قدم اول: تعریف دقیق مسئله، فضای جستجو و نقش LLM

در مسیر دوم (بهینه‌سازی پرامپت)، ما با یک مسئله بهینه‌سازی روی «رشته‌ها» (Strings) روبه‌رو هستیم. باید مشخص کنیم سیستم دقیقاً چه چیزی می‌گیرد، چطور پرامپت‌های جدید می‌سازد و چطور آن‌ها را ارزیابی می‌کند .

من این ساختار پایه را برای گزارش فاز اولت پیشنهاد می‌دهم:

۱. تعریف ورودی و خروجی (Input/Output):

- **ورودی (Input)**: 1. یک پرامپت خام و اولیه (مثلاً: *"متن زیر را خلاصه کن"*).

2. یک مجموعه داده کوچک برای تست (شامل چند نمونه متن + خلاصه‌های طلایی و درست آن‌ها).

- **خروجی (Output)**: پرامپت بهینه‌شده‌ای که بیشترین دقت/امتیاز را روی مجموعه داده تست ما می‌گیرد (مثلاً: *"به عنوان یک ویراستار حرفه‌ای، متن زیر را در سه خط و با لحن رسمی خلاصه کن"*).

۲. نقش LLM (ماژول Oracle): در این سیستم، ما از یک LLM (مثل GPT-3.5 یا Gemini API یا یک مدل متن‌باز کوچک) در دو جای الگوریتم استفاده می‌کنیم:

- **تولید کاندید (Candidate Generation/Mutation)**: پرامپت فعلی را به LLM می‌دهیم و از او می‌خواهیم ۳ نسخه متفاوت از آن بنویسد (مثلاً لحن را عوض کند یا کلمات کلیدی اضافه کند). اینطوری "همسایه‌ها" در فضای جستجو ساخته می‌شوند.
- **ارزیاب (Evaluator/Fitness Function)**: پرامپت‌های جدید را روی مجموعه داده تست اجرا می‌کنیم و کیفیت خروجی را می‌سنجیم.

۳. فضای جستجو (Search Space): فضای جستجوی ما تمام ترکیبات ممکن از کلمات برای یک پرامپت است (که بی‌نهایت و بسیار بزرگ است). بنابراین، این یک مسئله **NP-hard** محسوب می‌شود و ما نمی‌توانیم تمام پرامپت‌های دنیا را تست کنیم. به همین دلیل به جای جستجوی کامل (Exhaustive Search)، باید از الگوریتم‌های جستجوی هیوریستیک مثل Beam Search یا الگوریتم‌های بهینه‌سازی محلی مثل Hill Climbing استفاده کنیم .

قدم دوم: طراحی الگوریتم و سودوکد (Pseudocode)

- تابع Evaluate_Prompt: پرامپت را روی مقالات دیتاست اجرا می‌کند، سپس از LLM (نقش اوراکل) می‌خواهد به کیفیت خلاصه تولید شده از ۱ تا ۱۰ نمره بدهد. میانگین نمرات می‌شود امتیاز آن پرامپت.
- تابع LLM_Generate_Mutations: پرامپت فعلی را می‌گیرد و با یک دستور سیستمی (مثلاً "این دستور را به ۳ لحن مختلف بازنویسی کن")، پرامپت‌های جدید تولید می‌کند.

۱. سودوکد الگوریتم Hill Climbing (تپه‌نوردی)

این الگوریتم از یک پرامپت شروع می‌کند، همسایه‌هایش (تغییر یافته‌هایش) را می‌سازد و اگر همسایه‌ای بهتر بود، به آن سمت حرکت می‌کند.

Algorithm 1: Prompt Optimization via Hill Climbing

Input:

- P_initial: Initial raw prompt (e.g., "Summarize the article")
- Dataset: A small set of articles for testing
- Max_Steps: Maximum number of iterations
- N_Mutations: Number of variations to generate per step

Output: P_best (The optimized prompt)

```
P_current = P_initial
Score_current = Evaluate_Prompt(P_current, Dataset) // Uses LLM Oracle to score
summary quality
Step = 0
WHILE Step < Max_Steps DO:
    // Generate candidates using LLM Oracle
    Candidates = LLM_Generate_Mutations(P_current, N_Mutations)
    Best_Candidate_Score = -1
    Best_Candidate = NULL
    FOR each candidate C in Candidates DO:
        Score_C = Evaluate_Prompt(C, Dataset)
        IF Score_C > Best_Candidate_Score THEN:
            Best_Candidate_Score = Score_C
            Best_Candidate = C
    // Move to the better neighbor if it exists
    IF Best_Candidate_Score > Score_current THEN:
        P_current = Best_Candidate
        Score_current = Best_Candidate_Score
    ELSE:
        // Local optimum reached
        BREAK
    Step = Step + 1
  RETURN P_current
```

۲. سودوکد الگوریتم Beam Search

این الگوریتم به جای نگه‌داشتن فقط یک پرامپت، در هر مرحله بهترین B پرامپت را نگه می‌دارد (پهنای پرتو = B) تا در تله‌های محلی (Local Optima) گیر نیفتد.

Algorithm 2: Prompt Optimization via Beam Search

Input:

- P_initial: Initial raw prompt
- Dataset: Validation dataset
- Beam_Width (B): Number of top prompts to keep
- Max_Depth: Maximum search depth
- N_Mutations: Mutations per prompt

Output: Best prompt found

```
Beam = [P_initial]
Depth = 0
WHILE Depth < Max_Depth DO:
    All_Candidates = []

    FOR each prompt P in Beam DO:
        Mutations = LLM_Generate_Mutations(P, N_Mutations)
        Add Mutations to All_Candidates

    // Evaluate all newly generated candidates
    Candidate_Scores = []
    FOR each candidate C in All_Candidates DO:
        Score = Evaluate_Prompt(C, Dataset)
        Add (C, Score) to Candidate_Scores

    // Select top B candidates
    Sort Candidate_Scores in descending order based on Score
    Beam = Top B prompts from Candidate_Scores

    Depth = Depth + 1
 RETURN Best prompt in Beam (Rank 1)
```

قدم سوم: تحلیل های ریاضی

- متغیر N: تعداد دفعات اجرای حلقه (در الگوریتم اول Max_Steps و در الگوریتم دوم Max_Depth).
- متغیر M: تعداد پرامپت‌های جدید (Mutations) تولید شده در هر مرحله.
- متغیر D: اندازه دیتاست ارزیابی (تعداد مقالاتی که برای تست هر پرامپت استفاده می‌کنیم).
- متغیر B: پهنای پرتو (Beam Width) در الگوریتم Beam Search.
- متغیر L_gen: زمان متوسطی که طول می‌کشد LLM یک پرامپت جدید بسازد.
- متغیر L_eval: زمان متوسطی که طول می‌کشد LLM یک متن را خلاصه کرده و ارزیابی کند.

۱. تحلیل ریاضی الگوریتم Hill Climbing (تپه‌نوردی):

تحلیل پیچیدگی زمانی (Time Complexity):

در این الگوریتم، در هر مرحله M همسایه جدید تولید می‌کنیم که هزینه آن O(M.L_gen) است. سپس هر کدام از این M پرامپت را روی D مقاله تست می‌کنیم که هزینه آن O(M.D.L_eval) می‌شود.

- **بدترین حالت (Worst-Case)**: الگوریتم تا آخرین مرحله ممکن یعنی N گام پیش برود و متوقف نشود.

Time_worst = O(N.M.(L_gen + D.L_eval))

- **بهترین حالت (Best-Case)**: الگوریتم در همان گام اول به یک نقطه بهینه محلی (Local Optimum) برسد و هیچ پرامپت بهتری پیدا نکند و حلقه BREAK شود.

Time_best = O(M.(L_gen + D.L_eval))

تحلیل پیچیدگی فضایی (Space Complexity):

- در هر لحظه فقط پرامپت فعلی و M پرامپت کاندید جدید را در حافظه نگه می‌داریم. بنابراین فضای جستجو خطی رشد می‌کند.

Space = O(M)

۲. تحلیل ریاضی الگوریتم Beam Search (جستجوی شعاعی)

تحلیل پیچیدگی زمانی (Time Complexity):

در هر عمق از جستجو، ما B پرامپت برتر داریم. برای هر کدام از آن‌ها M همسایه می‌سازیم (در مجموع B.M پرامپت جدید). ارزیابی همه این‌ها روی دیتاست به اندازه O(B.M.D.L_eval) زمان می‌برد. سپس برای انتخاب B پرامپت برتر، باید لیست کاندیداها را مرتب‌سازی (Sort) کنیم که هزینه آن O((B.M)log(B.M)) است.

- **بدترین حالت و بهترین حالت:** چون Beam Search معمولاً تا عمق مشخصی (N) پیش می‌رود تا فضای بیشتری را بگردد، پیچیدگی آن ثابت‌تر است.

Time = O(N.[B.M.(L_gen + D.L_eval) + (B.M) log(B.M)])

نکته برای ارائه: معمولاً زمان پاسخگویی API مدل زبانی (L_eval) آن‌قدر بزرگ است که زمان مرتب‌سازی آرایه در برابر آن ناچیز می‌شود و می‌توانیم بخش لگاریتمی را در تحلیل نهایی نادیده بگیریم.

تحلیل پیچیدگی فضایی (Space Complexity):

- در الگوریتم Beam Search، ما باید در هر مرحله B پرامپت اصلی و B.M پرامپت تولید شده جدید را همراه با امتیازهایشان در حافظه سیستم (آرایه‌ها) نگه داریم.

Space = O(B.M)

قدم چهارم: طراحی تست‌های دستی (Manual Tests)

۱. تست ساده (Simple Case)

- **هدف تست:** بررسی عملکرد الگوریتم روی یک متن کوتاه و سرراست.
- **ورودی (Input):**
  - *پرامپت اولیه:* «متن را خلاصه کن.»
  - *مقاله نمونه:* یک خبر کوتاه ورزشی ۴ خطی درباره نتیجه یک مسابقه فوتبال.
- **خروجی مورد انتظار (Output – پرامپت بهینه‌شده):** «متن زیر را در یک پاراگراف کوتاه خلاصه کن و نتیجه نهایی مسابقه را بگو.»
- **دلیل:** در متن‌های ساده، الگوریتم باید یاد بگیرد که پرامپت را محدودتر کند تا LLM زیاده‌گویی نکند.

۲. تست متوسط (Medium Case)

- **هدف تست:** بررسی توانایی الگوریتم در استخراج اطلاعات کلیدی از متون ساختاریافته.
- **ورودی (Input):**
  - *پرامپت اولیه:* «لطفا این مقاله را به صورت خلاصه بنویس.»
  - *مقاله نمونه:* یک مقاله ویکی‌پدیا (حدود ۳ صفحه) درباره یک واقعه تاریخی شامل تاریخ‌ها و نام افراد.
- **خروجی مورد انتظار (Output – پرامپت بهینه‌شده):** «مهم‌ترین وقایع تاریخی، تاریخ‌های کلیدی و نام افراد تأثیرگذار در متن زیر را استخراج کرده و در قالب ۳ تا ۵ خط خلاصه کن.»
- **دلیل:** الگوریتم با دریافت امتیازهای پایین برای خلاصه‌های کلیشه‌ای، پرامپت کاندیدی را انتخاب می‌کند که صراحتاً درخواست استخراج «تاریخ» و «نام» را دارد.

۳. تست سخت (Hard Case)

- **هدف تست:** هندل کردن متون تخصصی با اصطلاحات پیچیده که LLM معمولاً در خلاصه‌سازی آن‌ها دچار توهم (Hallucination) می‌شود.
- **ورودی (Input):**
  - *پرامپت اولیه:* «خلاصه متن.»
  - *مقاله نمونه:* چکیده یک مقاله علمی-پژوهشی سنگین در حوزه فیزیک کوانتوم پر از فرمول و اصطلاحات تخصصی.
- **خروجی مورد انتظار (Output – پرامپت بهینه‌شده):** «به عنوان یک استاد دانشگاه، متن تخصصی زیر را برای یک دانشجوی سال اول به زبان ساده و قابل فهم خلاصه کن، اما حتماً اصطلاحات علمی کلیدی را بدون تغییر حفظ کن.»
- **دلیل:** در متون سخت، پرامپت باید مفهوم «پرسونا» (مثل استاد دانشگاه) و دستورالعمل‌های متناقض (ساده‌سازی + حفظ اصطلاحات) را به LLM تزریق کند تا بالاترین امتیاز را از تابع ارزیاب (Evaluator) بگیرد.

۴. تست لبه - حالت اول (Edge Case 1: متن بسیار کوتاه)

- **هدف تست:** رفتار الگوریتم وقتی مقاله‌ای برای خلاصه‌سازی وجود ندارد یا بسیار کوتاه است.
- **ورودی (Input):**
  - *پرامپت اولیه:* «متن زیر را خلاصه کن.»
  - *مقاله نمونه:* «امروز هوا آفتابی است.» (فقط یک جمله).

- **خروجی مورد انتظار (Output – رفتار الگوریتم):** پرامپت تغییر زیادی نمی‌کند یا به چیزی شبیه به این تبدیل می‌شود: «اگر متن کوتاه‌تر از دو خط است، عیناً همان را تکرار کن، در غیر این صورت آن را خلاصه کن.»
- **دلیل:** تابع ارزیاب ما (LLM Oracle) باید به خلاصه‌هایی که از خودشان اطلاعات می‌سازند (چون متن کوتاه بوده) نمره صفر بدهد.

۵. تست لبه - حالت دوم (Edge Case 2: فروپاشی پرامپت یا Prompt Drift)

- **هدف تست:** مقاومت الگوریتم جستجو (مثل Hill Climbing) در برابر تولید کاندیداهای نامفهوم توسط LLM.
- **ورودی (Input):**
  - *پرامپت فعلی:* «متن را به صورت خلاصه بنویس.»
  - *کاندیدای تولید شده توسط تابع Mutation:* «خلاصه خلاصه Summary متن متن 123!» (یک پرامپت خراب و بی‌معنی).
- **خروجی مورد انتظار (Output – رفتار الگوریتم):** الگوریتم این کاندیدا را روی دیتاست تست می‌کند. LLM ارزیاب به دلیل کیفیت افتضاح خروجی، امتیاز بسیار پایینی (مثلاً ۱ از ۱۰) به آن می‌دهد. الگوریتم این همسایه را **رد (Reject)** می‌کند و پرامپت فعلی را نگه می‌دارد.
- **دلیل:** تضمین اینکه جستجوی ما همیشه رو به جلو است و با جهش‌های اشتباه، کل سیستم خراب نمی‌شود.