

AGENTES AUTÔNOMOS: ESTRATÉGIAS DE RACIOCÍNIO EM LLMs

Nome do Grupo: **Grupo_294**

Participantes do Grupo:

Nome	E-mail	Telefone
Larissa	larissagarcialp.lg@gmail.com	+5591981623492
Filipe	fanmello@gmail.com	+5531984204490
Jair	jairmilani21@gmail.com	+5561982745892
Wesley	mdisolucoestecnologia@gmail.com	+5513982296448
Akila	akylaaquino@hotmail.com	+5588981904521
Samuel	fisica141@gmail.com	+5524988790300
Isis	isis.coelho@meta.com.br	+5555996718633

1. INTRODUÇÃO

Este relatório apresenta uma visão abrangente das principais estratégias de raciocínio aplicadas em Modelos de Linguagem de Grande Escala (LLMs), conforme explorado no curso Inteligência Artificial Aplicada. Com base nos estudos realizados, o objetivo é descrever como cada uma dessas estratégias contribui para a geração de respostas mais coerentes, interpretáveis e eficazes em tarefas complexas, por meio da realização de testes práticos. Acreditamos que o domínio dessas abordagens é fundamental não apenas para o desenvolvimento de agentes autônomos inteligentes, mas também para sua aplicação consciente e eficiente em atividades do cotidiano profissional e acadêmico.

2. ESTRATÉGIAS DE RACIOCÍNIO ESTUDADAS

2.1. Estratégia de Treinamento (Ajuste de Parâmetros)

As Estratégias de Treinamento por Ajuste de Parâmetros consistem em modificar os pesos internos de um modelo de linguagem por meio de re-treinamento com dados rotulados. Essa abordagem, como no Fully Supervised Finetuning, permite que o modelo aprenda tarefas específicas de forma permanente. É usada quando se deseja alta precisão e desempenho consistente. Exige maior custo computacional e preparação de dados.

2.1.1. Fully Supervised Finetuning

Essa técnica consiste em continuar o treinamento de um modelo pré-treinado utilizando um conjunto de dados rotulados, onde cada entrada tem uma saída esperada.

Objetivo: Realizar o fine-tuning do modelo pré-treinado “bert-base-uncased” com uma abordagem supervisionada completa (Fully Supervised Fine-tuning) para a tarefa de classificação de sentimentos em frases curtas, rotuladas como positivas (1), negativas (0) ou neutras (2).

Vantagens: Alto desempenho em tarefas especializadas.

Desvantagens: Requer grandes volumes de dados rotulados e limita a generalização, pois o modelo pode ficar restrito ao domínio de treinamento.

Etapas de desenvolvimento:

Preparação dos Dados:

Iniciamos com a criação de um conjunto de dados composto por frases rotuladas manualmente. Inicialmente foram fornecidas 10 frases, e em seguida, foram adicionadas mais 20 frases, balanceando entre as três classes (positivo, negativo, neutro). O pandas foi usado para compor o DataFrame, e a biblioteca datasets do HuggingFace transformou esses dados em um DatasetDict com divisão 24/6 para treino e teste.

Tokenização:

Utilizamos o AutoTokenizer do HuggingFace para tokenizar as frases com truncamento, padding e limite de 64 tokens por entrada. A tokenização foi aplicada tanto ao conjunto de treino quanto ao de teste.

Fine-tuning do Modelo:

Foi usado o modelo pré-treinado bert-base-uncased com a classe AutoModelForSequenceClassification, ajustando para 3 rótulos de saída. O treinamento foi conduzido por meio da classe Trainer da biblioteca transformers, com os seguintes parâmetros principais:

```
[ ] from transformers import TrainingArguments, Trainer

training_args = TrainingArguments(
    output_dir="/content/model",
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    num_train_epochs=50,
    weight_decay=0.01,
    logging_dir="/content/log",
    logging_steps=10,
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["test"],
)

trainer.train()
trainer.save_model("/content/model_final")
```

[300/300 02:07, Epoch 50/50]

Step	Training Loss
10	1.110700
20	0.898900
30	0.839600
40	0.504200
50	0.196800
60	0.065800
70	0.020200
80	0.009900
90	0.005300
100	0.003900

Durante o treinamento, checkpoints foram salvos automaticamente, incluindo os pesos do modelo, configuração e tokenizer.

Implementação e testes:

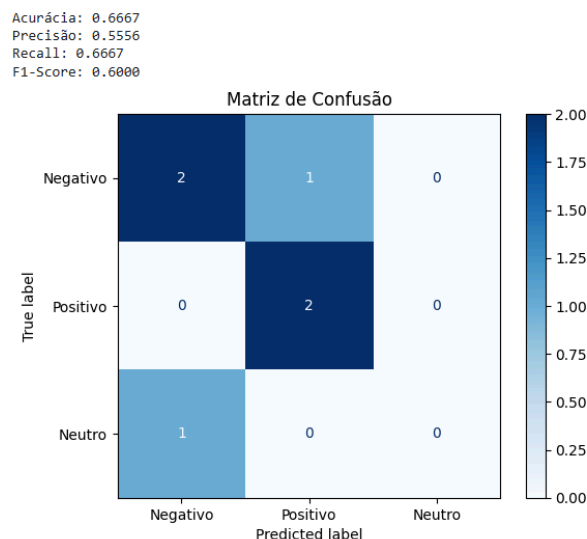
Após o treinamento, o modelo foi carregado com sucesso a partir da pasta /content/model_final, onde estavam salvos os arquivos config.json, tokenizer.json, model.safetensors, entre outros.

O modelo foi testado com diversas frases fora do conjunto de treino para verificar sua capacidade de generalização. A função predict_sentiment usou

softmax e argmax para obter a classe mais provável. As previsões foram relativamente coerentes com o sentimento expressado nas frases, considerando a quantidade pequena dos dados utilizados no treinamento.

```
Frases: Total desperdício de dinheiro | Predição: Negativo
Frases: Muito bom, recomendo! | Predição: Positivo
Frases: Achei o produto razoável | Predição: Neutro
Frases: Excelente qualidade | Predição: Negativo
Frases: Preço alto para o que é | Predição: Negativo
Frases: Gostei bastante | Predição: Positivo
```

Avaliação Quantitativa: utilizamos o conjunto de testes para calcular métricas de avaliação: acurácia, precisão, recall e F1-score. Também foi plotada uma matriz de confusão, mostrando o desempenho do modelo em cada classe. O modelo demonstrou capacidade regular de acerto, especialmente nas classes positivas e negativas, com leve confusão na classe neutra, que é esperada em problemas de interpretação subjetiva.



Conclusão

O fine-tuning supervisionado com o modelo pré-treinado “bert-base-uncased” demonstrou ser eficiente para a tarefa de classificação de sentimentos em português, mesmo com um conjunto pequeno e simples de frases. O processo teve acertos superiores a 60%, com treinamento estável e resultados satisfatórios. Futuros aprimoramentos podem incluir mais dados, uso de data augmentation e ajuste fino dos hiperparâmetros.

2.2. Estratégias de Inferência (sem alteração de parâmetros)

As Estratégias de Inferência consistem em orientar o comportamento do modelo utilizando apenas a entrada textual (*prompt*), sem alterar seus parâmetros internos. Técnicas como Prompting, In-Context Learning e suas ramificações permitem que o modelo execute tarefas com base em instruções e exemplos fornecidos no momento da inferência. São flexíveis, rápidas e não exigem re-treinamento. Ideais para prototipagem e uso interativo.

2.2.1. Prompting

Prompting é a estratégia fundamental de interação com modelos de linguagem de grande escala (LLMs), baseada na formulação de entradas textuais (prompts) que instruem o modelo a realizar uma tarefa específica. Em vez de reconfigurar ou reprogramar o modelo internamente, o Prompting atua apenas no nível da entrada: ao fornecer um comando, pergunta, contexto ou exemplo bem estruturado, o modelo é guiado a produzir uma saída coerente com a intenção do usuário.

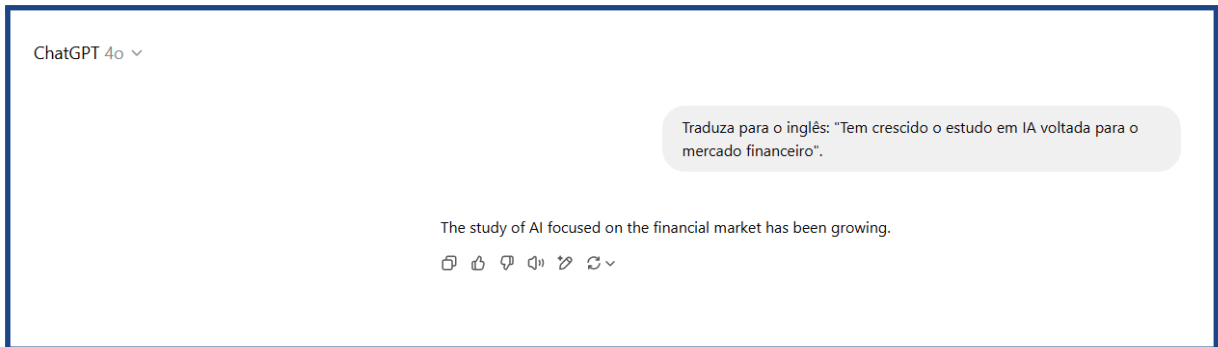
Essa técnica é possível porque os LLMs foram treinados em grandes quantidades de texto e são capazes de generalizar padrões linguísticos e semânticos. Assim, ao receber um prompt, o modelo busca prever a sequência de texto mais provável que continue ou responda àquele comando.

O prompting pode variar desde instruções simples (como “Traduza para o inglês: ‘Bom dia’”) até estruturas mais elaboradas com múltiplos exemplos e instruções, conhecidas como few-shot prompting. Há também variantes como zero-shot prompting (sem exemplos) e one-shot prompting (com um único exemplo), que fazem parte das estratégias de In-Context Learning.

A eficácia do prompting depende da clareza, especificidade e estrutura da entrada. Pequenas variações no texto podem alterar significativamente a qualidade e a utilidade da resposta gerada. Por isso, desenvolver bons prompts tornou-se uma habilidade essencial na engenharia de sistemas baseados em LLMs.

Em resumo, Prompting é uma técnica leve, flexível e poderosa, que permite aproveitar a inteligência do modelo sem a necessidade de treinamento adicional, tornando-se a base para diversas estratégias de raciocínio mais complexas.

A seguir testarmos um comando direto, claro e específico:



Esse é um exemplo de zero-shot prompting, pois o modelo não recebe nenhum exemplo anterior. A saída textual é coerente com o comando fornecido.

2.2.2. In-Context Learning

A estratégia de In-Context Learning utiliza o raciocínio em LLMs em que o modelo aprende uma nova tarefa no momento da inferência, apenas com base em exemplos e instruções fornecidos no **prompt** — ou seja, sem atualizar seus pesos internos, como ocorre no fine-tuning tradicional.

Em vez de reprogramar ou retreinar o modelo, você "ensina" o comportamento desejado dentro da própria entrada (prompt), fornecendo exemplos explícitos de entrada-saída.

Características principais:

- Não exige re-treinamento.
- Baseado em exemplos:
 - Zero-shot: só dá a instrução;
 - One-shot: 1 exemplo + a frase a classificar;
 - Few-shot: 2 ou mais exemplos no mesmo prompt.
- O modelo aprende o padrão de mapeamento a partir do contexto oferecido e tenta generalizar esse padrão para novos casos.

Exemplo de aplicação: consideremos ensinar o modelo a traduzir frases simples do português para o inglês utilizando a abordagem In-Context Learning, sem fazer nenhum ajuste interno.



Resultado: O modelo aprende a estrutura da tarefa com base nos três primeiros pares e generaliza para o quarto item.

Essa abordagem exige a configuração de prompt, não é necessário re-treinamento, pode-se aplicar os formatos: Zero-shot, One-shot, Few-shot, é flexível, aplicação rápida, é útil para prototipagem. Como limitação pode-se citar a limitação de tokens, sensibilidade aos exemplos e a qualidade do prompt.

2.3. Especializações

As estratégias a seguir não são totalmente independentes, mas sim ramificações, especialização ou extensão das estratégias de Prompting e/ou In-Context Learning.

2.3.1. Rationale Engineering

Rationale Engineering é a prática de projetar prompts e sistemas de interação com LLMs para incentivar ou forçar a geração de raciocínios explícitos — ou seja, fazer com que o modelo explique passo a passo como chegou à resposta final, em vez de simplesmente dar a resposta direta.

Essa estratégia ajuda a:

- Tornar a resposta mais transparente e compreensível.
- Permitir verificação lógica do processo.
- Aumentar a confiança do usuário nos sistemas baseados em IA.

Objetivos principais

- Induzir explicações justificadas no output.
- Simular o raciocínio humano estruturado.
- Reduzir erros de *alucinação* ou inferência precipitada.

Rationale Engineering está bastante relacionado com as técnicas:

- Chain-of-Thought Prompting (CoT): uma das formas mais comuns de aplicar rationale.
- Justificativas de múltipla escolha: explicar por que a alternativa correta foi escolhida e as outras descartadas.
- Explicações em classificações: indicar os critérios de decisão usados pelo modelo.

Práticas para melhor aplicar a Rationale Engineering:

Técnica aplicada	Exemplo no prompt
Use instruções claras	“Explique o seu raciocínio.” ou “Justifique a resposta.
Use exemplos ilustrativos	Mostre uma pergunta com raciocínio completo antes da nova
Divida em etapas	Passo 1: ... Passo 2: ...
Peça comparação	Explique por que esta opção é melhor que as demais.
Incentive autoavaliação	Você tem certeza da resposta? Por quê?

Essa abordagem utiliza o aprendizado “Explícito via prompt”, não necessariamente são utilizados exemplos, no entanto seu uso pode ajudar, reforça o raciocínio incentivando a coerência e transparência. Aplicações comuns em casos reais englobam: matemática, raciocínio lógico, tomada de decisão, perguntas abertas, justificativas em múltipla escolha. Como vantagem pode-se mencionar a transparência, confiabilidade e auditabilidade. Falando de limitações pode ocorrer um aumento de custo computacional (mais tokens), nem sempre garante a razão perfeita.

Exemplo de um Problema matemático:

Pergunta: João tem 8 balas. Ele comeu 3 e ganhou mais 5. Quantas balas ele tem agora?
Explique seu raciocínio passo a passo.

Vamos resolver passo a passo:

1. **Quantidade inicial de balas:** João começou com 8 balas.
2. **João comeu 3 balas:** Se ele comeu 3, precisamos subtrair:
 $8 - 3 = 5$ balas.
3. **João ganhou mais 5 balas:** Agora somamos as que ele ganhou:
 $5 + 5 = 10$ balas.

Resposta final: João tem agora **10 balas**.



gABrDeHZDNGAwa3cES

Hoje

Pergunta: João tem 8 balas. Ele comeu 3 e ganhou mais 5. Quantas balas ele tem agora?
Explique seu raciocínio passo a passo.

Vamos analisar a situação passo a passo:

1. **Quantidade inicial:** João começa com 8 balas.
2. **Balas consumidas:** Ele comeu 3 balas, então precisamos subtrair isso do total inicial:

$$8 - 3 = 5$$

Agora ele tem 5 balas.

3. **Balas recebidas:** João ganhou mais 5 balas, então precisamos adicionar isso ao total que ele tinha:

$$5 + 5 = 10$$

Portanto, agora João tem **10 balas!** 🎉 😊

Se precisar de mais explicações, é só perguntar!



Mensagem para o Copilot



Resposta rápida ▾



Essa resposta é desejável porque:

- Mostra o processo lógico;
- Expõe os cálculos intermediários;
- Permite ao usuário verificar cada etapa;
- Justifica a resposta final.

Resumidamente envolve projetar o prompt de modo que o modelo gere raciocínios estruturados, justificados e explicáveis. Está totalmente ligada à engenharia de prompt para induzir raciocínio (é uma ramificação de Prompting), pode ser combinada com In-Context Learning.

2.3.2. Chain-of-Thought Prompting

O Chain-of-Thought Prompting (Prompting em Cadeia de Pensamento) é uma técnica de prompt engineering usada para fazer com que o modelo explique o processo de raciocínio passo a passo, em vez de apenas fornecer a resposta final.

Essa técnica simula o modo de pensar humano, no qual decisões ou soluções são obtidas por meio de várias etapas intermediárias, tornando o raciocínio explícito, interpretável e menos propenso a erros.

O objetivo principal dessa estratégia é incentivar o modelo a “pensar em voz alta”, descrevendo o caminho lógico que leva à resposta, o que melhora significativamente a precisão em tarefas complexas.

No teste executado foi apresentado no prompt um comando (Explique passo a passo como você chegou à resposta), em seguida a pergunta de cunho matemático e na sequência foi dado o comando de resposta. O modelo então atendeu a solicitação respondendo de forma explicativa em passo-a-passo como chegou na resposta.

ChatGPT 4o ▾

Compartilhar

Explique passo a passo como você chegou à resposta.

Pergunta: João tem 3 maçãs. Ele ganha mais 5 maçãs de um amigo. Quantas maçãs João tem agora?

Resposta:

Pergunta: João tem 3 maçãs. Ele ganha mais 5 maçãs de um amigo. Quantas maçãs João tem agora?







ChatGPT 4o ▾

Passo a passo para chegar à resposta:

1. Identificar a quantidade inicial de maçãs:
João começa com 3 maçãs.
2. Identificar a quantidade adicional recebida:
Ele ganha mais 5 maçãs de um amigo.
3. Somar as quantidades:
Para saber quantas maçãs João tem agora, somamos as duas quantidades:

$$3 + 5 = 8$$

Resposta: João tem agora 8 maçãs.

Para melhorar a consistência, pode-se usar exemplos resolvidos antes da nova pergunta:







ChatGPT 4o ▾

Exemplo 1:
Pergunta: João tinha 10 balas. Ele deu 4 para seu amigo. Com quantas ficou?
Resposta:
João começou com 10. Deu 4. $10 - 4 = 6$. Resposta: 6.

Agora, resolva:
Pergunta: Ana tinha 12 reais. Gastou 5 e ganhou 8. Quanto tem?
Resposta:

Ana começou com 12 reais. Gastou 5.
 $12 - 5 = 7$. Depois, ganhou 8.
 $7 + 8 = 15$.

Resposta: 15.

Resumidamente é uma ramificação de Prompting + In-Context Learning (quando aplicada com exemplos). Usa prompting para forçar o modelo a pensar passo a passo. Pode ser aplicado com zero-shot ("explique seu raciocínio") ou few-shot, fazendo parte do In-Context Learning com exemplos de raciocínio.

2.3.3. Self-Consistency Sampling

O Self-Consistency Sampling é uma estratégia de raciocínio em LLMs que consiste em gerar múltiplas respostas para a mesma pergunta (geralmente com variação estocástica), e depois escolher a resposta mais frequente (a mais "consistente") entre elas.

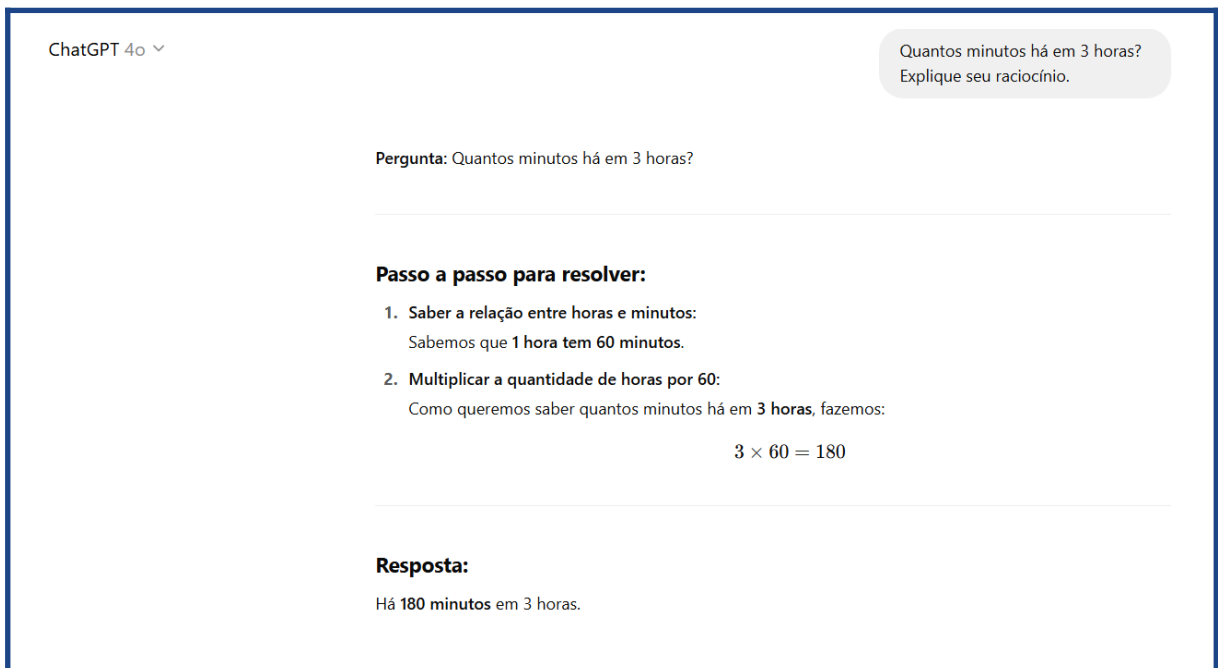
A ideia é que, ao perguntar a mesma coisa várias vezes, o modelo pode explorar diferentes caminhos de raciocínio — e a resposta mais comum tende a ser a mais correta.

As LLMs podem produzir respostas diferentes para a mesma pergunta quando a temperatura está acima de 0 (introduzindo variabilidade). Isso pode levar a erros ou inconsistências em tarefas complexas. A autoconsistência é uma forma de mitigar isso, nesse ponto o uso da abordagem visa minimizar possíveis erros.

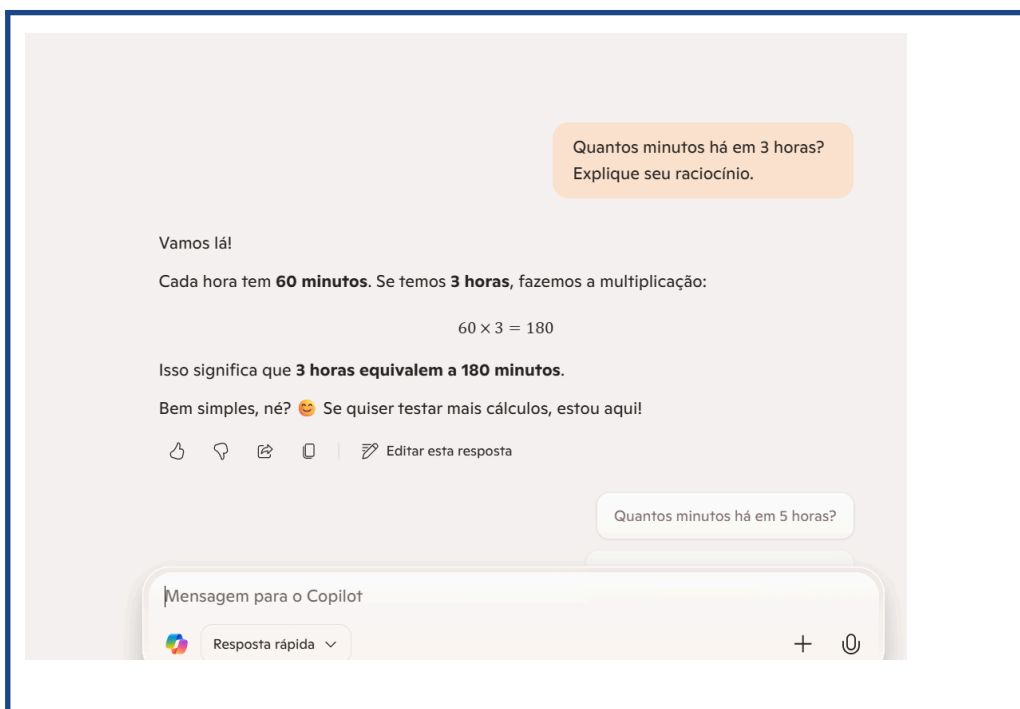
Funciona da seguinte forma:

- É feita a mesma pergunta ao modelo várias vezes;
- O modelo responde com variações no raciocínio e na resposta final;
- Coleta-se todas as respostas;
- A resposta mais frequente (a moda) é selecionada como a mais confiável.

No exemplo a seguir o prompt foi executado por mais de 4x, as respostas foram iguais e consistentes, pode-se concluir que no exemplo utilizado a LLM já possui uma base sólida para responder.



O mesmo teste foi executado em outra LLM:



Em ambas a resposta foi sólida.

O uso da abordagem Self-Consistency Sampling pode ser bem aproveitado nas situações onde:

- Problemas de lógica com múltiplos passos: reduzindo erros ocasionais e melhorando a precisão;
- Questões ambíguas ou abertas: ajudando a encontrar um "consenso do modelo";
- Prompts com Chain-of-Thought: amplificando os efeitos positivos de explicação.

Essa estratégia utiliza a inferência com múltiplas execuções, não requer um novo treinamento, não é necessário a utilização de exemplos, no entanto pode-se combiná-la com Chain-of-Thought ou few-shot prompting. Tem como vantagem redução de erros, aumento de precisão e melhora da confiabilidade. Falando nas suas desvantagens pode-se mencionar o seu alto custo computacional (várias execuções), e a necessidade de pós-processamento.

Em resumo o Self-Consistency Sampling é uma estratégia complementar ao Chain-of-Thought Prompting que aplica variações estocásticas em múltiplas execuções do mesmo prompt, selecionando como resposta final a mais frequente entre as geradas, com o objetivo de aumentar a robustez e a confiabilidade do raciocínio.

2.3.4. Decomposição de Problemas

A técnica de decomposição de problemas é uma estratégia fundamental para lidar com desafios complexos, seja em Inteligência Artificial, ciência da computação, engenharia ou até mesmo no cotidiano. Em sua essência, ela consiste em quebrar um problema grande e difícil em partes menores, mais simples e gerenciáveis.

O objetivo principal da decomposição é tornar o problema original mais fácil de entender, analisar e resolver. Ao dividir o problema em subproblemas menores, cada parte pode ser abordada individualmente, muitas vezes utilizando técnicas ou algoritmos específicos mais adequados à sua natureza.

Principais aspectos da decomposição de problemas:

- Identificação dos componentes: O primeiro passo é analisar o problema como um todo e identificar seus componentes ou subproblemas constituintes.

- Definição de sub-objetivos: Para cada subproblema identificado, define-se um objetivo mais específico e alcançável.
- Resolução individual: Cada subproblema é então resolvido separadamente, utilizando as ferramentas e técnicas apropriadas.
- Integração das soluções: Finalmente, as soluções para os subproblemas são combinadas para formar a solução completa para o problema original.

Benefícios da decomposição de problemas:

- Simplificação da complexidade: Problemas complexos se tornam mais fáceis de entender e abordar.
- Melhora da organização: Facilita a estruturação do trabalho e a atribuição de tarefas.
- Aumento da eficiência: A resolução de problemas menores costuma ser mais rápida e eficiente.
- Facilita a reutilização de soluções: As soluções para subproblemas podem, por vezes, ser reaproveitadas em outros contextos.
- Melhora a depuração e o teste: É mais fácil identificar e corrigir erros em partes menores do que em um sistema complexo como um todo.

Três técnicas de prompting que exploram a decomposição de problemas para melhorar o desempenho de modelos de linguagem (LLMs) são:

1. Lost in the Middle Prompting (ou simplesmente "Middle-Out"): Embora não seja estritamente uma técnica de decomposição explícita, ela reconhece que LLMs podem ter dificuldade em acessar informações cruciais que estão no meio de um prompt longo. Uma forma de mitigar isso é reestruturar o prompt para colocar informações importantes no início e no final, cercando informações menos críticas. Isso indiretamente ajuda o modelo a focar nas partes essenciais para iniciar a decomposição do problema. Em vez de uma decomposição lógica passo a passo, ela otimiza a entrada para que o modelo *internamente* consiga identificar melhor os componentes chave do problema.
2. Decomposed Prompting: Esta técnica envolve explicitamente instruir o LLM a quebrar o problema complexo em subproblemas menores e sequenciais. O

prompt guia o modelo a pensar passo a passo, resolvendo cada subproblema individualmente antes de combinar as soluções para chegar à resposta final. Isso pode ser feito através de perguntas direcionadas em cada etapa ou solicitando ao modelo que liste os passos necessários para resolver o problema. O objetivo é imitar o processo de raciocínio humano ao abordar tarefas complexas, tornando a solução mais transparente e controlável.

3. Successive Prompting: Similar ao Decomposed Prompting, o Successive Prompting também envolve uma abordagem passo a passo. No entanto, em vez de definir todos os subproblemas de uma vez no prompt inicial, cada resposta do modelo para um subproblema serve como o prompt para a próxima etapa. Isso permite uma interação mais dinâmica e adaptativa, onde o modelo pode refinar seu raciocínio e a decomposição do problema com base nas respostas anteriores. É útil para problemas que exigem exploração ou onde os subproblemas exatos podem não ser claros desde o início.

A decomposição de problemas é crucial na modelagem de sistemas inteligentes. Tarefas complexas, como planejamento, reconhecimento de padrões, processamento de linguagem natural e aprendizado complexo, são frequentemente decompostas em subproblemas menores para serem resolvidos por algoritmos específicos. Por exemplo, um sistema de planejamento robótico pode decompor a tarefa de "pegar um objeto em uma sala" em sub-tarefas como "navegar até o objeto", "identificar o objeto", "alcançar o objeto" e "agarrar o objeto".

Em resumo, a decomposição de problemas é uma técnica poderosa que simplifica a resolução de desafios complexos ao dividi-los em partes menores e mais gerenciáveis, facilitando a análise, o desenvolvimento de soluções e a integração dessas soluções para alcançar o objetivo final.

A Decomposição de Problemas é uma ramificação avançada de Prompting que estrutura o prompt para dividir um problema complexo em subproblemas menores e sequenciais, facilitando a resolução por meio de prompting modular e compatível com estratégias de In-Context Learning, especialmente em abordagens few-shot.

Vamos testar uma problema que é possível quebrar em múltiplas etapas:

ChatGPT 4o ▾

Resolva o problema passo a passo, dividindo em etapas menores.

Pergunta: Um aluno fez três provas. Tirou 6, 8 e 7. Qual foi a média? Ele passou, considerando que a média mínima para aprovação é 7?

Vamos resolver o problema passo a passo:

Etapa 1: Identificar as notas

O aluno tirou as seguintes notas:

- Primeira prova: 6
- Segunda prova: 8
- Terceira prova: 7

ChatGPT 4o ▾

Etapa 2: Somar as notas

Vamos somar as três notas:

$$6 + 8 + 7 = 21$$

Etapa 3: Dividir pelo número de provas

Agora dividimos o total pelas 3 provas para calcular a média:

$$21 \div 3 = 7$$






Etapa 4: Comparar com a média mínima

A média mínima para passar é 7. A média do aluno foi 7, então:

Resposta final:

A média foi 7.

Sim, ele passou. 

Os critérios para avaliar a resposta na estratégia de Decomposição de Problemas envolvem verificar se o modelo organiza o raciocínio em etapas lógicas, apresenta soluções corretas para cada subproblema, mantém clareza explicativa e chega a uma conclusão coerente e bem justificada.

2.3.5. Tool-Augmented Reasoning

Tool-Augmented Reasoning é uma estratégia de raciocínio que amplia as capacidades de um modelo de linguagem ao integrá-lo com ferramentas externas, como calculadoras, interpretadores de código, navegadores, mecanismos de busca ou bases de conhecimento. Em vez de depender exclusivamente do que foi aprendido durante o pré-treinamento, o modelo utiliza recursos auxiliares em tempo real para obter, processar ou validar informações antes de concluir sua resposta.

Essa abordagem é considerada uma estratégia híbrida, pois combina elementos de Prompting e In-Context Learning com a execução de subtarefas fora do modelo, geralmente coordenadas por meio de prompts que indicam quando e como uma ferramenta deve ser utilizada. O modelo atua como um agente de orquestração: ele entende a tarefa, decide se precisa de ajuda externa e, então, delega parte da solução a uma ferramenta especializada.

Por exemplo, se o modelo recebe uma pergunta do tipo "qual é a raiz quadrada de 289?", ele pode identificar que se trata de um cálculo e acionar uma calculadora integrada, retornando a resposta correta com base no resultado da ferramenta. Da mesma forma, se precisar buscar a cotação atual do dólar ou consultar a definição de um termo técnico, ele pode acionar um mecanismo de busca.

Na prática, essa estratégia exige que o modelo esteja inserido em uma infraestrutura que suporte integração com ferramentas externas, como agentes de IA (por exemplo, AutoGPT, ReAct ou LangChain) ou plataformas com plug-ins (como o ChatGPT com navegação e código ativado).

Tool-Augmented Reasoning é especialmente útil em tarefas que envolvem:

- Cálculos matemáticos precisos;
- Acesso a informações atualizadas ou específicas (ex: dados da internet);
- Execução de código (interpretação Python, SQL, etc.);
- Processamento de arquivos (ex: leitura de tabelas, PDFs, imagens).

Ao combinar raciocínio linguístico com ferramentas especializadas, essa estratégia aumenta significativamente a precisão, a utilidade e a versatilidade dos modelos,

tornando-os aptos a operar como agentes cognitivos completos, capazes de atuar em ambientes reais e dinâmicos.

Nessa estratégia não foi possível a implementação de um teste, vez que existe a necessidade de preparação de ambiente externo, bem como sua configuração.

2.3.6. Memory and Contextual Reasoning

Memory and Contextual Reasoning é uma estratégia de raciocínio que busca preservar e utilizar o contexto ao longo de múltiplas interações com o modelo, permitindo que ele recupere informações de conversas anteriores ou fatos aprendidos em tempo de execução para construir respostas mais coerentes, personalizadas e consistentes.

Essa abordagem se baseia na ideia de que, para que um agente de linguagem se comporte de forma realmente inteligente, ele precisa lembrar do que já foi dito, perguntado ou decidido anteriormente — algo fundamental para aplicações como assistentes virtuais, tutores inteligentes, sistemas de recomendação e agentes autônomos.

O funcionamento dessa estratégia depende de dois componentes principais:

- Memória episódica: onde são armazenados elementos de interações passadas específicas (ex: "O usuário me pediu para lembrar que seu nome é Carlos").
- Memória semântica: onde se armazenam conceitos gerais ou inferências persistentes (ex: "O usuário gosta de respostas detalhadas").

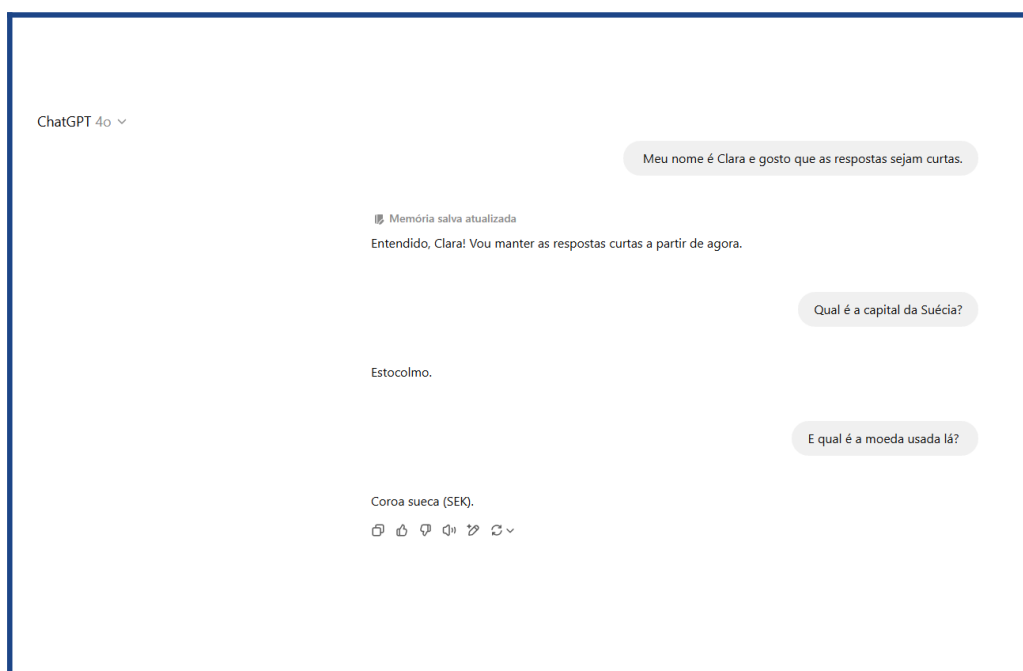
Diferente das estratégias baseadas apenas em prompting ou in-context learning, que dependem do conteúdo presente no prompt atual, o raciocínio com memória e contexto permite que o modelo mantenha continuidade entre sessões, ou mesmo ao longo de interações longas. Para isso, é necessário o suporte de uma infraestrutura externa de armazenamento e recuperação de memória, como bancos de dados vetoriais ou mecanismos de indexação.

Na prática, quando essa estratégia é aplicada, o modelo pode:

- Lembrar de preferências do usuário (ex: “Resuma sempre em até 5 linhas”).
- Retomar discussões anteriores (ex: “Continuando de onde paramos na última aula...”).
- Construir raciocínios cumulativos (ex: “Com base nas premissas anteriores, a nova conclusão é...”).

Essa capacidade contextual aprimorada aumenta significativamente a naturalidade das interações, aproximando a IA de um comportamento verdadeiramente humano em termos de memória, continuidade e adaptação ao interlocutor.

Para aplicar essa estratégia a ideia é criar um diálogo dividido em etapas, onde o modelo deverá lembrar de uma informação anterior e usá-la depois.





O modelo deve lembrar o nome do usuário e a preferência por respostas curtas, mesmo que essas informações não estejam presentes no último prompt.

2.3.7. Model Context Protocol

Model Context Protocol (MCP) é uma estratégia voltada à padronização e gerenciamento do contexto em modelos de linguagem, com o objetivo de organizar, reutilizar e controlar o acesso a informações contextuais durante múltiplas interações ou tarefas. Diferentemente das estratégias de prompting ou memória simples, o MCP propõe uma estrutura formal para lidar com o histórico, os estados e os dados relevantes usados em sessões de raciocínio ou geração de texto.

O MCP atua como um protocolo de comunicação e organização entre o modelo e seu ambiente, permitindo que ele lembre, acesse e reutilize dados anteriores de maneira coerente, mesmo em sessões ou tarefas diferentes. Isso é

especialmente importante em cenários em que o modelo precisa lidar com múltiplos documentos, etapas de raciocínio encadeadas, memória persistente ou colaboração entre agentes.

Em termos práticos, o MCP define como as informações são armazenadas, indexadas e injetadas no prompt, podendo incluir:

- Dados de interações passadas;
- Parâmetros de tarefas anteriores;
- Fatos aprendidos dinamicamente;
- Histórico de raciocínio.

O protocolo é especialmente útil em aplicações com multiagentes, longas conversas, sistemas de recomendação, tutores inteligentes ou workflows automatizados, onde é essencial manter consistência entre contextos e decisões.

Para funcionar corretamente, o MCP depende de uma infraestrutura externa de memória ou gerenciamento de estado, como bancos de dados vetoriais, sistemas de versionamento de contexto ou arquiteturas de agente (ex.: LangChain Agents, AutoGPT, ReAct). Ele pode ser visto como um sistema de memória estruturada + convenções de reutilização contextual.

Em resumo, o MCP não é uma técnica de raciocínio por si só, mas uma infraestrutura que suporta raciocínio complexo e contínuo em modelos de linguagem, garantindo que decisões anteriores e contexto relevante possam ser recuperados, atualizados e utilizados com precisão durante a geração de texto.

Como exemplo prático de como o Model Context Protocol (MCP) pode ser aplicado em um sistema com múltiplas etapas de raciocínio e persistência de contexto, usando um cenário simples de assistente virtual para planejamento de viagens.

Assistente de Viagem com Múltiplas Etapas: criando um agente de IA que ajuda um usuário a planejar uma viagem internacional. O assistente precisa:

- Coletar dados pessoais
- Selecionar o destino com base em preferências
- Reservar voos e hospedagem
- Gerar um roteiro com base nos interesses do usuário

Como o plano é feito em várias etapas e interações, o agente precisa manter e acessar o contexto previamente informado. Aqui entra o Model Context Protocol (MCP).

O MCP organiza e estrutura o contexto em blocos reutilizáveis (por exemplo: usuário, preferências, voos, hospedagem, roteiro). Cada bloco pode ser armazenado e recuperado conforme necessário.

```
# Instalar bibliotecas necessárias

!pip install langchain transformers

!pip install langchain huggingface_hub transformers
langchain-community

# Importar bibliotecas

from langchain.llms import HuggingFacePipeline
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM,
pipeline

from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferMemory

# Carregar modelo localmente

model_id = "google/flan-t5-small"

tokenizer = AutoTokenizer.from_pretrained(model_id)

model = AutoModelForSeq2SeqLM.from_pretrained(model_id)

pipe = pipeline("text2text-generation", model=model,
tokenizer=tokenizer, max_length=256)

llm = HuggingFacePipeline(pipeline=pipe)

# Criar cadeia de conversa com memória

memory = ConversationBufferMemory()

chain = ConversationChain(llm=llm, memory=memory, verbose=True)

# Etapa 1: Informações iniciais

chain.predict(input="Meu nome é Clara e gosto de clima frio e locais
históricos.")

# Etapa 2: Preferências de viagem

chain.predict(input="Quero viajar por 10 dias em setembro.")

# Etapa 3: Roteiro usando contexto

resposta = chain.predict(input="Sugira um roteiro de 3 dias para
essa viagem.")

print(resposta)
```

```
MCP_HuggingFacePipeline_Exemplo.ipynb ☆ 🔒
Arquivo Editar Ver Inserir Ambiente de execução Ferramentas Ajuda

andoss + Código + Texto

> Entering new ConversationChain chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If ti

Current conversation:
Human: Meu nome é Clara e gosto de clima frio e locais históricos.
AI:
Human: Quero viajar por 10 dias em setembro.
AI:

> Finished chain.
'I'm sorry, but I'm not sure what to do.'
```

```
[7]: 1 # Etapa 3: Roteiro usando contexto
2 resposta = chain.predict(input="Sugira um roteiro de 3 dias para essa viagem.")
3 print(resposta)
```

```
> Entering new ConversationChain chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If ti

Current conversation:
Human: Meu nome é Clara e gosto de clima frio e locais históricos.
AI:
Human: Quero viajar por 10 dias em setembro.
AI: I'm sorry, but I'm not sure what to do.
Human: Sugira um roteiro de 3 dias para essa viagem.
AI:
```

Essa estratégia exige conhecimento prévio de codificação e base de tokens, com essas limitações o exemplo não foi bem executado.

3. CONCLUSÕES

Este relatório apresentou uma análise das principais estratégias de raciocínio utilizadas em Modelos de Linguagem de Grande Escala (LLMs), com foco tanto em técnicas de treinamento supervisionado quanto em abordagens de inferência e suas especializações. O estudo foi conduzido a partir de fundamentos teóricos, testes práticos e aplicação de diferentes frameworks, com ênfase na compreensão do comportamento dos modelos diante de tarefas que exigem raciocínio estruturado, coerência contextual e justificativa lógica.

A aplicação do Fully Supervised Finetuning demonstrou como a adaptação de modelos a partir de dados rotulados pode gerar bons resultados mesmo em cenários com poucos exemplos, embora com limitações esperadas em termos de generalização. Já as estratégias de inferência, como Prompting e In-Context Learning, evidenciaram sua eficácia em tarefas rápidas, exigindo apenas uma boa formulação de entrada textual para orientar o modelo.

As especializações derivadas — como Rationale Engineering, Chain-of-Thought Prompting, Self-Consistency Sampling e Decomposição de Problemas — mostraram-se úteis para induzir raciocínios mais explícitos e confiáveis, especialmente em tarefas que exigem etapas intermediárias ou justificativas. Estratégias mais avançadas, como Tool-Augmented Reasoning, Memory and Contextual Reasoning e Model Context Protocol (MCP), revelaram-se fundamentais para a construção de agentes inteligentes capazes de operar com continuidade, memória, ferramentas externas e organização de contexto em múltiplas etapas.

A realização dos testes práticos, especialmente com o uso de modelos da Hugging Face e ambientes como o LangChain, possibilitou validar as estratégias em simulações reais, evidenciando tanto seu potencial quanto os desafios técnicos envolvidos na sua implementação. Ainda que algumas limitações tenham impedido a execução completa de certos experimentos, como no caso do Tool-Augmented Reasoning e MCP, o aprendizado adquirido e a estruturação das abordagens testadas reforçam a importância do tema.

Concluimos, portanto, que o domínio conceitual e prático dessas estratégias de raciocínio é essencial para a construção de agentes autônomos mais eficazes, explicáveis e interativos, e constitui um passo importante rumo ao desenvolvimento de sistemas de IA mais robustos, adaptativos e confiáveis no contexto acadêmico e profissional.

REFERÊNCIAS BIBLIOGRÁFICAS

AZEVEDO, Celso. **I2A2 – Agentes Autônomos: Reasoning**. I2A2 Academy, 2025.

BROWN et al. (2020) – **Language Models are Few-Shot Learners**. Disponível em: <https://arxiv.org/abs/2005.14165>.

HUANG, Y. et al. **Improving LLM Reasoning via Prompt Engineering**. arXiv:2212.10403, 2023. Disponível em: <https://arxiv.org/pdf/2212.10403.pdf>

SUN, Z. et al. **Reasoning in Large Language Models: A Survey**. arXiv:2312.11562, 2023. Disponível em:

QIAO, X. et al. **Reasoning in Language Models**. arXiv:2212.09597, 2023.

OpenAI. **GPT-4 Technical Report**. 2023. Disponível em:

OpenAI. **Tool-Use guide**. Disponível em:

<https://platform.openai.com/docs/guides/tool-use>

WANG et al. (2022) – **Self-Consistency Improves Chain of Thought Reasoning in Language Models**. Disponível em: <https://arxiv.org/abs/2203.11171>

Yao et al. (2022). **ReAct: Synergizing Reasoning and Acting in Language Models**. Disponível em: <https://arxiv.org/abs/2210.03629>