}$

```
This code is slow as
it uses brute force.
A better approach is
to use the formula
... (n(n+1))/2.
```

(f) **REFINE** $\mathbf{y_{t+1}}$

```
Code (refined)

def sum_faster(n):
  return (n*(n+1))//2
```

Figure 2: Examples of SELF-REFINE: an initial output 🟦 generated by the base LLM and then passed back to the *same* LLM to receive feedback 🟥 to the *same* LLM to refine the output 🟧. The top row illustrates this for dialog generation where an initial dialogue response can be transformed into a more engaging one that also understands the user by applying feedback. The bottom row illustrates this for code optimization where the code is made more efficient by applying feedback.

Algorithm 1 SELF-REFINE algorithm

```
Require: input $x$, model $\mathcal{M}$, prompts $\{p_{\text{gen}}, p_{\text{fb}}, p_{\text{refine}}\}$, stop condition $\text{stop}(\cdot)$
 1: $y_0 = \mathcal{M}(p_{\text{gen}} \| x)$                                  ▷ Initial generation (Eqn. 1)
 2: for iteration t $\in 0, 1, \ldots$ do
 3:     $fb_t = \mathcal{M}\left(p_{\text{fb}} \| x \| y_t\right)$                      ▷ Feedback (Eqn. 2)
 4:     if $\text{stop}(fb_t, t)$ then                                  ▷ Stop condition
 5:         break
 6:     else
 7:         $y_{t+1} = \mathcal{M}\left(p_{\text{refine}} \| x \| y_0 \| fb_0 \| \ldots \| y_t \| fb_t\right)$    ▷ Refine (Eqn. 4)
 8:     end if
 9: end for
10: return $y_t$
```

Figure 3: The SELF-REFINE algorithm. See (§2) for a discussion of each component.

For example, in Figure 2(d), the model generates functionally correct code for the given input. Here, p_{gen} is a task-specific few-shot prompt (or instruction) for an initial generation, and $\|$ denotes concatenation. The few-shot prompt contains input-output pairs $\langle x^{(k)}, y^{(k)} \rangle$ for the task.[2]

FEEDBACK Next, SELF-REFINE uses the same model $\mathcal{M}$ to provide feedback fb_t on its own output, given a task-specific prompt p_{fb} for generating feedback:

$$fb_t = \mathcal{M}\left(p_{\text{fb}} \| x \| y_t\right). \tag{2}$$

Intuitively, the feedback may address multiple aspects of the output. For example, in code optimization, the feedback might address the efficiency, readability, and overall quality of the code.

[2]Few-shot prompting (also referred to as "in-context learning") provides a model with a prompt consisting of k in-context examples of the target task, each in the form of input-output pairs $\langle x_i, y_i \rangle$ (Brown et al., 2020).