UNIVERSIDADE FEDERAL DE GOIÁS / INSTITUTO DE INFORMÁTICA

Assistente Virtual Baseado em Multiagentes Colaborativos

Sistema Iterativo com Cadeia de Pensamento (CoT) e Mixture of Agents (MoA)

Victor Emanuel da Silva Monteiro

UFG
UNIVERSIDADE FEDERAL DE GOIÁS

UNIVERSIDADE FEDERAL DE GOIÁS (UFG)
INSTITUTO DE INFORMÁTICA (INF)

VICTOR EMANUEL DA SILVA MONTEIRO

Assistente Virtual Baseado em Multiagentes Colaborativos (LLMs)

Sistema Iterativo com Cadeia de Pensamento (CoT) e Mixture of Agents (MoA)

Goiânia
2025

UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO PARA DISPONIBILIZAR VERSÕES ELETRÔNICAS DE TRABALHO DE CONCLUSÃO DE CURSO DE GRADUAÇÃO NO REPOSITÓRIO INSTITUCIONAL DA UFG

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio do Repositório Institucional (RI/UFG), regulamentado pela Resolução CEPEC no 1240/2014, sem ressarcimento dos direitos autorais, de acordo com a Lei no 9.610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou download, a título de divulgação da produção científica brasileira, a partir desta data.

O conteúdo dos Trabalhos de Conclusão dos Cursos de Graduação disponibilizado no RI/UFG é de responsabilidade exclusiva dos autores. Ao encaminhar(em) o produto final, o(s) autor(a)(es)(as) e o(a) orientador(a) firmam o compromisso de que o trabalho não contém nenhuma violação de quaisquer direitos autorais ou outro direito de terceiros.

1. Identificação do Trabalho de Conclusão de Curso de Graduação (TCCG)

Nome(s) completo(s) do(a)(s) autor(a)(es)(as): VICTOR EMANUEL DA SILVA MONTEIRO

Título do trabalho: Assistente Virtual Baseado em Multiagentes Colaborativos (LLMs)

Sistema Iterativo com Cadeia de Pensamento (CoT) e Mixture of Agents (MoA)

2. Informações de acesso ao documento (este campo deve ser preenchido pelo orientador) Concorda com a liberação total do documento [X] SIM [ ] NÃO[1]

[1] Neste caso o documento será embargado por até um ano a partir da data de defesa. Após esse período, a possível disponibilização ocorrerá apenas mediante: a) consulta ao(à)(s) autor(a)(es)(as) e ao(à) orientador(a); b) novo Termo de Ciência e de Autorização (TECA) assinado e inserido no arquivo do TCCG. O documento não será disponibilizado durante o período de embargo.

Casos de embargo:

- Solicitação de registro de patente;

- Submissão de artigo em revista científica;

- Publicação como capítulo de livro.

Obs.: Este termo deve ser assinado no SEI pelo orientador e pelo autor.

Documento assinado eletronicamente por **Victor Emanuel Da Silva Monteiro**, **Discente**, em 12/01/2025, às 14:37, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

Documento assinado eletronicamente por **Fernando Marques Federson**, **Professor do Magistério Superior**, em 15/01/2025, às 16:28, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

A autenticidade deste documento pode ser conferida no site https://sei.ufg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **5089807** e o código CRC **5A0F7BF0**.

Referência: Processo nº 23070.001601/2025-78 SEI nº 5089807

VICTOR EMANUEL DA SILVA MONTEIRO

Assistente Virtual Baseado em Multiagentes Colaborativos (LLMs)

Sistema Iterativo com Cadeia de Pensamento (CoT) e Mixture of Agents (MoA)

Relatório final de Trabalho de Conclusão de Curso, apresentado à Universidade Federal de Goiás, como parte das exigências para a obtenção do título de Bacharel em Inteligência Artificial.
Orientador: Prof. Dr. Fernando Marques Federson

Goiânia
2025

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFG.

MONTEIRO, VICTOR EMANUEL DA SILVA
Assistente Virtual Baseado em Multiagentes Colaborativos (LLMs) [manuscrito] : Sistema Iterativo com Cadeia de Pensamento (CoT) e Mixture of Agents (MoA) / VICTOR EMANUEL DA SILVA MONTEIRO. - 2025.
166 f.

Orientador: Prof. Dr. Fernando Marques Federson.
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Goiás, Instituto de Informática (INF), Inteligência Artificial, Goiânia, 2025.

1. inteligência artificial. 2. modelos de linguagem. 3. sistemas multiagentes. I. Federson, Fernando Marques , orient. II. Título.

CDU 004

VICTOR EMANUEL DA SILVA MONTEIRO

Assistente Virtual Baseado em Multiagentes Colaborativos (LLMs)

Sistema Iterativo com Cadeia de Pensamento (CoT) e Mixture of Agents (MoA)

Relatório final de Trabalho de Conclusão de Curso, apresentado à Universidade Federal de Goiás, como parte das exigências para a obtenção do título de Bacharel em Inteligência Artificial.

Data da Aprovação: 17 de dezembro de 2024.

Prof. Dr. Fernando Marques Federson
Orientador (INF-UFG)

Prof. Dr. Aldo André Díaz Salazar
Coordenador de TCC do BIA (INF-UFG)

Prof. Dr. Anderson da Silva Soares
Coordenador do BIA (INF-UFG)

Prof. Dr. Iwens Gervasio Sene Junior
(INF-UFG)

VICTOR EMANUEL DA SILVA MONTEIRO

Assistente Virtual Baseado em Multiagentes Colaborativos (LLMs)

Sistema Iterativo com Cadeia de Pensamento (CoT) e Mixture of Agents (MoA)

RESUMO

Este Relatório de Conclusão de Curso tem como objetivo reunir os resultados da minha jornada para me tornar um especialista em **Multiagentes Colaborativos (LLMs)**. Uma ilustração e sua narrativa descrevem os períodos de trabalho. Os Apêndices contêm os Termos de Aceite de Entrega e os resultados obtidos durante cada período de trabalho.

Palavras-chave: inteligência artificial, modelos grandes de linguagem, geração automática de datasets.

ABSTRACT

This Course Completion Report aims to bring together the results of my journey to become an expert in **Collaborative Multiagents (LLMs)**. An illustration and its narrative describe the work periods. The Appendices contain the Delivery Acceptance Terms and the results obtained during each work period.

Keywords: artificial intelligence, large language models, automatic dataset generation.

Goiânia

2025

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

Minha Jornada

Victor Emanuel da Silva Monteiro
Especialista em: Multiagentes Colaborativos (LLMs)

Instituto de Informática (INF) - UFG
Alameda Palmeiras, Quadra D, Câmpus Samambaia
Fone: (62) 3521-1181 / Fax: (62) 3521-1182

MINHA JORNADA

Nome: Victor Emanuel da Silva Monteiro

Especialidade: Multiagentes Colaborativos (LLM)

Objetivo deste documento

Durante o processo da disciplina Residência em IA[1], foram gerados diversos resultados na construção da minha especialização. A cada semana, um conjunto de resultados foi formalizado por um Termo de Aceite de Entrega e avaliado por uma banca, considerando o planejado e o realizado para o período. Este documento tem como objetivo descrever esses resultados obtidos, fazendo referência aos Termos de Aceite de Entrega e seus documentos associados.

Minha Jornada

Minha jornada começou na **Semana 1**, com um estudo aprofundado sobre os fundamentos e desafios dos Modelos de Linguagem de Grande Escala (LLMs). Durante essa etapa, explorei o *Survey*[2], que forneceu uma visão abrangente sobre a evolução histórica e os aspectos técnicos desses modelos. Entre as capacidades analisadas, destacou-se a habilidade dos LLMs em realizar aprendizado no contexto, permitindo que novos problemas sejam resolvidos com poucos exemplos, sem a necessidade de ajustes adicionais. No entanto, um desafio importante identificado foi a tendência dos LLMs de tentar fornecer respostas diretamente, sem realizar as etapas intermediárias necessárias para um raciocínio estruturado, que pode comprometer a precisão em tarefas complexas que exigem um processo passo a passo. Essa imersão inicial foi essencial para consolidar os alicerces teóricos que orientaram as etapas da minha residência, permitindo uma

[1] Dez semanas, entre setembro de 2024 e dezembro de 2024.

[2] ZHAO, Wayne Xin; ZHOU, Kun; LI, Junyi; TANG, Tianyi; WANG, Xiaolei; HOU, Yupeng; MIN, Yingqian; ZHANG, Beichen; ZHANG, Junjie; DONG, Zican; et al. **A Survey of Large Language Models.** 2023. Disponível em: https://arxiv.org/abs/2303.18223. Acesso em: 16 out. 2024.

compreensão sólida das capacidades e limitações dos LLMs. Os materiais relacionados a esta Semana podem ser encontrados no **Apêndice 1**.

Nas **Semanas 2 e 3**, meu foco esteve na exploração e análise de diferentes técnicas de fine-tuning[3] aplicadas a LLMs[4], com ênfase em métodos eficientes para cenários de recursos limitados. Estudei comparações entre técnicas como In-Context Learning e Few-shot Fine-Tuning, aprofundando-me em abordagens inovadoras como Mixture-of-Agents (MoA)[5] e Chain-of-Thought Prompting[6]. Uma das principais descobertas foi que o Fine-tuning frequentemente supera o In-Context Learning em modelos maiores, enquanto o Chain-of-Thought Prompting demonstrou melhorar o raciocínio em tarefas complexas. Além disso, investiguei alternativas eficientes, como Low-Rank Adaptation (LoRA) e Context Distillation, que oferecem um bom equilíbrio entre custo e generalização. Essa etapa foi essencial para compreender como adaptar LLMs a diferentes contextos e demandas, garantindo maior eficiência e precisão nos resultados. Os materiais relacionados a estas duas Semanas podem ser encontrados no **Apêndice 2**.

Na **Semana 4**, dediquei-me à investigação de frameworks que pudessem suportar as técnicas analisadas nas semanas anteriores, com o objetivo de integrar e potencializar suas aplicações. Entre os frameworks estudados, o LangChain destacou-se por facilitar a construção de cadeias de prompts complexos, viabilizando o uso estruturado do método Chain-of-Thought para gerenciar interações com modelos de linguagem. Além disso, explorei o Mirascope, um framework modular e confiável para desenvolvimento de agentes

[3] PARTHASARATHY, Venkatesh Balavadhani; ZAFAR, Athsham; KHAN, Afaq; SHAHID, Arsalan. **The Ultimate Guide to Fine-Tuning LLMs from Basics to Breakthroughs: An Exhaustive Review of Technologies, Research, Best Practices, Applied Research Challenges and Opportunities.** 2024. Disponível em: https://arxiv.org/abs/2408.13296. Acesso em: 18 set. 2024.

[4] ZHENG, Jiawei; HONG, Hanghai; WANG, Xiaoli; SU, Jingsong; LIANG, Yonggui; WU, Shikai. **Fine-tuning Large Language Models for Domain-specific Machine Translation.** 2024. Disponível em: https://arxiv.org/abs/2402.15061. Acesso em: 18 set. 2024.

[5] WANG, Junlin; WANG, Jue; ATHIWARATKUN, Ben; ZHANG, Ce; ZOU, James. **Mixture-of-Agents Enhances Large Language Model Capabilities.** 2024. Disponível em: https://arxiv.org/abs/2406.04692. Acesso em: 4 dez. 2024.

[6] WEI, Jason; WANG, Xuezhi; SCHUURMANS, Dale; BOSMA, Maarten; ICHTER, Brian; XIA, Fei; CHI, Ed; LE, Quoc; ZHOU, Denny. **Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.** 2023. Disponível em: https://arxiv.org/abs/2201.11903. Acesso em: 4 dez. 2024.

personalizados, oferecendo uma abordagem prática e de código aberto. Por fim, analisei o Tree of Thoughts, que expande o Chain-of-Thought para uma estrutura em árvore, permitindo maior exploração de direções de raciocínio em problemas complexos. Essa etapa foi essencial para compreender como utilizar ferramentas modernas na criação de soluções mais eficientes e adaptáveis. Os materiais relacionados a esta Semana podem ser encontrados no **Apêndice 3**.

Nas **Semanas 5, 6 e 7**, dediquei-me ao desenvolvimento de um Sistema Multiagente Iterativo e Dinâmico, projetado para resolver problemas complexos em etapas claras e adaptáveis. No sistema idealizado, o agente Propositor Alpha analisaria as solicitações, definiria objetivos e geraria hipóteses iniciais. Os Executores E1 e E2 explorariam soluções de forma independente, aplicando raciocínio estruturado para construir respostas detalhadas. Por fim, o agente Agregador Ômega sintetizaria as contribuições dos outros agentes, avaliaria os resultados e determinaria ajustes ou conclusões. Também seriam incorporadas ferramentas para consultas externas e novos exemplos de raciocínio, permitindo maior precisão e flexibilidade do sistema. Essa abordagem, moldada conforme a literatura, proporcionou uma possibilidade para lidar com problemas complexos. Os materiais relacionados a estas três Semanas podem ser encontrados no **Apêndice 4**.

Na **Semana 8**, foquei em aprimorar a autonomia do Sistema Multiagente ao implementar um mecanismo de escolha automática para determinar o modelo mais adequado a cada tipo de tarefa. Essa funcionalidade permite que o sistema analise a complexidade e os requisitos das solicitações, selecionando, de forma independente, o modelo ideal para atender às necessidades específicas. Além disso, o sistema foi projetado para decidir dinamicamente quantas iterações seriam necessárias para resolver cada problema, ajustando o número de ciclos de forma adaptativa. Essa evolução reforçou a capacidade do sistema de lidar com uma ampla variedade de solicitações de maneira eficiente e personalizada, aumentando sua flexibilidade e precisão na entrega de resultados. Os materiais relacionados a esta Semana podem ser encontrados no **Apêndice 5**.

Nas **Semanas 9 e 10**, foquei na avaliação e refinamento do Assistente Virtual Multiagente Colaborativo para garantir maior precisão e rastreabilidade nos resultados. Durante os testes, identifiquei que, em algumas etapas do processo de cadeia de raciocínio, havia perda de informações, o que comprometia a validação e o controle do processo. Para solucionar esse problema, implementei uma estrutura de registro detalhada, onde cada etapa documenta a ação realizada, os resultados obtidos, observações relevantes e o status de sucesso ou falha. Além disso, utilizei um conjunto de dados de avaliação[7] dedicado para para avaliar sistemas com cadeia de pensamento para testar a capacidade do sistema em lidar com raciocínios sequenciais, alcançando um desempenho superior a 90% de acertos. Essa fase também incluiu a criação de uma interface gráfica, promovendo maior acessibilidade e transparência para o usuário. Essa etapa final consolidou o sistema como uma solução eficiente e confiável para problemas complexos. Os materiais relacionados a estas duas Semanas podem ser encontrados no **Apêndice 6**.

Em função de tudo que vivi nesta jornada, gostaria de destacar que o principal aprendizado não foi apenas o desenvolvimento do sistema multiagente, mas a compreensão aprofundada do papel dos LLMs como ferramentas versáteis e poderosas. Aprendi a integrar diferentes LLMs para atuarem como agentes especializados, capazes de realizar ações no código de forma autônoma, explorando suas capacidades para analisar, planejar e executar tarefas de maneira colaborativa. A combinação de técnicas, como o uso de prompts estruturados, cadeias de raciocínio e ferramentas externas, mostrou-se essencial para aumentar a eficiência e a adaptabilidade do sistema. Mais do que isso, pude entender como a integração de LLMs com outros recursos permite construir soluções robustas e inovadoras, evidenciando o potencial transformador da IA em diversas aplicações. Essa jornada revelou que o verdadeiro valor está na interseção entre criatividade, técnica e o poder de combinar diferentes abordagens para resolver problemas complexos de forma inteligente.

[7] JACOVI, Alon et al. **A Chain-of-Thought Is as Strong as Its Weakest Link: A Benchmark for Verifiers of Reasoning Chains.** 2024. Disponível em: https://arxiv.org/abs/2402.00559. Acesso em: 4 dez. 2024.

APÊNDICE 1
Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 18 de set. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

Victor Emanuel da Silva Monteiro

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Essa semana foi dedicada ao estudo e análise do artigo "A Survey of Large Language Models", com o objetivo de adquirir uma compreensão abrangente sobre o estado atual dos modelos de linguagem de grande porte (LLMs). O estudo abordou desde os fundamentos históricos dos LLMs até técnicas avançadas de pré-treinamento, ajuste fino, aplicações e métodos de avaliação de desempenho.

Resumo do estudo realizado:

- *Resumo - A Survey of Large Language Models*

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

- Aprofundar em tópicos específicos identificados: Com base nos insights obtidos do artigo, pretendo explorar mais profundamente áreas como Técnicas de Fine Tuning e Aplicações de LLMs.
- Leitura e análise de artigos complementares: Procurar e estudar artigos adicionais que abordem os tópicos específicos selecionadas, bem como desafios e soluções atuais nessas áreas.

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

Resumo - A Survey of Large Language Models

O artigo "A Survey of Large Language Models" explora o contexto histórico, as principais descobertas e as técnicas fundamentais para entender e utilizar LLMs, com ênfase em quatro aspectos principais: pré-treinamento, ajuste fino, aplicação e avaliação de desempenho. Para facilitar estudos futuros, os principais tópicos abordados no documento foram organizados em seções, conforme listado abaixo:

1. Introdução:

- **A modelagem de linguagem:** A capacidade de compreender e gerar linguagem humana é um dos grandes desafios da inteligência artificial. A modelagem de linguagem, técnica que visa modelar a probabilidade de sequências de palavras, tem sido fundamental para o avanço da IA na linguagem, com aplicações em áreas como tradução automática, análise de sentimentos e geração de texto.
- **LLMs evoluíram de modelos de linguagem estatísticos para modelos neurais e, mais recentemente, para modelos pré-treinados em larga escala:** A modelagem de linguagem passou por diversas fases, começando com modelos estatísticos baseados em n-gramas, evoluindo para modelos neurais como RNNs e LSTMs, e culminando nos atuais modelos pré-treinados, como BERT e GPT, que são treinados em conjuntos de dados massivos de forma não supervisionada.
- **LLMs com grande numero de parâmetros exibem habilidades emergentes, como o aprendizado no contexto, que os diferenciam dos modelos menores:** Com o aumento do número de parâmetros e dados de treinamento, os LLMs têm demonstrado habilidades surpreendentes que não eram observadas em modelos menores, como a capacidade de aprender novas tarefas a partir de poucos exemplos, chamada de aprendizado no contexto (ICL), ou a habilidade de raciocínio passo a passo.
- **ChatGPT, baseado em LLMs, tem chamado a atenção pela sua capacidade de interagir em diálogos complexos com humanos:** O ChatGPT, chatbot desenvolvido pela OpenAI, é um exemplo notável da aplicação de LLMs. Sua capacidade de manter conversas complexas e realizar tarefas como tradução, geração de texto criativo e resposta a perguntas de forma natural e coerente tem gerado grande interesse e entusiasmo.
- **Apesar dos avanços, os princípios subjacentes aos LLMs e suas habilidades ainda não são totalmente compreendidos:** Apesar dos progressos impressionantes, ainda não há uma compreensão profunda de como os LLMs funcionam e como adquirem suas habilidades. A relação entre o tamanho do modelo,

dados de treinamento e capacidades, assim como o surgimento de habilidades emergentes, ainda são áreas de pesquisa ativa.

- **O artigo revisa os avanços em LLMs, abrangendo pré-treinamento, ajuste, utilização e avaliação, além de recursos e desafios:** O artigo oferece uma revisão completa da pesquisa em LLMs, abordando as etapas de desenvolvimento, técnicas de aplicação e métodos de avaliação, destacando os recursos disponíveis e os desafios a serem superados.

2. Visão Geral:

- **LLMs são modelos de linguagem Transformer com bilhões de parâmetros, treinados em dados massivos:** Os LLMs são baseados na arquitetura Transformer, que utiliza mecanismos de auto-atenção para processar sequências de dados. O grande diferencial dos LLMs é o seu número de parâmetros, que pode chegar a centenas de bilhões, e o volume de dados utilizado para o seu treinamento, que compreende textos de diversas fontes como livros, artigos, código e conversas.
- **Leis de escala: descrevem a relação entre o desempenho do modelo, tamanho do modelo, tamanho do conjunto de dados e poder computacional, fornecendo diretrizes para treinamento otimizado de LLMs:** Estudos empíricos têm demonstrado que existe uma relação entre o desempenho dos LLMs e fatores como o tamanho do modelo, volume de dados e poder computacional. Essa relação, formulada como leis de escala, permite prever o desempenho de modelos maiores a partir do comportamento de modelos menores, otimizando o processo de treinamento e desenvolvimento de LLMs.
- **Habilidades emergentes: como o aprendizado no contexto, o acompanhamento de instruções e o raciocínio passo a passo, surgem em LLMs de grande escala, diferenciando-os dos modelos menores:** As habilidades emergentes são capacidades que surgem em LLMs quando estes atingem um determinado tamanho e volume de dados. O aprendizado no contexto, a habilidade de seguir instruções em linguagem natural e o raciocínio passo a passo são exemplos de habilidades que não são observadas em modelos menores, tornando os LLMs ferramentas poderosas para uma gama de aplicações.
- **Técnicas-chave: incluem escalonamento, treinamento distribuído, elicitação de habilidades, ajuste de alinhamento e manipulação de ferramentas:** O desenvolvimento e a utilização de LLMs dependem de um conjunto de técnicas-chave. O escalonamento do modelo e dos dados, o treinamento distribuído em clusters de GPUs, a elicitação de habilidades através de prompts específicos, o ajuste para alinhamento com valores humanos e a manipulação de ferramentas externas são exemplos de técnicas que contribuem para a capacidade dos LLMs.
- **Evolução dos modelos GPT:** GPT-1 estabeleceu a arquitetura básica. GPT-2 introduziu o aprendizado multitarefa não supervisionado e o escalonamento do

modelo. GPT-3 apresentou o aprendizado no contexto e explorou os limites do escalamento. O treinamento com dados de código aprimorou a capacidade de raciocínio, enquanto o alinhamento humano, por meio do aprendizado por reforço com feedback humano (RLHF), melhorou o alinhamento com valores humanos. ChatGPT e GPT-4 representam marcos importantes, com o último expandindo a entrada de texto para sinais multimodais: A série de modelos GPT desenvolvidos pela OpenAI representa um marco na pesquisa em LLMs. Desde o GPT-1, que estabeleceu a arquitetura básica, até o GPT-4, que suporta entradas multimodais, cada versão introduziu avanços significativos. O GPT-2 explorou o aprendizado multitarefa e o escalonamento, o GPT-3 introduziu o aprendizado no contexto, e o treinamento com dados de código aprimorou as capacidades de raciocínio. O ChatGPT, especializado em conversação, e o GPT-4, com suas capacidades multimodais, demonstram o potencial dos LLMs como ferramentas versáteis.

3. Recursos de LLMs:

- **Pontos de verificação de modelo (ou APIs): modelos pré-treinados com dezenas de bilhões e centenas de bilhões de parâmetros estão disponíveis publicamente, como GPT-NeoX-20B, LLaMA e OPT, fornecendo pontos de partida para pesquisa e desenvolvimento:** Para facilitar a pesquisa e desenvolvimento de LLMs, diversos modelos pré-treinados com diferentes números de parâmetros são disponibilizados publicamente, permitindo que pesquisadores e desenvolvedores explorem as capacidades dos LLMs sem a necessidade de realizar o treinamento do zero, que requer alto poder computacional e grande volume de dados.
- **Corpora: conjuntos de dados massivos, incluindo livros, páginas da Web, conversas de mídia social e código, são usados para pré-treinamento de LLMs (e.g., C4, OpenWebText, the Pile):** Para treinar LLMs, são utilizados conjuntos de dados massivos que englobam uma variedade de textos, como livros, páginas da web, conversas de mídia social e código. Esses conjuntos de dados fornecem a base para que os LLMs aprendam padrões de linguagem, conhecimento factual e até mesmo habilidades de raciocínio.
- **Bibliotecas: ferramentas de software, como Transformers, DeepSpeed e Megatron-LM, facilitam o desenvolvimento e treinamento de LLMs:** Diversas bibliotecas de software foram desenvolvidas para facilitar o processo de desenvolvimento e treinamento de LLMs. Bibliotecas como Transformers, DeepSpeed e Megatron-LM fornecem implementações da arquitetura Transformer, algoritmos de otimização para treinamento distribuído e ferramentas para gerenciar o grande número de parâmetros dos LLMs.

4. Pré-treinamento:

- **Coleta e preparação de dados: LLMs exigem dados de alta qualidade de várias fontes. O pré-processamento de dados, incluindo filtragem de qualidade, deduplicação, redução de privacidade e tokenização, é crucial para um corpus de treinamento eficaz:** A qualidade dos dados utilizados para pré-treinar LLMs é fundamental para o seu desempenho. É crucial coletar dados de diversas fontes e aplicar técnicas de pré-processamento para remover informações ruidosas, duplicadas ou que violem a privacidade, além de realizar a tokenização, que divide o texto em unidades menores para serem processadas pelo modelo.
- **Arquitetura:** a arquitetura Transformer é amplamente utilizada, com variantes como codificador-decodificador, decodificador causal e decodificador de prefixo. Configurações detalhadas, como métodos de normalização, incorporações de posição e funções de ativação, influenciam o desempenho do modelo. A arquitetura Transformer, com seus mecanismos de auto-atenção, tem se mostrado altamente eficaz para modelar sequências de dados, tornando-se a base para a maioria dos LLMs. Diversas variantes da arquitetura Transformer, como codificador-decodificador, decodificador causal e decodificador de prefixo, são utilizadas para diferentes tarefas, e as configurações específicas de cada arquitetura, como os métodos de normalização, incorporações de posição e funções de ativação, têm grande impacto no desempenho final do modelo.
- **Tarefas de pré-treinamento: a modelagem de linguagem e a autocodificação de redução de ruído são tarefas comuns de pré-treinamento, com variantes como Mixture-of-Denoisers (MoD) para unificar objetivos:** A escolha da tarefa de pré-treinamento é crucial para determinar as capacidades do LLM. Tarefas como a modelagem de linguagem, que visa prever a próxima palavra em uma sequência, e a autocodificação de redução de ruído, que treina o modelo a reconstruir textos corrompidos, são amplamente utilizadas para dotar os LLMs de habilidades de compreensão e geração de texto. Variantes como o Mixture-of-Denoisers (MoD) combinam diferentes objetivos de pré-treinamento para alcançar um aprendizado mais robusto.
- **Modelagem de contexto longo: técnicas como escalonamento de incorporações de posição, janelas de contexto paralelas, janelas de contexto em forma de Λ e memória externa estendem a capacidade dos LLMs de lidar com sequências longas:** Um dos desafios na modelagem de linguagem é lidar com sequências longas de texto. Diversas técnicas têm sido propostas para estender a janela de contexto dos LLMs, permitindo que eles processem textos mais extensos. Entre elas, destacam-se o escalonamento de incorporações de posição, que permite codificar posições mais distantes na sequência, janelas de contexto paralelas, que dividem o texto em segmentos menores para serem processados em paralelo, janelas de contexto em forma de Λ, que priorizam informações relevantes no início e no final do texto, e a utilização de memória externa para armazenar informações do contexto que excedem o limite da janela.

- **Estratégia de decodificação: a pesquisa gulosa e as técnicas baseadas em amostragem, como amostragem de temperatura, amostragem top-k e amostragem top-p, são usadas para gerar texto a partir de LLMs:** A decodificação é o processo pelo qual os LLMs geram texto a partir de uma sequência de entrada. A estratégia de decodificação influencia a qualidade e a diversidade do texto gerado. A pesquisa gulosa escolhe a palavra com maior probabilidade em cada passo, enquanto as técnicas baseadas em amostragem, como a amostragem de temperatura, a amostragem top-k e a amostragem top-p, introduzem aleatoriedade no processo de geração, permitindo a criação de textos mais criativos e menos repetitivos.
- **Treinamento de modelo: configurações de otimização, como treinamento em lote, taxa de aprendizado e otimizador (e.g., Adam e Adafactor), são importantes. Técnicas escalonáveis de treinamento, como paralelismo 3D, ZeRO e treinamento de precisão mista, são essenciais para modelos de grande escala:** O treinamento de LLMs é um processo computacionalmente intensivo que requer configurações de otimização cuidadosamente ajustadas. Técnicas como treinamento em lote, que processa múltiplos exemplos simultaneamente, taxa de aprendizado, que controla a velocidade do aprendizado, e a escolha do otimizador, como Adam e Adafactor, influenciam a estabilidade e a velocidade do treinamento. Para modelos de grande escala, técnicas como o paralelismo 3D, que distribui o modelo e os dados em múltiplas GPUs, o ZeRO, que otimiza o uso de memória, e o treinamento de precisão mista, que utiliza diferentes tipos de dados numéricos para acelerar o treinamento, são essenciais para viabilizar o processo.

5. Adaptação de LLMs:

- **Ajuste de instrução: envolve o ajuste fino de LLMs em instâncias formatadas em linguagem natural, geralmente coletadas de conjuntos de dados de tarefas de PNL ou conversas diárias. Estratégias como balanceamento de dados, combinação com pré-treinamento e ajuste multiestágio aprimoram o desempenho:** O ajuste de instrução é uma técnica que visa adaptar os LLMs para realizar tarefas específicas, ajustando seus parâmetros a partir de exemplos formatados em linguagem natural. Esses exemplos podem ser coletados de conjuntos de dados de tarefas de PNL, como tradução e resumo de texto, ou de conversas diárias. Para garantir a eficácia do ajuste, é fundamental balancear a distribuição dos dados de treinamento, combinar o ajuste com o pré-treinamento original do modelo e, em alguns casos, realizar o ajuste em múltiplas etapas para um aprendizado gradual e eficaz.
- **Ajuste de alinhamento: visa alinhar LLMs com valores humanos usando *Reinforcement Learning from Human Feedback* (RLHF). Envolve coletar feedback humano, treinar um modelo de recompensa e ajustar o LLM com base**

nas recompensas do modelo: O ajuste de alinhamento é uma técnica crucial para garantir que os LLMs gerem textos que estejam em conformidade com valores e normas sociais. O aprendizado por reforço com feedback humano (RLHF) é uma técnica que permite incorporar a avaliação humana no processo de treinamento. Através do RLHF, são coletadas avaliações humanas sobre a qualidade das respostas geradas pelos LLMs, que são utilizadas para treinar um modelo de recompensa. O LLM é então ajustado para maximizar as recompensas recebidas do modelo de recompensa, aprendendo a gerar textos que sejam mais bem avaliados pelos humanos.

- **Adaptação de modelo com eficiência de parâmetros: métodos como ajuste de adaptador, ajuste de prefixo, ajuste de prompt e LoRA reduzem o número de parâmetros ajustáveis, tornando o ajuste mais eficiente:** Para reduzir o custo computacional do ajuste de LLMs, diversos métodos têm sido propostos para realizar o ajuste de forma eficiente em termos de parâmetros. Técnicas como o ajuste de adaptador, que adiciona módulos pequenos e treináveis ao modelo, o ajuste de prefixo, que adiciona prefixos treináveis às camadas do modelo, o ajuste de prompt, que utiliza prompts treináveis para guiar a geração de texto, e o LoRA, que realiza o ajuste através de matrizes de baixa dimensão, permitem ajustar os LLMs com um número menor de parâmetros, tornando o processo mais rápido e eficiente.
- **Adaptação de modelo com eficiência de memória: a quantização do modelo, usando técnicas como decomposição de precisão mista, quantização de granularidade fina e quantização em camadas, comprime LLMs para implantação em ambientes com recursos limitados:** Para reduzir o uso de memória e acelerar a inferência dos LLMs, a quantização do modelo é uma técnica importante. A quantização consiste em reduzir a precisão dos parâmetros do modelo, utilizando um número menor de bits para representá-los. Técnicas como a decomposição de precisão mista, que utiliza diferentes níveis de precisão para diferentes partes do modelo, a quantização de granularidade fina, que utiliza diferentes quantizadores para diferentes partes do modelo, e a quantização em camadas, que realiza a quantização em cada camada do modelo, permitem comprimir os LLMs sem comprometer significativamente o seu desempenho.

6. Utilização:

- **Prompting: envolve projetar prompts eficazes para solicitar LLMs a executar tarefas. Princípios de design incluem expressar claramente o objetivo da tarefa, decompor tarefas complexas em subtarefas, fornecer demonstrações de poucos disparos e utilizar formatos amigáveis ao modelo:** A técnica de prompting é fundamental para instruir os LLMs a realizar tarefas específicas. Um prompt bem projetado deve expressar o objetivo da tarefa de forma clara e concisa, decompor tarefas complexas em subtarefas mais simples, fornecer exemplos

demonstrativos para ilustrar a tarefa e utilizar formatos que sejam facilmente compreendidos pelo modelo.

- **Aprendizagem no contexto (ICL): permite que LLMs realizem novas tarefas com base em uma descrição da tarefa e alguns exemplos, sem ajuste explícito de parâmetros. O projeto de demonstração, incluindo seleção, formato e ordem, é crucial para um ICL eficaz:** O aprendizado no contexto (ICL) é uma capacidade notável dos LLMs que permite que eles aprendam novas tarefas a partir de poucos exemplos, sem a necessidade de realizar o ajuste de parâmetros. Para que o ICL seja eficaz, o design da demonstração é crucial. A seleção dos exemplos demonstrativos, o formato em que são apresentados ao modelo e a ordem em que são fornecidos podem influenciar significativamente o desempenho do modelo na nova tarefa.
- **Prompting do tipo cadeia de pensamento (CoT): aprimora o ICL incorporando etapas intermediárias de raciocínio nos prompts, melhorando o desempenho em tarefas de raciocínio complexo:** O prompting do tipo cadeia de pensamento (CoT) é uma técnica que expande o aprendizado no contexto (ICL) ao incluir etapas intermediárias de raciocínio nos prompts. Essa técnica permite que os LLMs realizem raciocínio passo a passo, melhorando o seu desempenho em tarefas que exigem raciocínio complexo, como resolução de problemas matemáticos ou compreensão de textos que exigem inferência lógica.
- **Planejamento: envolve decompor tarefas complexas em subtarefas e gerar um plano de ações para realizar a tarefa, utilizando LLMs como planejadores de tarefas e alavancando ferramentas externas para execução:** O planejamento é uma técnica que permite que os LLMs resolvam tarefas complexas de forma autônoma. LLMs podem ser utilizados como planejadores de tarefas, decompondo tarefas complexas em subtarefas menores e gerando um plano de ações para alcançar o objetivo final. A execução do plano pode ser realizada pelo próprio LLM ou por ferramentas externas, como um interpretador de código ou um robô.

7. Capacidade e Avaliação:

- **Habilidade básica: abrange geração de linguagem (modelagem de linguagem, geração de texto condicional e síntese de código), utilização de conhecimento (QA de livro fechado, QA de livro aberto e preenchimento de conhecimento) e raciocínio complexo (raciocínio de conhecimento, raciocínio simbólico e raciocínio matemático):** A avaliação das habilidades básicas dos LLMs é crucial para compreender a sua capacidade de lidar com tarefas fundamentais em processamento de linguagem natural. Essas habilidades incluem a geração de linguagem, que abrange tarefas como modelagem de linguagem, geração de texto condicional e síntese de código, a utilização de conhecimento, que avalia a capacidade do modelo de responder a perguntas com base em conhecimento factual

adquirido durante o pré-treinamento, e o raciocínio complexo, que testa a capacidade do modelo de realizar inferências lógicas, manipular símbolos e resolver problemas matemáticos.

- **Habilidade avançada: inclui alinhamento humano (ajuda, honestidade e segurança), interação com o ambiente externo (e.g., ambientes de IA incorporados) e manipulação de ferramentas (e.g., mecanismos de plug-in):** Além das habilidades básicas, os LLMs também são avaliados em suas habilidades avançadas, que refletem a sua capacidade de interagir com o mundo real de forma mais sofisticada. Essas habilidades incluem o alinhamento humano, que avalia a capacidade do modelo de gerar textos que estejam em conformidade com valores e normas sociais, a interação com o ambiente externo, que testa a capacidade do modelo de operar em ambientes simulados ou reais, e a manipulação de ferramentas, que avalia a capacidade do modelo de utilizar ferramentas externas para realizar tarefas complexas.
- **Benchmarks: conjuntos de dados padronizados, como MMLU, BIG-bench e HELM, são usados para avaliar as capacidades de LLMs em várias tarefas e domínios:** Para avaliar de forma sistemática e padronizada as capacidades dos LLMs, diversos benchmarks foram desenvolvidos. Esses benchmarks consistem em conjuntos de dados padronizados que abrangem diferentes tarefas e domínios, como compreensão de texto, raciocínio lógico e resolução de problemas.
- **Abordagens de avaliação: incluem avaliação baseada em benchmark, avaliação humana e avaliação baseada em modelo, cada uma com seus próprios benefícios e limitações:** Diversas abordagens são utilizadas para avaliar os LLMs. A avaliação baseada em benchmark utiliza conjuntos de dados padronizados para medir o desempenho do modelo em diferentes tarefas. A avaliação humana utiliza a avaliação de especialistas ou crowdsourcing para avaliar a qualidade do texto gerado pelo modelo. A avaliação baseada em modelo utiliza outros LLMs para avaliar a qualidade do texto gerado. Cada abordagem apresenta seus próprios benefícios e limitações, sendo a escolha da abordagem mais adequada dependente do objetivo da avaliação.
- **Análise empírica: a avaliação em tarefas e benchmarks representativos destaca os pontos fortes e fracos dos LLMs em diferentes habilidades, como geração de linguagem, utilização de conhecimento e raciocínio complexo:** A análise empírica dos resultados de avaliação em diferentes tarefas e benchmarks fornece insights sobre os pontos fortes e fracos dos LLMs. Essa análise permite identificar as áreas em que os LLMs se destacam, como a geração de texto criativo ou a tradução automática, e as áreas em que ainda precisam ser aprimorados, como o raciocínio complexo ou a resolução de problemas matemáticos.

8. Aplicações:

- **LLMs para tarefas clássicas de PNL: LLMs têm impacto em tarefas de PNL como análise de sentimentos, reconhecimento de entidades nomeadas e extração de informações, embora ainda enfrentem desafios em áreas como tarefas de recursos baixos:** Os LLMs têm demonstrado grande potencial para revolucionar a forma como realizamos tarefas clássicas de PNL. Sua capacidade de generalização e aprendizado no contexto os torna ferramentas poderosas para tarefas como análise de sentimentos, reconhecimento de entidades nomeadas e extração de informações. No entanto, ainda enfrentam desafios em cenários com poucos dados disponíveis, como a tradução de línguas menos comuns.
- **LLMs para recuperação de informações: LLMs são usados como modelos de IR ou para aprimorar modelos de IR existentes, com foco na classificação de texto e aumento de dados:** A área de recuperação de informações (IR) também se beneficia das capacidades dos LLMs. Os LLMs podem ser utilizados como modelos de IR, substituindo ou complementando os modelos tradicionais, ou podem ser utilizados para aprimorar os modelos existentes, gerando dados sintéticos para treinamento ou realizando a classificação de textos para melhorar a qualidade dos resultados.
- **LLMs para sistemas de recomendação: LLMs atuam como modelos de recomendação, aprimoram os modelos de recomendação tradicionais e atuam como simuladores de recomendação para modelar o comportamento do usuário:** Os LLMs também encontram aplicações em sistemas de recomendação. Eles podem atuar como modelos de recomendação, substituindo ou complementando os modelos tradicionais, podem ser utilizados para aprimorar os modelos existentes através da geração de dados sintéticos ou da extração de características semânticas, ou podem servir como simuladores de recomendação, modelando o comportamento de usuários para testar e melhorar os sistemas de recomendação.
- **Modelo de linguagem multimodal grande: integra informações de várias modalidades, como texto e imagem, para tarefas como resposta a perguntas visuais. O pré-treinamento de alinhamento visão-linguagem e o ajuste fino de instrução visual são etapas de treinamento essenciais:** Os modelos de linguagem multimodal grande (MLLMs) expandem a capacidade dos LLMs para processar diferentes tipos de dados, como imagens, áudio e vídeo. Através do pré-treinamento de alinhamento visão-linguagem e do ajuste fino de instrução visual, os MLLMs podem realizar tarefas como responder a perguntas sobre imagens, descrever cenas e gerar legendas para vídeos.
- **LLM aprimorado por KG: a integração de gráficos de conhecimento (KGs) em LLMs, por meio de métodos aumentados por recuperação ou sinergia, visa melhorar o desempenho em tarefas intensivas em conhecimento:** Para aprimorar a capacidade dos LLMs em lidar com tarefas que exigem conhecimento factual preciso, a integração de gráficos de conhecimento (KGs) tem se mostrado

uma estratégia promissora. LLMs aumentados por recuperação utilizam KGs para fornecer informações adicionais aos prompts, enquanto LLMs aumentados por sinergia interagem com KGs em múltiplas etapas para realizar raciocínio e inferência complexos.

- **Agente baseado em LLM: LLMs atuam como a unidade central de computação de agentes, equipados com memória, planejamento e execução, para realizar tarefas de forma autônoma:** Os LLMs podem ser utilizados como a base para o desenvolvimento de agentes autônomos. Equipados com memória, planejamento e execução, os agentes baseados em LLMs podem interagir com o ambiente, tomar decisões e realizar ações de forma autônoma para alcançar objetivos específicos.
- **LLM para avaliação: LLMs são usados como avaliadores automáticos de texto gerado, substituindo ou auxiliando a avaliação humana, com formatos de avaliação baseados em pontuação ou linguagem:** A avaliação da qualidade do texto gerado por LLMs é uma tarefa importante. Os LLMs podem ser utilizados como avaliadores automáticos, substituindo ou complementando a avaliação humana. Eles podem gerar pontuações para a qualidade do texto, realizar comparações entre diferentes textos gerados ou fornecer feedback em linguagem natural para auxiliar na melhoria do processo de geração.
- **Domínios específicos: LLMs têm aplicações promissoras em áreas como saúde, educação, direito, finanças e assistência à pesquisa científica:** As capacidades dos LLMs têm impacto em diversos domínios específicos. Na saúde, eles podem auxiliar no diagnóstico médico e na geração de relatórios. Na educação, podem auxiliar na criação de materiais didáticos personalizados. No direito, podem auxiliar na análise de documentos legais e na pesquisa jurídica. Na área financeira, podem auxiliar na análise de mercado e na gestão de investimentos. Na pesquisa científica, podem auxiliar na revisão da literatura e na geração de hipóteses.

9. Conclusão e Direções Futuras:

- **Fundamentos e princípios: compreender os princípios subjacentes às habilidades de LLMs, o papel do escalonamento e os mecanismos de generalização são direções de pesquisa essenciais:** A pesquisa em LLMs ainda está em seus estágios iniciais, e muitas questões fundamentais precisam ser respondidas. Compreender como os LLMs adquirem suas habilidades, como o escalonamento do modelo e dos dados influencia o seu desempenho e como eles generalizam para novas tarefas são áreas de pesquisa essenciais para avançar o campo.
- **Arquitetura do modelo: explorar arquiteturas alternativas para reduzir custos de treinamento e melhorar a eficiência de inferência é crucial:** A arquitetura Transformer tem sido a base para a maioria dos LLMs, mas o seu custo computacional para treinamento e inferência é alto. Explorar novas arquiteturas que

sejam mais eficientes em termos de tempo e memória é crucial para tornar os LLMs mais acessíveis e escalonáveis.

- **Treinamento de modelo: desenvolver infraestruturas e procedimentos centrados em dados, investigar receitas de treinamento e métodos de ajuste fino eficientes e eficazes são importantes:** O processo de treinamento de LLMs é complexo e sensível à qualidade dos dados e às configurações de otimização. Desenvolver infraestruturas e procedimentos mais robustos e eficientes para coletar, pré-processar e agendar dados para treinamento, assim como investigar novas estratégias de treinamento e ajuste fino, é fundamental para otimizar o desenvolvimento de LLMs.
- **Utilização do modelo: compreender os princípios de prompting, aprimorar técnicas de prompting avançadas e reduzir os custos de inferência são áreas de foco:** A forma como utilizamos os LLMs através de prompts é crucial para o seu desempenho em tarefas específicas. Compreender os princípios de prompting, aprimorar as técnicas de prompting avançadas, como ICL e CoT, e reduzir os custos de inferência para tornar os LLMs mais práticos para aplicações reais são áreas de pesquisa importantes.
- **Segurança e alinhamento: melhorar os métodos de alinhamento, explorar abordagens de anotação eficientes e desenvolver algoritmos de otimização simplificados são cruciais para mitigar os riscos associados a LLMs:** Os LLMs apresentam desafios em termos de segurança e alinhamento com valores humanos. Aprimorar as técnicas de alinhamento, como o RLHF, explorar métodos de anotação mais eficientes e desenvolver algoritmos de otimização mais robustos são cruciais para garantir que os LLMs sejam utilizados de forma ética e responsável.
- **Aplicação e ecossistema: impulsionado pelo rápido progresso em LLMs, um ecossistema de aplicações baseadas em LLM está surgindo, impactando vários domínios e potencialmente levando a sistemas de IA mais inteligentes e uma exploração contínua da AGI:** O rápido progresso em LLMs tem impulsionado o desenvolvimento de um ecossistema de aplicações que utilizam essas tecnologias para realizar tarefas complexas em diversos domínios. Esse ecossistema promete revolucionar a forma como interagimos com a tecnologia, impulsionando o desenvolvimento de sistemas de IA mais inteligentes e abrindo caminho para a exploração da inteligência artificial geral (AGI).

APÊNDICE 2
Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 18 de set. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

Victor Emanuel da Silva Monteiro

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Nesta semana, o estudo focou na exploração de 15 principais técnicas de fine-tuning, identificando aquelas mais adequadas para diferentes contextos de adaptação de modelos. Além de compreender o funcionamento de cada método, foram analisadas suas vantagens e limitações, assim como as aplicações mais viáveis e comuns para cada técnica.

Papers estudados:

- The Ultimate Guide to Fine-Tuning LLMs from Basics to Breakthroughs: An Exhaustive Review of Technologies, Research, Best Practices, Applied Research Challenges and Opportunities
- Fine-tuning Large Language Models for Domain-specific Machine Translation

Estudo completo das técnicas: Técnicas de Fine-Tuning

Técnicas utilizadas de Fine-Tuning em LLMs:

- **Supervisionado**: Utilizada quando se dispõe de um conjunto robusto de dados rotulados, como na classificação de sentimentos e tradução de idiomas.
- **Adaptação de Domínio**: Usada para adaptar o modelo a contextos específicos utilizando grandes corpora de texto não-rotulados de um determinado domínio
- **Instrução**: Aplicada quando se deseja que o modelo siga comandos complexos e variados, como na criação de assistentes virtuais personalizados.
- **MoA (Mixture of Agents)**: Ideal para tarefas que exigem colaboração e especialização de diferentes modelos, como geração de respostas complexas e interativas.
- **PEFT (Parameter-efficient fine-tuning)**: Escolhida para otimizar modelos de grande porte com eficiência, especialmente quando há restrição de recursos computacionais e memória.

Principais desafios de Fine-Tuning em LLMs:

- **Aprendizado Contínuo (Continuous Learning)**: Adaptar o modelo continuamente sem perder o

conhecimento adquirido anteriormente.

- **Tamanho e Heterogeneidade de Dados**: Gerenciar dados com características variadas, com qualidade, garantindo que o modelo consiga lidar com diferentes tipos de contextos.
- **Recursos Computacionais**: Realizar o ajuste fino em modelos de grande porte com alta demanda por memória e processamento, especialmente com recursos limitados.
- **Generalização**: Garantir que o modelo, após o fine-tuning, consiga generalizar bem para novos dados, evitando o risco de superespecialização no conjunto de dados de treinamento.

Planejamento: **[descrever o que pretende fazer para realizar a próxima ENTREGA]**

- Comparar o desempenho entre as técnicas de fine-tuning,
- Entender quais técnicas são mais eficientes
- Avaliar comparação de custo de fine-tuning

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

LEONARDO ALVES: Go!

Técnicas de Fine-Tuning

Supervisão (Supervised): O ajuste fino supervisionado usa dados rotulados para treinar o modelo a realizar uma tarefa específica. Por exemplo, para ajustar finamente um modelo para classificação de sentimentos, você forneceria uma base de dados de textos rotulados como positivos, negativos ou neutros. O modelo aprende a partir dos dados rotulados para prever corretamente o sentimento de novos textos.

- Vantagens: Alta precisão em tarefas específicas, bom desempenho em domínios conhecidos.
- Desvantagens: Requer dados rotulados, pode ser caro e demorado para rotular dados.
- Aplicações: Classificação de sentimentos, tradução de idiomas, resposta de perguntas.

Não-supervisionado (Unsupervised): O ajuste fino não-supervisionado usa dados não rotulados para treinar o modelo a melhorar sua compreensão da linguagem. Por exemplo, você pode fornecer um modelo de linguagem com um grande corpo de texto não-rotulado em um domínio específico, como artigos de medicina, e ele aprenderá a usar a linguagem desse domínio.

- Vantagens: Requer menos dados rotulados, pode ser usado para domínios desconhecidos.
- Desvantagens: Pode ser menos preciso do que o ajuste fino supervisionado, pode ser difícil avaliar o desempenho.
- Aplicações: Compreensão de linguagem, preenchimento de texto, geração de texto.

Instrução (Instruction): O ajuste fino de instruções visa treinar o modelo a seguir instruções de linguagem natural. Por exemplo, você pode dar instruções para o modelo gerar um resumo de um texto ou traduzir um texto para outro idioma.

- Vantagens: Permite que o modelo execute tarefas complexas, pode ser usado para criar assistentes personalizados.
- Desvantagens: Requer instruções de alta qualidade, pode ser difícil avaliar o desempenho.
- Aplicações: Geração de texto, assistentes, chatbots.

Ajuste Fino de Baixo Posto (PEFT - Parameter-Efficient Fine-Tuning): As técnicas PEFT visam reduzir o número de parâmetros que precisam ser ajustados durante o ajuste fino. Isso torna o processo de ajuste fino mais rápido e eficiente, especialmente para modelos de linguagem de grande porte (LLMs), que costumam ter bilhões de parâmetros.

- Vantagens: Economia de recursos computacionais, menores requisitos de memória, menor risco de esquecimento catastrófico.
- Desvantagens: Pode não ser tão preciso quanto o ajuste fino completo, pode não ser adequado para todos os cenários.
- Aplicações: Ajuste fino de LLMs para tarefas específicas, transferência de conhecimento.

LoRA (Low-Rank Adaptation): LoRA é uma técnica PEFT que aproxima a atualização dos pesos do modelo usando o produto de duas matrizes de baixo posto. Isso reduz significativamente o número de parâmetros que precisam ser ajustados, tornando o processo de ajuste fino mais eficiente.

- Vantagens: Altamente eficiente em termos de recursos, mantém o desempenho do modelo pré-treinado.
- Desvantagens: Pode não ser tão preciso quanto o ajuste fino completo, requer ajuste fino de hiperparâmetros meticuloso.
- Aplicações: Ajuste fino de LLMs para tarefas específicas, transferência de conhecimento.

QLoRA (Quantized LoRA): QLoRA é uma extensão do LoRA que usa quantização para reduzir ainda mais o uso de memória. Os parâmetros do modelo são quantizados para uma precisão de 4 bits, o que reduz significativamente o uso de memória, tornando o ajuste fino possível em hardware mais acessível.

- Vantagens: Altamente eficiente em termos de memória, mantém o desempenho do modelo pré-treinado.
- Desvantagens: Pode não ser tão preciso quanto o ajuste fino completo, requer ajuste fino de hiperparâmetros meticuloso.
- Aplicações: Ajuste fino de LLMs para tarefas específicas, transferência de conhecimento, especialmente em cenários com recursos limitados.

DoRA (Weight-Decomposed Low-Rank Adaptation): DoRA é uma técnica de ajuste fino de baixo posto que decompõe os pesos do modelo em magnitude e direção, ajustando apenas a direção durante o processo de ajuste fino. Isso permite atualizar significativamente os parâmetros sem modificar a arquitetura do modelo.

- Vantagens: Melhora a capacidade de aprendizado, mantém a eficiência do ajuste fino.
- Desvantagens: Pode ser mais complexo do que o LoRA, requer ajuste fino meticuloso de hiperparâmetros.

- Aplicações: Ajuste fino de LLMs para tarefas específicas, transferência de conhecimento, especialmente em cenários que exigem ajuste fino completo.

Ajuste Fino pela Metade (Half Fine-Tuning): O ajuste fino pela metade congela metade dos parâmetros do modelo durante cada rodada de ajuste fino, enquanto a outra metade é atualizada. Isso ajuda a manter o conhecimento do modelo pré-treinado, ao mesmo tempo em que permite que o modelo aprenda novas tarefas.

- Vantagens: Ajuda a manter o conhecimento pré-treinado, melhora o desempenho em tarefas específicas, é relativamente fácil de implementar.
- Desvantagens: Pode não ser tão preciso quanto o ajuste fino completo.
- Aplicações: Ajuste fino de LLMs para tarefas específicas, transferência de conhecimento.

Ajuste Fino de Memória Lamini (Lamini Memory Tuning): O ajuste fino de memória Lamini usa uma arquitetura específica que inclui um grande número de ”experts” de memória, capazes de armazenar fatos específicos com precisão. Isso ajuda a melhorar o desempenho do modelo em tarefas que exigem precisão factual, como recuperação de informações.

- Vantagens: Melhora o desempenho em tarefas de precisão factual, ajuda a reduzir as alucinações.
- Desvantagens: Pode ser mais complexo de implementar do que outros métodos de ajuste fino, requer hardware específico.
- Aplicações: Tarefas que exigem precisão factual, como recuperação de informações.

Mistura de Especialistas (MoE - Mixture of Experts): A arquitetura MoE divide a computação em uma camada em vários sub-redes especializadas, chamadas ”experts”. Cada expert executa sua computação independentemente, e os resultados são combinados para produzir a saída final.

- Vantagens: Melhora o desempenho em tarefas complexas, permite a especialização do modelo.
- Desvantagens: Pode ser mais complexo de implementar do que outras arquiteturas de modelos, requer mais recursos computacionais.
- Aplicações: Tarefas que exigem precisão factual, como tradução de idiomas.

Mistura de Agentes (MoA - Mixture of Agents): MoA é uma estrutura de múltiplos agentes que usa vários LLMs para realizar tarefas específicas. Cada LLM tem um papel específico no processo, como propor respostas ou agregar respostas de outros LLMs.

- **Vantagens:** Melhora o desempenho em tarefas complexas, permite a colaboração entre vários modelos.
- **Desvantagens:** Pode ser mais complexo de implementar do que outras arquiteturas de modelos, requer mais recursos computacionais.
- **Aplicações:** Tarefas que exigem raciocínio complexo e geração de linguagem.

Otimização de Política Próxima (PPO - Proximal Policy Optimization): PPO é um algoritmo de aprendizado por reforço usado para treinar agentes que executam tarefas em ambientes complexos. PPO usa uma função de objetivo ”substituto” para atualizar a política do agente, garantindo uma atualização estável e eficiente.

- **Vantagens:** Atualizações de política estáveis e eficientes, fácil de implementar.
- **Desvantagens:** Dependente de um bom sinal de recompensa, pode ser sensível a hiperparâmetros.
- **Aplicações:** Aprendizado por reforço, ajuste fino de LLMs usando feedback humano.

Otimização de Preferência Direta (DPO - Direct Preference Optimization): DPO é um método que alinha diretamente o modelo com preferências humanas, simplificando o processo de ajuste fino usando feedback humano. DPO supera o aprendizado por reforço, eliminando a necessidade de modelar recompensas explícitas e ajuste fino de hiperparâmetros.

- **Vantagens:** Alinhamento direto com as preferências humanas, mais estável e eficiente do que o aprendizado por reforço.
- **Desvantagens:** Requer dados de preferência de alta qualidade, pode não ser tão preciso quanto o aprendizado por reforço.
- **Aplicações:** Ajuste fino de LLMs para tarefas que exigem julgamento subjetivo, como geração de diálogo e escrita criativa.

Poda (Pruning): A poda visa reduzir o tamanho e a complexidade dos modelos, eliminando componentes desnecessários ou redundantes. Isso torna os modelos mais eficientes e rápidos de executar.

- **Vantagens:** Modelos menores e mais eficientes, menor uso de memória, maior robustez.
- **Desvantagens:** Pode afetar o desempenho, pode ser difícil encontrar os componentes certos para podar.
- **Aplicações:** Ajuste fino de LLMs para tarefas específicas, desdobramento em ambientes com recursos limitados.

ORPO (Optimised Routing and Pruning Operations): ORPO combina poda com otimização de roteamento, tentando minimizar o uso de recursos computacionais sem afetar muito o desempenho.

- **Vantagens:** Melhora a eficiência e o desempenho dos modelos, menor uso de recursos.
- **Desvantagens:** Pode ser mais complexo de implementar do que a poda.
- **Aplicações:** Desdobramento de LLMs em ambientes com recursos limitados.

Otimização e Avaliação (Optimization and Evaluation):

- **Função de perda de entropia cruzada (Cross-Entropy Loss Function):** A entropia cruzada é uma métrica chave usada para avaliar o desempenho dos modelos de linguagem durante o treinamento ou o ajuste fino. Ela mede a diferença entre duas distribuições de probabilidade - a distribuição prevista pelo modelo e a distribuição real dos dados.
- **Perplexidade (Perplexity):** A perplexidade mede o quão bem uma distribuição de probabilidade ou um modelo prevê uma amostra. No contexto dos LLMs, ela avalia a incerteza do modelo sobre a próxima palavra em uma sequência. Uma perplexidade mais baixa indica melhor desempenho, pois o modelo está mais confiante em suas previsões.
- **Factualidade (Factuality):** A factualidade avalia a precisão das informações geradas pelo LLM. É especialmente importante para aplicações onde informações imprecisas podem ter consequências graves. Pontuações de factualidade mais altas estão correlacionadas com uma qualidade de saída mais alta.
- **Incerteza do LLM (LLM Uncertainty):** A incerteza do LLM é medida usando a probabilidade logarítmica, o que ajuda a identificar gerações de baixa qualidade. Uma incerteza mais baixa indica uma qualidade de saída mais alta.
- **Perplexidade do Prompt (Prompt Perplexity):** Essa métrica avalia o quão bem o modelo entende o prompt de entrada. Uma perplexidade do prompt mais baixa indica um prompt claro e compreensível, o que provavelmente levará a um melhor desempenho do modelo.
- **Relevância do Contexto (Context Relevance):** Em sistemas de geração aumentados por recuperação (RAG), a relevância do contexto mede o quão pertinente o contexto recuperado é à consulta do usuário. Uma relevância de contexto mais alta melhora a qualidade das respostas geradas, garantindo que o modelo utilize as informações mais relevantes.
- **Completude (Completeness):** A completude avalia se a resposta do modelo aborda completamente a consulta com base nas informações fornecidas. Uma completude

alta garante que todas as informações relevantes estejam incluídas na resposta, o que aumenta sua utilidade e precisão.

- **Atribuição e Utilização de Pedaços (Chunk Attribution and Utilisation):** Essas métricas avaliam o quão eficazmente os pedaços de informação recuperados contribuem para a resposta final. Pontuações mais altas de atribuição e utilização de pedaços indicam que o modelo está usando eficazmente o contexto disponível para gerar respostas precisas e relevantes.
- **Potencial de Erro de Dados (Data Error Potential):** Essa métrica quantifica a dificuldade que o modelo enfrenta para aprender com os dados de treinamento. Uma qualidade de dados mais alta resulta em um potencial de erro mais baixo, o que leva a um melhor desempenho do modelo.
- **Métricas de Segurança (Safety Metrics):** As métricas de segurança garantem que as saídas do LLM sejam apropriadas e não-prejudiciais. Elas são incluídas nas seções finais do capítulo.
- **Análise da Curva de Perda de Treinamento (Training Loss Curve Analysis):** A curva de perda de treinamento plota o valor da perda em relação às épocas de treinamento e é essencial para monitorar o desempenho do modelo.
- **Técnicas para Evitar Overfitting (Techniques to Avoid Overfitting):** Regularização, parada precoce, dropout, validação cruzada, normalização por lote, conjuntos de dados maiores e tamanhos de lote maiores.
- **Detecção de Desvio (Drift Detection):** A detecção de desvio monitora continuamente a distribuição dos dados de entrada e o desempenho do modelo. Avaliando regularmente o modelo em conjuntos de dados de retenção para detectar mudanças nos dados de entrada ou no desempenho do modelo.

Escalabilidade e Desempenho (Scalability and Performance):

- **Paralelismo de Dados (Data Parallelism):** O paralelismo de dados distribui os dados de treinamento em vários GPUs ou nós, o que reduz significativamente o tempo de treinamento.
- **Paralelismo de Modelos (Model Parallelism):** O paralelismo de modelos divide o modelo de linguagem em vários GPUs ou nós, o que permite treinar modelos muito maiores do que seria possível em um único dispositivo.
- **Checkpointing:** Checkpointing salva regularmente os pesos do modelo durante o treinamento, de modo que, se o treinamento for interrompido, o modelo possa ser reiniciado a partir do último checkpoint salvo. Isso ajuda a evitar a perda de tempo e recursos de treinamento.
- **Treinamento de Precisão Mista (Mixed Precision Training):** O treinamento de precisão mista usa diferentes tipos de ponto flutuante para reduzir o uso de memória e aumentar a eficiência computacional.

Desafios e Oportunidades:

- **Alinhamento com a Tarefa (Alignment with the Task):** É essencial que o modelo pré-treinado esteja alinhado com a tarefa específica que você deseja executar. Por exemplo, um modelo pré-treinado para tradução de idiomas pode não ser ideal para tarefas de geração de texto.
- **Compreensão do Modelo Pré-treinado (Understanding the Pre-trained Model):** É crucial entender a arquitetura, as capacidades e as limitações do modelo pré-treinado antes de ajustá-lo finamente.
- **Disponibilidade e Compatibilidade (Availability and Compatibility):** É importante escolher um modelo pré-treinado que seja compatível com seu framework e hardware. Você também deve considerar a disponibilidade de atualizações e manutenção.
- **Arquitetura do Modelo (Model Architecture):** Cada arquitetura de modelo tem seus pontos fortes e fracos. É importante escolher uma arquitetura que seja adequada para a tarefa específica que você deseja executar.
- **Restrições de Recursos (Resource Constraints):** O ajuste fino de LLMs é intensivo em recursos. É preciso considerar cuidadosamente os requisitos de hardware, memória e computação antes de iniciar o processo de ajuste fino.
- **Privacidade (Privacy):** O ajuste fino geralmente envolve o uso de conjuntos de dados sensíveis ou exclusivos. É importante tomar medidas para proteger a privacidade das informações durante o processo de ajuste fino, como usar técnicas de privacidade diferencial ou aprendizagem federada.
- **Custo e Manutenção (Cost and Maintenance):** O ajuste fino de LLMs pode ser caro, especialmente para modelos de linguagem de grande porte. É preciso considerar os custos de hardware, memória e computação, bem como os custos de manutenção contínua do modelo.
- **Tamanho do Modelo e Quantização (Model Size and Quantization):** Os modelos de linguagem de grande porte podem ser muito grandes, tornando difícil seu desdobramento e execução em dispositivos com recursos limitados. A quantização pode ser usada para reduzir o tamanho do modelo e torná-lo mais eficiente.
- **Conjuntos de Dados Pré-treinamento (Pre-training Datasets):** É crucial avaliar cuidadosamente os conjuntos de dados usados para treinar o modelo pré-treinado. Isso ajudará você a entender as limitações do modelo e a garantir que ele seja adequado para a tarefa específica que você deseja executar.
- **Consciência de Viés (Bias Awareness):** É importante ter cuidado com possíveis vieses nos modelos pré-treinados. Os vieses podem ser introduzidos a partir de conjuntos de dados enviesados ou de algoritmos de treinamento enviesados.

Desafios e Oportunidades:

- **Heterogeneidade de Dados e Domínios (Data Heterogeneity and Domains):** Os dados de treinamento geralmente são heterogêneos e vêm de vários domínios. É importante garantir que os dados sejam de alta qualidade e relevantes para a tarefa específica que você deseja executar.
- **Tamanho de Dados (Data Size):** O ajuste fino de LLMs geralmente requer grandes quantidades de dados de treinamento. É preciso considerar os desafios de armazenamento, pré-processamento e carregamento desses dados.
- **Pré-processamento de Dados (Data Preprocessing):** É crucial pré-processar os dados de treinamento para garantir que eles sejam limpos, consistentes e compatíveis com o modelo.
- **Anotação de Dados (Data Annotation):** A anotação de dados pode ser um processo demorado e caro. É importante considerar o uso de técnicas de anotação semiautomática ou automatizada.
- **Aumento de Dados (Data Augmentation):** O aumento de dados pode ser usado para aumentar o tamanho dos conjuntos de dados de treinamento, ajudando o modelo a generalizar melhor para novos dados.
- **Geração de Dados Sintéticos (Synthetic Data Generation):** A geração de dados sintéticos pode ser usada para gerar dados de treinamento quando os dados reais são escassos ou caros.
- **Integração com Tecnologias Emergentes (Integration with Emerging Technologies):** A integração de LLMs com tecnologias emergentes, como a Internet das Coisas (IoT) e computação de ponta, abre novas oportunidades para aprimorar o desempenho de LLMs e ampliar suas aplicações.
- **Aprendizagem Contínua (Continuous Learning):** A aprendizagem contínua é uma área de pesquisa em desenvolvimento que visa permitir que os LLMs aprendam continuamente com novos dados, sem precisar ser retrabalhados do zero. Isso ajudará os LLMs a manter sua precisão e relevância ao longo do tempo.

Plataformas e Frameworks de Fine-tuning utilizados pela indústria:

- **AutoTrain:** AutoTrain é uma plataforma que automatiza o processo de ajuste fino dos LLMs, tornando-o acessível mesmo para aqueles com pouca experiência em aprendizado de máquina.
- **Transformers Library and Trainer API:** A Transformers Library é uma biblioteca abrangente que oferece uma variedade de modelos pré-treinados e ferramentas para ajustar finamente modelos de linguagem de grande porte. O Trainer API automatiza e gerencia as complexidades do processo de ajuste fino.
- **Optimum:** Optimum é uma ferramenta que otimiza o desdobramento dos LLMs, aplicando uma variedade de otimizações específicas de hardware, como quantização, poda e destilação do modelo.

- **Amazon SageMaker JumpStart:** Amazon SageMaker JumpStart é uma ferramenta que fornece uma biblioteca rica de modelos pré-treinados e soluções que podem ser rapidamente personalizados para vários casos de uso.
- **OpenAI's Fine-Tuning API:** O OpenAI's Fine-Tuning API é uma plataforma que facilita a personalização dos modelos OpenAI pré-treinados para tarefas e domínios específicos.
- **NVIDIA NeMo Customizer:** O NVIDIA NeMo Customizer é parte do framework NeMo, que visa facilitar o ajuste fino de modelos de linguagem de grande porte para tarefas e domínios específicos.

Ferramentas (Tools):

- **Transformers Library:** Uma biblioteca abrangente que fornece uma variedade de modelos pré-treinados e ferramentas para ajustar finamente modelos de linguagem de grande porte.
- **Trainer API:** Uma API que automatiza e gerencia as complexidades do processo de ajuste fino.
- **Optimum:** Uma ferramenta que otimiza o desdobramento de LLMs, aplicando uma variedade de otimizações específicas de hardware, como quantização, poda e destilação do modelo.
- **TensorFlow Federated (TFF):** Uma biblioteca que fornece ferramentas para executar aprendizagem federada, permitindo o treinamento de modelos em dispositivos distribuídos sem compartilhar os dados.
- **PySyft:** Uma biblioteca que fornece ferramentas para executar a computação federada, permitindo o treinamento de modelos em dispositivos distribuídos sem compartilhar os dados.
- **HuggingFace:** Uma plataforma que fornece infraestrutura para o desenvolvimento, desdobramento e compartilhamento de modelos de linguagem de grande porte.
- **Amazon Bedrock:** Um serviço gerenciado que fornece acesso a modelos de base de ponta de vários fornecedores de IA, incluindo o AI21 Labs, Anthropic, Cohere, Meta, Mistral AI, Stability AI e Amazon.
- **Azure Machine Learning:** Uma plataforma que fornece ferramentas para o desenvolvimento, treinamento, desdobramento e monitoramento de modelos de machine learning.
- **Azure OpenAI Service:** Um serviço que fornece acesso aos modelos OpenAI, como GPT-3 e GPT-4.
- **LangChain:** Uma biblioteca que fornece ferramentas para construir aplicativos usando LLMs, com workflows modulares e personalizáveis.

- **NVIDIA NeMo:** Um framework que fornece ferramentas e modelos para o desenvolvimento, treinamento e desdobramento de modelos de linguagem de grande porte.

Desafios:

- **Escalabilidade (Scalability):** O ajuste fino de LLMs é um processo intensivo em recursos. É preciso considerar cuidadosamente os requisitos de hardware, memória e computação antes de iniciar o processo de ajuste fino.
- **Ética (Ethics):** É importante considerar as implicações éticas do ajuste fino dos LLMs, especialmente no que diz respeito a viés, justiça, privacidade e segurança.
- **Desenvolvimento de Novas Tecnologias (New Technology Development):** O campo do ajuste fino dos LLMs está em constante evolução. É preciso acompanhar as novas tecnologias e técnicas, e considerar os desafios e oportunidades associados a essas tecnologias.
- **Integração com Tecnologias Emergentes (Integration with Emerging Technologies):** A integração de LLMs com tecnologias emergentes, como a Internet das Coisas (IoT) e a computação de ponta, abre novas oportunidades para aprimorar o desempenho de LLMs e ampliar suas aplicações.

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 2 de out. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

Victor Emanuel da Silva Monteiro

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Nesta Semana, estudei artigos que analisam o desempenho de diferentes técnicas aplicadas a LLMs, como uma comparação entre In-Context Learning e Few-shot Fine-tuning. Também aprofundei em abordagens como Mixture-of-Agents e Chain-of-Thought. Além disso, investiguei o custo computacional de algumas dessas técnicas de fine-tuning e quais alternativas oferecem melhor eficiência em cenários com poucos recursos.

Few-shot Fine-tuning vs. In-context Learning- A Fair Comparison and Evaluation.pdf
- Resumo: Resumo: Few-shot Fine-tuning vs. In-context Learning: A Fair Comparison and Eva…

Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.pdf
- Resumo: Resumo: Chain-of-Thought Prompting Elicits Reasoning in Large Language Models

Mixture-of-Agents Enhances Large Language Model Capabilities.pdf
- Resumo: Mixture-of-Agents Enhances Large Language Model Capabilities

COMPARATIVE ANALYSIS OF DIFFERENT EFFICIENT FINE TUNING METHODS OF LARGE LAN…
- Resumo: Resumo: COMPARATIVE ANALYSIS OF DIFFERENT EFFICIENT FINE-TUNING M…

Dentre as principais descobertas, destaco quatro:

- **Fine-tuning costuma superar In-context Learning** em modelos maiores, com mais de 30 bilhões de parâmetros, apresentando também menor tempo de inferência, embora haja ressalvas.
- **Chain-of-Thought Prompting** melhora o desempenho de LLMs para tarefas complexas, mas é

ineficaz para modelos menores.

- **Mixture-of-Agents (MoA)** oferece melhor desempenho que grandes modelos individuais, com eficiência de custo superior, usando colaboração entre múltiplos LLMs.
- **Low-Rank Adaptation (LoRA) e Context Distillation** são alternativas de fine-tuning eficientes em termos de custo, com boa generalização em ambientes de poucos recursos.

Planejamento: **[descrever o que pretende fazer para realizar a próxima ENTREGA]**

Pretendo investigar frameworks que suportam as técnicas analisadas,
- Meu objetivo é encontrar ferramentas que facilitam o uso dessas abordagens

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

Resumo: Few-shot Fine-tuning vs. In-context Learning: A Fair Comparison and Evaluation

Informações do paper que merecem destaque:

- **Vantagem do FT em grandes modelos**: O artigo mostra que, à medida que o tamanho do modelo aumenta, o Fine-tuning supera o In-context Learning em termos de generalização fora do domínio. Para modelos com 30 bilhões de parâmetros, o FT não apenas se iguala, mas em alguns casos, supera o ICL .
- **Variabilidade de desempenho**: Tanto o FT quanto o ICL apresentaram variações significativas de desempenho dependendo do tamanho do modelo, do número de exemplos usados e dos padrões de entrada escolhidos. Essa variabilidade destaca a complexidade da adaptação robusta de tarefas, e que a escolha do método e da configuração precisa ser feita de forma cautelosa.

1. Abstract

- O artigo compara duas estratégias de adaptação de tarefas para modelos de linguagem: Fine-tuning (FT) e In-context Learning (ICL).
- Embora o ICL tenha ganho popularidade devido à sua simplicidade e generalização fora do domínio (OOD), o estudo questiona se as fraquezas do FT em OOD são inerentes ou relacionadas ao tamanho do modelo usado nas comparações anteriores.
- Os resultados mostram que, ao controlar o tamanho do modelo, o FT pode generalizar tão bem quanto o ICL fora do domínio, especialmente à medida que o tamanho do modelo aumenta.

2. Introduction

- A adaptação de modelos de linguagem pré-treinados para tarefas específicas é fundamental no processamento de linguagem natural.
- O FT envolve treinar um modelo em um conjunto de dados específico, enquanto o ICL ajusta o comportamento de um modelo sem alterar seus parâmetros.
- Comparações anteriores entre FT e ICL não foram justas devido ao uso de modelos de tamanhos diferentes, e o artigo propõe uma comparação justa usando os mesmos modelos e condições.

3. Background

- Fine-tuning (FT): O FT baseado em padrões (PBFT) envolve o ajuste de parâmetros do modelo usando um padrão para transformar a tarefa em um problema de modelagem de linguagem.
- In-context Learning (ICL): ICL adapta o modelo a uma tarefa ao fornecer exemplos de entrada e saída, sem atualizar os parâmetros do modelo.

4. A fair comparison of FT and ICL

- A comparação justa foi realizada com modelos OPT de diferentes tamanhos (125M a 30B de parâmetros) em tarefas de classificação de texto.
- Para garantir a justiça, foram usados os mesmos conjuntos de exemplos e o mesmo número de parâmetros em ambos os métodos.

5. Results

- Desempenho in-domain: Tanto o ICL quanto o FT mostraram melhorias no desempenho conforme o tamanho do modelo aumentava.
- Desempenho OOD: Embora o ICL tenha apresentado melhor desempenho OOD para os menores modelos, os modelos maiores de FT eventualmente superaram o ICL.
- Variabilidade: Ambos os métodos mostraram variabilidade no desempenho dependendo dos padrões usados e do tamanho dos modelos.

6. Comparing FT and ICL

- Interação do usuário: ICL é mais fácil de usar para não especialistas, enquanto o FT requer mais experiência técnica.
- Tempo de inferência: O FT tem menor tempo de inferência, pois o ICL requer que todos os exemplos de demonstração sejam processados a cada nova entrada.
- Reutilização: O ICL é mais reutilizável, pois o mesmo modelo pode ser usado para várias tarefas, enquanto o FT resulta em modelos especializados.

7. Discussion

- Ambos os métodos podem ser aprimorados com novas técnicas, e há uma necessidade de entender melhor os desafios e as características de cada abordagem.
- O artigo sugere que, embora o ICL tenha vantagens em facilidade de uso e reutilização, o FT ainda é mais aplicável em cenários de poucos recursos.

8. Related Work

- O trabalho destaca pesquisas anteriores que compararam FT e ICL, mostrando que essas comparações muitas vezes favoreceram o ICL devido ao uso de modelos de tamanhos diferentes.

9. Conclusion

- Ambos os métodos de adaptação de tarefas podem generalizar bem dentro e fora do domínio.
- O artigo sugere que o FT pode superar o ICL em cenários OOD, especialmente quando mais dados estão disponíveis para o treinamento.

10. Limitations

- O artigo reconhece que se concentra em uma forma específica de generalização fora do domínio e que outros tipos de generalização podem levar a conclusões diferentes.
- Também limita a comparação a modelos do tipo OPT e não explora outras arquiteturas de modelos, como encoder-decoder.

Resumo: Chain-of-Thought Prompting Elicits Reasoning in Large Language Models

Informações do paper que merecem destaque:

- Chain of thought apresenta maiores ganhos de desempenho em problemas mais complexos.
- Chain of thought não impacta positivamente o desempenho em modelos pequenos e só apresenta ganhos de desempenho quando usado com modelos de aproximadamente 100 bilhões de parâmetros.

1. Resumo (Abstract):

- **Utilidade**: O artigo demonstra como a geração de cadeias de pensamento melhora o raciocínio em grandes modelos de linguagem.
- **Aplicação**: Cadeias de pensamento ajudam em tarefas complexas como aritmética, raciocínio de senso comum e simbólico.
- **Impacto**: Mostra resultados impressionantes, como a superação do estado da arte em benchmarks, usando exemplos simples de cadeias de pensamento.

2. Introdução:

- **Escalabilidade**: Modelos maiores melhoram o desempenho, mas ainda enfrentam dificuldades em tarefas complexas, como raciocínio simbólico e de senso comum.
- **Solução Proposta**: O prompting com cadeias de pensamento combina raciocínios intermediários e aprendizado com poucos exemplos (few-shot learning).
- **Vantagem**: Esse método permite usar um único modelo para diversas tarefas sem a necessidade de treinar modelos específicos para cada tarefa.

3. Chain-of-Thought Prompting:

- **Processo de Raciocínio**: A cadeia de pensamento imita o processo de raciocínio humano, decompondo tarefas complexas em etapas intermediárias.
- **Transparência**: O método permite observar o raciocínio do modelo, facilitando o diagnóstico de erros.
- **Versatilidade**: O prompting com cadeia de pensamento pode ser aplicado a qualquer tarefa que humanos possam resolver por meio da linguagem.

4. Raciocínio Aritmético (Arithmetic Reasoning):

- **Melhoria de Desempenho**: Cadeias de pensamento melhoram significativamente o desempenho em problemas matemáticos.
- **Comparação com Modelos Ajustados**: O prompting com cadeia de pensamento alcança desempenho similar a modelos ajustados para tarefas específicas.
- **Generalização**: A abordagem é capaz de resolver diversos tipos de problemas matemáticos sem a necessidade de ajustes finos.

5. Estudo de Caso: Comportamento de Cadeias de Pensamento (Case Studies: Chain-of-Thought Behavior):

- **Correção de Erros**: Modelos grandes geram raciocínios corretos na maioria das vezes, mas ainda cometem erros menores, como cálculos incorretos.
- **Escalabilidade**: O aumento no tamanho do modelo corrige muitos erros que ocorrem em modelos menores.
- **Interpretação de Resultados**: Cadeias de pensamento ajudam a entender onde o modelo pode falhar, oferecendo insights sobre o raciocínio por trás das respostas corretas e incorretas.

6. Estudo de Robustez (Robustness Study):

- **Sensibilidade ao Estilo de Exemplos**: Diferentes estilos de exemplos de cadeia de pensamento podem impactar o desempenho, mas sempre superam o prompting padrão.
- **Variabilidade de Resultados**: Embora haja variações dependendo dos exemplos usados, o prompting com cadeia de pensamento consistentemente melhora o desempenho.
- **Robustez**: O método é robusto a diferentes anotadores e ordens de exemplares, demonstrando flexibilidade na sua aplicação.

7. Raciocínio de Senso Comum (Commonsense Reasoning):

- **Aplicabilidade em Tarefas Diversas**: Cadeias de pensamento também melhoram o desempenho em tarefas que envolvem raciocínio baseado em conhecimento do mundo.
- **Superação do Estado da Arte**: Modelos como o PaLM 540B superam benchmarks anteriores em tarefas de raciocínio de senso comum.
- **Desafios Mínimos**: Apesar de ganhos em muitas tarefas, algumas, como CSQA, apresentam melhorias mínimas, indicando áreas onde o método pode ser menos eficaz.

8. Raciocínio Simbólico (Symbolic Reasoning):

- **Desempenho em Tarefas Simples**: Cadeias de pensamento ajudam modelos a resolver tarefas simbólicas simples, como concatenação de letras e flips de moeda.
- **Generalização para Sequências Maiores**: O prompting com cadeias de pensamento facilita a generalização para problemas com sequências mais longas do que as vistas durante o treino.
- **Limitações de Modelos Pequenos**: Apenas modelos maiores, com mais de 100 bilhões de parâmetros, conseguem resolver essas tarefas de maneira consistente.

9. Discussão (Discussion):

- **Habilidade Emergente**: O raciocínio por cadeias de pensamento é uma habilidade emergente que se manifesta em modelos de grande escala.
- **Expansão de Tarefas**: O método amplia a gama de tarefas que os modelos de linguagem conseguem resolver, indo além das capacidades do prompting padrão.
- **Limitações**: Embora útil, o método só é eficaz em modelos grandes, o que pode ser um obstáculo para uso em aplicações reais devido aos altos custos computacionais.

10. Trabalhos Relacionados (Related Work):

- **Raciocínio com Passos Intermediários**: O artigo se baseia em estudos anteriores que exploraram o uso de raciocínios intermediários para resolver problemas de raciocínio.
- **Prompting Few-Shot**: O método se relaciona com o conceito de prompting de poucos exemplos, expandindo essa abordagem para incluir raciocínio mais elaborado.
- **Contribuição Original**: A principal inovação é a aplicação das cadeias de pensamento para uma variedade de tarefas sem a necessidade de ajuste fino.

11. Conclusões (Conclusion):

- **Método Simples e Eficaz**: O prompting com cadeias de pensamento é uma maneira simples de melhorar o raciocínio em grandes modelos de linguagem.
- **Eficácia em Diversas Tarefas**: O método melhora o desempenho em raciocínio aritmético, simbólico e de senso comum, demonstrando sua aplicabilidade generalizada.
- **Direções Futuras**: O artigo sugere que trabalhos futuros explorem formas de melhorar a factualidade dos raciocínios gerados e investiguem como habilitar raciocínio em modelos menores.

Resumo: Mixture-of-Agents Enhances Large Language Model Capabilities

Informações do paper que merecem destaque:

- **Colaboratividade entre LLMs**: O artigo revela que os modelos de linguagem (LLMs) tendem a melhorar suas respostas quando têm acesso às saídas de outros modelos, mesmo que essas saídas sejam de qualidade inferior. Esse fenômeno, chamado de *colaboratividade*, é central para a eficácia da abordagem Mixture-of-Agents (MoA).
- **Desempenho superior ao GPT-4 Omni**: A abordagem MoA, utilizando apenas modelos de código aberto, alcança uma taxa de vitória de 65.1% no benchmark AlpacaEval 2.0, superando significativamente o GPT-4 Omni, que obteve 57.5%. Isso demonstra que a colaboração entre múltiplos LLMs pode superar até mesmo os maiores modelos individuais.
- **Eficiência de custo e desempenho**: O MoA não só melhora a qualidade das respostas, como também é altamente eficiente em termos de custo. A versão MoA-Lite, por exemplo, oferece um equilíbrio eficaz entre custo e qualidade, superando o GPT-4 Turbo em qualidade de resposta em cerca de 4%, enquanto é mais de duas vezes mais eficiente em termos de custo.

1. Introdução

- O artigo explora como alavancar os pontos fortes coletivos de vários modelos de linguagem (LLMs) por meio da metodologia Mixture-of-Agents (MoA).
- Propõe-se uma arquitetura em camadas onde múltiplos agentes LLM trabalham em colaboração, refinando as respostas uns dos outros.

- Os experimentos mostram que o MoA supera modelos como o GPT-4 Omni, especialmente em benchmarks como AlpacaEval 2.0, MT-Bench, e FLASK, destacando-se ao usar apenas modelos de código aberto.

2. Metodologia Mixture-of-Agents

- O conceito de colaboratividade entre LLMs é central para o MoA. LLMs tendem a gerar respostas melhores quando têm acesso às saídas de outros modelos.
- Dois papéis são destacados:
  - **Proposers** (propositores), que geram referências
  - **Aggregators** (agregadores), que sintetizam respostas.
- A estrutura do MoA tem múltiplas camadas, onde os agentes refinam as respostas de camadas anteriores até chegar à final, sem necessidade de ajuste fino.

3. Colaboratividade dos LLMs

- A colaboratividade é testada com diferentes modelos e demonstra que mesmo respostas de baixa qualidade de outros modelos podem melhorar as respostas de um LLM.
- Alguns modelos são melhores em propor (como o WizardLM) e outros em agregar (como GPT-4o e LLaMA-3), demonstrando a diversidade de capacidades.

4. Mixture-of-Agents

- O MoA utiliza uma série de camadas com vários LLMs que refinam as respostas.
- Cada camada pode usar os mesmos modelos ou novos, e o processo é repetido até chegar a uma resposta robusta.
- A agregação final é feita através de prompts que incentivam a síntese crítica das respostas.

5. Analogia com Mixture-of-Experts

- O MoA se inspira no Mixture-of-Experts (MoE), que usa redes especializadas em diferentes tarefas.

- Diferentemente do MoE, o MoA opera no nível de modelos completos e utiliza apenas prompts e geração, sem modificação nas ativações internas dos modelos.

6. Avaliação

6.1 Configuração

- Modelos são avaliados em benchmarks como AlpacaEval 2.0, MT-Bench, e FLASK para comparar a qualidade de resposta dos modelos.

6.2 Resultados de Benchmark

- O MoA apresenta melhorias significativas, superando o GPT-4 Omni no AlpacaEval 2.0 e no FLASK, com maior qualidade de resposta e maior eficiência de custo.

6.3 O Que Faz o MoA Funcionar Bem?

- O MoA supera abordagens baseadas em ranqueamento de LLMs, pois agrega respostas de forma mais sofisticada.
- O uso de respostas de vários modelos (proposers) resulta em uma melhoria contínua nas respostas sintetizadas.

6.4 Análise de Orçamento e Tokens

- Uma análise do orçamento de inferência e do uso de tokens mostra que o MoA é mais eficiente em termos de custo em comparação com outros modelos de ponta, oferecendo melhor valor pelo desempenho.

7. Trabalhos Relacionados

7.1 Raciocínio em LLMs

- Estudos recentes focam em otimizar LLMs para tarefas específicas por meio de engenharia de prompts, como as técnicas Chain of Thought (CoT) e Tree of Thought (ToT).

7.2 Ensemble de Modelos

- Diferentes abordagens, como reranking de saídas e colaboração entre múltiplos modelos LLMs, são exploradas para aproveitar melhor as capacidades dos LLMs.

8. Conclusão

- O artigo conclui que o MoA melhora substancialmente a qualidade das respostas dos LLMs ao integrar as forças coletivas de múltiplos agentes, demonstrando uma taxa de vitória de 65% no benchmark AlpacaEval 2.0.
- Aponta para futuras otimizações do MoA para reduzir a latência e melhorar a experiência do usuário.

Apêndices

Correlação de Spearman usando Funções de Similaridade Diferentes

- As funções de similaridade TF-IDF e Levenshtein mostram correlações positivas entre a qualidade das respostas agregadas e as preferências de avaliadores baseados no GPT-4.

LLM Ranker

- Um sistema de ranqueamento baseado em LLMs é usado para comparar saídas de diferentes modelos, selecionando a melhor resposta com base na perspectiva humana.

Estudo de Caso

- Apresenta exemplos práticos de como o MoA agrega respostas de diferentes modelos, demonstrando a eficácia da agregação de propostas de múltiplos agentes.

Tarefa de Matemática (MATH)

- O MoA também é eficaz em tarefas de raciocínio matemático, superando abordagens existentes em benchmarks como o MATH.

Resumo: COMPARATIVE ANALYSIS OF DIFFERENT EFFICIENT FINE-TUNING METHODS OF LARGE LANGUAGE MODELS (LLMS) IN LOW-RESOURCE SETTING

Informações do paper que merecem destaque:

- 1. **Desempenho da Destilação de Contexto (Context Distillation):** Técnica de destilação de contexto, que treina um modelo “estudante” para replicar as saídas de um modelo “professor”, supera os métodos tradicionais de fine-tuning em termos de generalização para dados fora do domínio (Out-of-Domain - OOD). Isso demonstra que essa técnica pode ser uma alternativa promissora, especialmente quando os recursos computacionais são limitados.
- 2. **Desempenho Consistente do LoRA:** A adaptação de baixa ordem (Low-Rank Adaptation - LoRA) demonstrou ser um método eficiente para reduzir os requisitos computacionais durante o fine-tuning, ao decompor matrizes de peso em submatrizes menores. Embora apresente desempenho semelhante aos métodos tradicionais de fine-tuning, o uso do LoRA é especialmente relevante para ambientes com restrições de memória e tempo de treinamento.
- 3. **Importância do Tamanho do Modelo e dos Dados:** Os experimentos revelaram que, conforme o tamanho do modelo aumenta, o desempenho do fine-tuning padrão (Vanilla FT) e do fine-tuning baseado em padrões (PBFT) melhora em tarefas de classificação de sequência. Contudo, o modelo maior (OPT 350M) nem sempre generaliza tão bem quanto o modelo menor (OPT 125M) em domínios fora do padrão, destacando o impacto das configurações do modelo e dos dados disponíveis.

-

Resumo

O artigo explora e compara diferentes métodos de fine-tuning de modelos de linguagem grande (LLMs) em ambientes de poucos recursos. Ele avalia métodos como o Fine-Tuning (FT) padrão, Fine-Tuning baseado em padrões (PBFT), Low-Rank Adaptation (LoRA), e Context Distillation. O estudo foca em duas tarefas de classificação de sequência usando os datasets COLA e MNLI. Os resultados indicam que estratégias alternativas podem apresentar generalização comparável ao FT, com Context Distillation superando os métodos padrões em alguns casos.

1. Introdução

O objetivo do estudo é avaliar a eficácia de diferentes métodos de fine-tuning em LLMs usando tarefas de classificação de sequência. Utilizando os datasets MNLI e COLA, os autores comparam o desempenho de técnicas padrão e alternativas, como LoRA e Context Distillation, em cenários de poucos dados.

1.1 Datasets

1.1.1 MNLI

O Multi-Genre Natural Language Inference (MNLI) é um conjunto de dados com pares de sentenças anotadas para inferência textual, com um subconjunto de 261.802 amostras usadas para treinamento. Para avaliação fora do domínio (OOD), foi utilizado o dataset HANS.

1.1.2 COLA

O Corpus of Linguistic Acceptability (COLA) contém sentenças anotadas sobre sua aceitabilidade gramatical, com 10.657 sentenças, sendo 9.594 usadas para treinamento. O dataset tem 17 fontes internas e 6 fontes externas (OOD).

2. Abordagem

Os autores exploram fine-tuning em LLMs como o OPT 125M e o OPT 350M, realizando experimentos em poucos dados e ajustando hiperparâmetros como taxa

de aprendizado e camadas congeladas. O objetivo era comparar métodos tradicionais com alternativas, como LoRA e Context Distillation.

2.1 Aprendizado com Poucos Exemplos (Few-Shot Learning)

Experimentos foram realizados com diferentes quantidades de exemplos (2, 16, 32, 64, e 128), com 10 execuções por modelo para garantir robustez nos resultados.

2.2 Infraestrutura e Replicação

Os experimentos foram conduzidos usando o Google Colab Pro com GPUs Nvidia Tesla, e o código foi disponibilizado no GitHub para replicação.

3. Métodos de Fine-Tuning

3.1 Fine-Tuning Padrão (Vanilla)

Realizado nos modelos OPT 125M e OPT 350M, usando dados dos conjuntos COLA e MNLI sem prompts específicos.

3.2 Fine-Tuning Baseado em Padrões (PBFT)

Usa padrões de linguagem para guiar o processo de fine-tuning, aproveitando prompts derivados do GPT-3 para transformar os dados de entrada e melhorar o aprendizado.

3.3 Fine-Tuning Adaptativo

Método que congela camadas iniciais do modelo e ajusta dinamicamente a taxa de aprendizado para otimizar o desempenho sem sobrecarregar a memória.

4. Modelagem Eficiente com Adaptação de Baixa Ordem (LoRA)

O LoRA adapta o modelo ao decompor matrizes de peso em submatrizes de menor ordem, reduzindo o número de parâmetros e tornando o fine-tuning mais eficiente. Esse método busca equilibrar a precisão e a utilização de recursos durante o treinamento.

5. Destilação de Contexto (Context Distillation)

Este método treina o modelo estudante com base nas saídas probabilísticas de um modelo professor, utilizando uma combinação de perdas de destilação e classificação para melhorar a generalização em dados não vistos.

6. Experimentos e Resultados

A seção apresenta uma comparação detalhada entre os métodos de fine-tuning, com gráficos mostrando como diferentes tamanhos de amostra afetam a precisão em domínios internos e externos para os modelos OPT 125M e 350M.

6.1 Configuração e Hiperparâmetros

Os experimentos mantiveram consistência nos hiperparâmetros, como taxa de aprendizado, épocas e tamanhos de lotes.

6.2 Vanilla FT, PBFT e Adaptive FT

Os resultados indicam que o fine-tuning padrão (Vanilla) tem bom desempenho em domínios internos, enquanto PBFT se beneficia de tamanhos de amostra maiores. O fine-tuning adaptativo melhorou a generalização, mas apresentou resultados inferiores ao FT completo.

6.3 Modelagem Eficiente com LoRA

Os experimentos com LoRA mostraram que a decomposição de matrizes pode impactar a adaptação do modelo, sendo menos eficaz em domínios fora do padrão.

6.4 Destilação de Contexto

A destilação de contexto melhorou a precisão no domínio interno, mas teve resultados inconsistentes fora do domínio. O método combinou eficiência de tempo com alta precisão em alguns casos.

7. Conclusão

O estudo conclui que, embora o fine-tuning padrão ainda seja competitivo, métodos como Context Distillation oferecem melhorias em generalização fora do domínio. A escolha do método de fine-tuning depende dos recursos disponíveis e da tarefa específica.

Instituto de Informática (INF) - UFG
Alameda Palmeiras, Quadra D, Câmpus Samambaia
Fone: (62) 3521-1181 / Fax: (62) 3521-1182

APÊNDICE 3

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 9 de out. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

Victor Emanuel da Silva Monteiro

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Nesta semana, investiguei frameworks que suportam as técnicas analisadas nas semanas anteriores, com o objetivo de encontrar ferramentas que permitam a unificação delas.

LangChain:

- LangChain é um framework que facilita a construção de aplicações baseadas em LLMs, permitindo a criação de cadeias de prompts e possibilita o uso de Chain-of-Thought. Analisei como o LangChain pode ser útil para estruturar prompts complexos e gerenciar interações com modelos de linguagem.
- LangChain
- CreatingLargeLanguageModelApplicationsUtilizingLangChain-APrimeronDevelopingLLMApp…

Mirascope

- Framework para desenvolver agentes de linguagem (LLMs) de forma modular, extensível e confiável. Ela fornece uma abstração prática para criação personalizada de agentes e é de código aberto.
- Mirascope

Tree of Thoughts

- Expande o CoT para uma estrutura em árvore, permitindo ao modelo explorar múltiplas direções de raciocínio antes de selecionar a melhor solução. Pode ser útil para problemas que exigem um alto nível de exploração.
- Tree of Thought
- Tree of Thoughts- Deliberate Problem Solving with Large Language Models.pdf

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

- Idealizar e formalizar descrição de arquitetura de um sistema de LLMs integrando as técnicas e os frameworks estudados conforme a necessidade
- Realizar experimentos para identificar pontos necessários na validação do sistema

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

LangChain

https://mirascope.com/docs/

Ao planejar um sistema de *mixture of agents* com *Chain of Thought* (CoT), LangChain oferece ferramentas que permitem combinar agentes de forma coordenada, suportando processamento de pensamento sequencial e fluxos de raciocínio elaborados. A plataforma facilita a integração e o desenvolvimento de agentes que utilizam passos intermediários para resolver tarefas complexas e atua como uma infraestrutura para gerenciar múltiplos agentes ou modelos.

Principais Ferramentas do LangChain

1. **LangGraph:** Focado em estruturação avançada, o LangGraph permite definir o fluxo de informações entre diferentes agentes e etapas de raciocínio. Ele é especialmente útil para a configuração de ambientes onde agentes especializados em subtarefas diferentes precisam colaborar para concluir uma tarefa complexa. Com o uso de gráficos de fluxos, é possível configurar etapas onde agentes fazem inferências intermediárias, o que é vantajoso em sistemas baseados em *Chain of Thought*.

2. **LangSmith:** Esta ferramenta é voltada para o monitoramento e aprimoramento contínuo dos agentes. Com o LangSmith, você pode avaliar a performance de cada agente de maneira detalhada, rastreando a acurácia, eficiência e até o tempo de resposta. Ele é especialmente útil para sistemas em desenvolvimento que necessitam de ajustes frequentes. O LangSmith também coleta métricas que ajudam a entender como os agentes atuam individualmente e em conjunto.

3. **LangServe:** Voltada para a implantação e escalabilidade, o LangServe permite transformar um conjunto de agentes em APIs utilizáveis para aplicações externas. É possível hospedar agentes LangChain e disponibilizá-los como serviços independentes ou como parte de uma arquitetura maior. Isso é vantajoso para empresas ou desenvolvedores que querem colocar suas aplicações em produção, oferecendo robustez e suporte a escalabilidade.

4. **Prompt Management:** A LangChain inclui uma ferramenta para gerenciamento e construção de *prompts* de forma detalhada, permitindo definir cadeias de prompts adaptativas para agentes com diferentes habilidades e contextos de aplicação. Isso é particularmente importante para o desenvolvimento de agentes que realizam raciocínios intermediários com *Chain of Thought*, onde cada passo deve ser formulado de maneira a dar continuidade lógica ao próximo.

5. **Memory Tools:** Os sistemas de memória disponíveis na LangChain ajudam a manter contexto ao longo de múltiplas interações, armazenando informações temporárias que podem ser consultadas por outros agentes na cadeia. Essa capacidade é importante para aplicações que exigem persistência e continuidade de dados, como chatbots, sistemas de recomendação e assistentes digitais que precisam lembrar de interações passadas para fornecer respostas precisas.

6. **Integração com Modelos e API's de Terceiros:** LangChain oferece uma estrutura que permite integrar facilmente modelos de LLMs (como OpenAI, Cohere, etc.) e APIs de terceiros. Essa flexibilidade possibilita combinar a inteligência do LangChain com serviços externos, expandindo as capacidades de agentes e fornecendo dados em tempo real, além de habilidades adicionais.

Vídeos úteis:

- **LangChain Explained in 13 Minutes | QuickStart Tutorial for Beginners**
- **What is LangChain?**
- **Building open source LLM agents with Llama 3**
- **Memory for agents (conceptual video)**
- **Long-Term Memory Agent Template**

PAPER: Creating Large Language Model Applications Utilizing LangChain: A Primer on Developing LLM Apps Fast

CreatingLargeLanguageModelApplicationsUtilizingLangC…

Estrutura e Funcionalidades do LangChain comentados no paper:

O LangChain organiza sua arquitetura em componentes principais para oferecer flexibilidade e modularidade:

1. Prompts: São entradas dinâmicas para o LLM, que podem incluir instruções específicas e exemplos para orientar a geração de respostas. O LangChain facilita a criação de *prompt templates*, garantindo consistência nas interações.

2. Modelos: A biblioteca permite o uso de LLMs tradicionais, modelos de chat estruturados e modelos de *text embeddings* para representar textos de forma numérica, úteis em tarefas de comparação semântica e recuperação de informações.

3. Chains: As cadeias (ou *chains*) são fundamentais para encadear uma sequência de operações. LangChain oferece *chains* simples e complexas, permitindo, por exemplo, o uso de diferentes modelos em sequência ou a implementação de cadeias que redirecionam a entrada com base no tipo de pergunta.

4. Memória: Como os LLMs são naturalmente “sem estado” e não lembram interações anteriores, o LangChain implementa componentes de memória que ajudam a armazenar o histórico de diálogos, essencial para manter o contexto em chatbots e assistentes virtuais.

5. Question Answering de Documentos: Um dos recursos mais populares é a capacidade de responder a perguntas baseadas em documentos. Para isso, o LangChain oferece suporte para carregamento, fragmentação e busca de documentos, combinados com modelos de embeddings e bancos de dados vetoriais.

6. Agentes: Esses componentes coordenam uma série de ações, escolhendo automaticamente ferramentas e operações baseadas na entrada do usuário. Agentes podem funcionar em modo de ação (escolhendo a próxima ferramenta de acordo com o histórico de interações) ou em modo de planejamento (definindo toda a sequência de ações antes de executá-las).

O artigo conclui que LangChain é uma ferramenta de grande potencial na construção de aplicativos personalizados de LLM, facilitando a integração de fontes de dados e a configuração de pipelines modulares. A abordagem flexível e modular do LangChain permite uma adaptação eficiente a diversos casos de uso, incentivando o desenvolvimento de soluções inovadoras e o avanço no uso prático de LLMs em IA.

Mirascope

https://mirascope.com/docs/

Destaques Mirascope

Mirascope é uma biblioteca versátil e acessível, projetada para otimizar o desenvolvimento com LLMs (Large Language Models) e simplificar integrações com múltiplos provedores de IA, como OpenAI, Anthropic, Cohere, entre outros. Destaca-se pela flexibilidade de integração com APIs de diversos modelos e pela organização modular, que permite uma combinação eficiente de agentes e sistemas de IA para automação e análise.

Principais Ferramentas do Mirascope

1. **Módulos de Provedores e APIs:**

Mirascope facilita chamadas de API e integrações com vários provedores de LLM. Para configurar, basta instalar o módulo do provedor específico (como OpenAI, Cohere, etc.), e definir as chaves de API correspondentes. Isso torna a biblioteca excelente para quem deseja desenvolver e testar modelos variados com a mesma interface unificada.

2. **Prompt Templates e Gerenciamento de Mensagens:**

A biblioteca fornece suporte para criação de prompts estruturados e altamente configuráveis, com capacidade de definir papéis como “USER”, “SYSTEM” e “ASSISTANT” em mensagens, o que é ideal para interações mais complexas entre usuário e IA. Isso permite formatar o diálogo de maneira sequencial e integrada, útil para quem desenvolve assistentes com múltiplos turnos de conversa e para fluxos de raciocínio.

3. **Suporte a História de Conversas e Continuidade:**

Uma das funcionalidades de destaque é o gerenciamento de histórico de conversas, com a capacidade de manter o contexto através de variáveis de memória, permitindo fluxos de interação mais naturais e contínuos. Ferramentas como o MESSAGES permitem adicionar históricos de mensagens anteriores, crucial para sistemas que requerem continuidade na comunicação, como assistentes virtuais.

4. **Flexibilidade para Ferramentas de Agentes:**

A Mirascope fornece uma série de módulos para construir agentes personalizados, com foco em tarefas específicas. Isso inclui acesso a prompts, templates de interação e módulos configuráveis, permitindo um desenvolvimento altamente customizável de agentes. Esses

agentes podem realizar desde simples recomendações até a execução de cadeias de ações, facilitando a criação de sistemas de decisão baseados em IA.

5. **Avaliações e Configurações Dinâmicas:**

Mirascope suporta avaliações de respostas, o que é valioso para ajustar o desempenho dos modelos e implementar melhorias contínuas. Além disso, a biblioteca permite configurar chamadas dinâmicas, o que possibilita ajustar o comportamento dos agentes com base em condições de entrada específicas, maximizando a eficiência e personalização dos resultados.

Conteúdos úteis

- Mirascope - Intuitively build LLM Apps
- LLM Toolkit for AI Applications - Mirascope
- Mirascope a New LLM Tool and using it with Ollama and Llama3 (9/30)

Tree of Thought

Tree of Thoughts- Deliberate Problem Solving with Large …

Principais Pontos Importantes

- Objetivo: Melhorar a capacidade de raciocínio dos modelos de linguagem (LMs) aplicando o *Tree of Thoughts* (ToT), que permite uma abordagem mais robusta e deliberada, aumentando a precisão em tarefas complexas.

- Comparação com Chain of Thought (CoT): Ao contrário do CoT, o ToT usa uma estrutura de árvore que permite explorar caminhos de raciocínio variados, auxiliando o modelo em decisões intermediárias com visão global e processos de verificação.

- Estratégia de Busca e Heurísticas: ToT integra algoritmos de busca, como busca em largura (BFS) e profundidade (DFS), com uma análise detalhada de estados intermediários para avaliar e refinar o processo de raciocínio.

- Casos de Teste Empíricos: Três tarefas complexas (Game of 24, Creative Writing e Mini Crosswords) mostraram melhorias significativas de desempenho com o ToT em comparação com métodos anteriores.

1. Introdução

A introdução discute os avanços de modelos de linguagem (LMs), como o GPT e o PaLM, para resolver tarefas complexas que exigem raciocínio matemático, simbólico, de bom senso e de conhecimento específico. No entanto, observa-se uma limitação fundamental nesses modelos: a natureza sequencial e o mecanismo de tomada de decisão *token* a *token*, que podem prejudicar a resolução de tarefas complexas e que demandam antecipação estratégica ou exploração.

Para resolver essas limitações, o artigo propõe o Tree of Thoughts (ToT), um framework que expande a abordagem do Chain of Thought (CoT). O ToT permite uma “exploração de pensamentos” ou caminhos de raciocínio múltiplos, usando uma árvore de decisões onde cada “pensamento” é um passo intermediário que ajuda o modelo a caminhar em direção à solução. O ToT também permite que o modelo autoavalie suas escolhas e faça correções, além de prever e revisar suas ações conforme necessário, criando uma abordagem de raciocínio mais global e eficaz. A introdução destaca que o ToT atingiu sucessos notáveis em tarefas que exigem planejamento detalhado, como o *Game of 24*, onde aumentou a taxa de acertos de 4% (usando CoT) para 74% com o ToT .

2. Background

Esta seção contextualiza as metodologias anteriores para resolução de problemas com LMs. O artigo descreve como métodos de entrada e saída (IO) e o Chain of Thought (CoT) estruturam a solução para problemas complexos. O CoT, por exemplo, introduz uma série de passos intermediários que guiam o LM a uma solução, particularmente útil em tarefas como questões matemáticas, onde uma sequência de equações é necessária para alcançar a resposta final.

O artigo também explora a Auto-Consistência CoT (CoT-SC), uma variante do CoT que cria múltiplas cadeias de raciocínio independentes e seleciona o resultado mais frequente como resposta final, melhorando a precisão ao explorar abordagens variadas para o mesmo problema. Ainda assim, essas abordagens têm limitações: a CoT-SC não permite exploração local de diferentes etapas intermediárias, nem inclui planejamento estratégico ou correção de erros, o que o ToT propõe solucionar .

3. Tree of Thoughts: Solução Deliberada de Problemas com Modelos de Linguagem (LM)

O ToT estrutura o processo de raciocínio como uma árvore de decisões, onde cada nó representa uma solução parcial e cada ramo uma possível continuação. O framework ToT transforma qualquer problema em uma busca em árvore, permitindo ao LM explorar e avaliar múltiplas trajetórias de raciocínio antes de escolher o melhor caminho.

Para implementar o ToT, os autores dividem o processo em quatro perguntas centrais:

1. Como decompor o processo em etapas de pensamento: O ToT permite a decomposição de cada problema em etapas de raciocínio, ajustando-as conforme a tarefa. Por exemplo, um pensamento para o *Game of 24* pode ser uma equação matemática, enquanto um pensamento para o *Creative Writing* pode ser uma introdução de um tema.

2. Como gerar pensamentos para cada estado: A geração de pensamentos no ToT pode ser feita com duas abordagens principais: (a) amostragem independente de pensamentos (para tarefas de maior complexidade) e (b) geração sequencial (útil para contextos mais restritos, como palavras cruzadas).

3. Como avaliar os estados de pensamento: O ToT utiliza uma avaliação heurística dos estados intermediários para priorizar aqueles que estão mais próximos da solução ideal. A avaliação pode ser feita de forma independente para cada estado ou comparando estados entre si, onde o modelo vota nas alternativas mais promissoras.

4. Qual algoritmo de busca utilizar: Dois algoritmos de busca foram testados: busca em largura (BFS), que mantém os estados mais promissores em cada passo, e busca em profundidade (DFS), que explora um caminho até o fim antes de retroceder. Esses algoritmos permitem uma análise sistemática do processo, viabilizando antecipação e correções durante o raciocínio .

4. Experimentos

Os autores realizaram experimentos com três tarefas desafiadoras para testar a eficácia do ToT. Essas tarefas incluíam:

- Game of 24: Uma tarefa matemática onde o objetivo é alcançar o número 24 usando quatro números e operações básicas. O ToT obteve uma taxa de sucesso de 74% em comparação a 4% com CoT e 9% com CoT-SC, demonstrando a eficiência da exploração sistemática de múltiplos caminhos de raciocínio.

- Creative Writing: Uma tarefa de redação criativa em que o modelo gera uma passagem de quatro parágrafos com base em frases aleatórias. O ToT, através de uma combinação de planos intermediários e votação, criou passagens mais coerentes e bem-estruturadas do que o CoT, com avaliações humanas preferindo as respostas do ToT.

- Mini Crosswords: Neste desafio, o modelo preenche palavras cruzadas com base em dicas, utilizando tanto BFS quanto DFS para avaliar e escolher o preenchimento correto. A abordagem ToT superou as técnicas CoT, permitindo uma análise em profundidade que resultou em respostas mais precisas .

5. Análise de Custo e Eficiência

A seção analisa o custo computacional do ToT em comparação com métodos CoT e IO. O ToT requer mais recursos computacionais devido à necessidade de explorar e avaliar múltiplas trajetórias de pensamento. Por exemplo, no *Game of 24*, o uso do ToT foi mais intensivo em termos de custo computacional, mas trouxe melhorias significativas na taxa de sucesso. Para otimizar o uso do ToT, os autores sugerem adaptações, como o uso de modelos menores para tarefas mais simples e de maior eficiência para tarefas que não exijam raciocínio exploratório profundo. Para desafios mais complexos, recomenda-se o uso de modelos avançados como o GPT-4 .

6. Discussão e Trabalho Futuro

Na conclusão, os autores discutem o potencial do ToT para ampliar as capacidades de resolução de problemas de LMs em cenários que demandam planejamento deliberado e exploração detalhada. Ao possibilitar a avaliação e exploração de múltiplos caminhos de

raciocínio, o ToT facilita maior transparência e interpretabilidade no raciocínio dos modelos, promovendo uma inteligência artificial mais alinhada aos métodos de resolução de problemas tradicionais da ciência cognitiva. Os autores sugerem que o ToT pode ser estendido para outras áreas que exigem raciocínio complexo e destacam o seu papel como um passo em direção à inteligência artificial avançada.

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião (“gate”) de aprovação: 16 de out. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

Victor Emanuel da Silva Monteiro

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Para essa entrega idealizei um sistema multiagente iterativo e dinâmico, baseado em cadeia de pensamento e colaboração entre LLMs, que aborda problemas complexos em etapas, ajusta o processo de resolução conforme necessário e integra informações externas para maior precisão.

Resumo do sistema:

- **Identificação e Planejamento**: O modelo Alpha recebe o problema, define objetivos, formula hipóteses iniciais e decide se há necessidade de pesquisa externa para obter informações adicionais.
- **Desenvolvimento de Hipóteses em Cadeia**: Os modelos A1 e A2 abordam partes específicas do problema em cada iteração, usando o método "chain of thought" para construir uma solução passo a passo e agregar profundidade ao raciocínio.
- **Avaliação e Ajustes Iterativos**: O modelo Ômega avalia os resultados dos agentes A1 e A2, verificando se os objetivos foram atingidos; caso contrário, define novos passos para continuar o processo.
- **Consultas Externas para Precisão**: Quando necessário, o sistema acessa a internet para obter informações atualizadas, enriquecendo o desenvolvimento das soluções com dados em tempo real.
- **Ciclo Adaptativo e Flexível**: A cada ciclo, o sistema decide de forma dinâmica se a solução está completa ou se novas iterações são necessárias, ajustando o número de etapas conforme a complexidade do problema.

Versão 1: Desenvolvimento de um Sistema Multiagente baseado em LLMs para Resolução Iterativa …

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

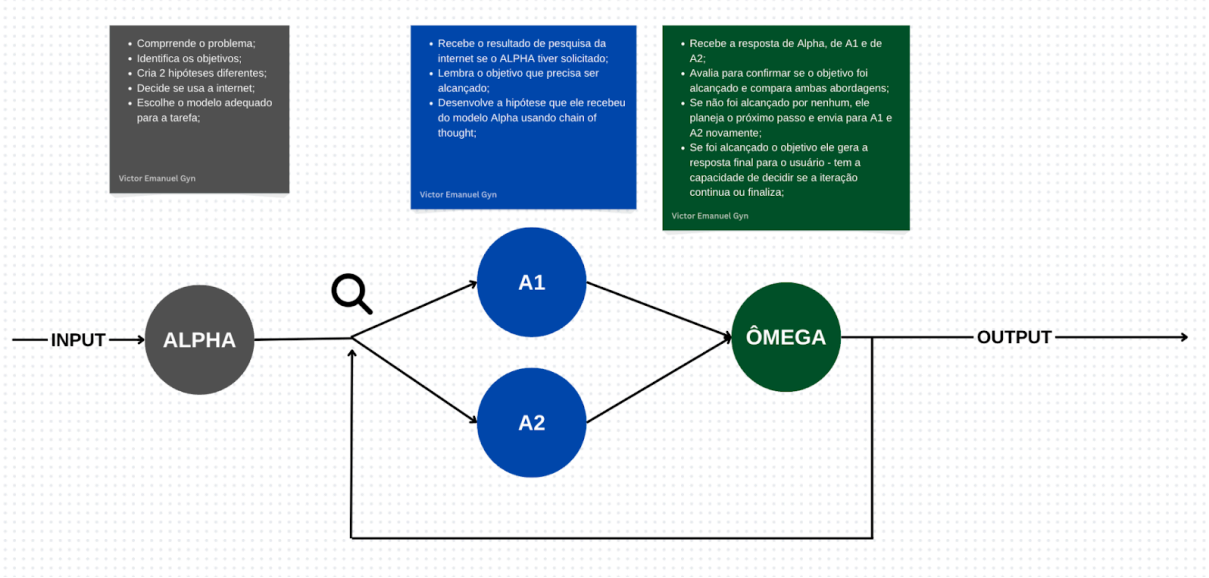
- Identificar modelos mais adequados para cada tipo de tarefa
- Planejar estrutura de prompt para cada modelo
- Iniciar implementação do sistema

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

Versão 1: Desenvolvimento de um Sistema Multiagente baseado em LLMs para Resolução Iterativa de Problemas



https://www.canva.com/design/DAGTvxr1G9Y/T7OdiTPKiGTAD2ojLM8DGg/view?utm_content=DAGTvxr1G9Y&utm_campaign=designshare&utm_medium=link&utm_source=editor

1. Introdução

Este documento detalha a arquitetura e o funcionamento de um sistema multiagente baseado em Modelos de Linguagem Grande (Large Language Models, LLMs) por meio de APIs, com foco na resolução iterativa de problemas complexos. O sistema integra três categorias de modelos — Alpha, A1/A2 e Ômega — que colaboram para compreender, formular, testar e verificar hipóteses, buscando soluções de forma eficiente e flexível. Esta abordagem assemelha-se à metodologia Mixture-of-Agents (MoA), abordada em estudos recentes, onde a colaboração entre diferentes LLMs permite construir respostas mais robustas e precisas. A arquitetura proposta distribui funções entre modelos especializados, otimizando a capacidade de compreensão e

execução em distintas áreas do problema, alinhando-se ao conceito de colaboratividade entre LLMs.

2. Componentes do Sistema

O sistema é composto por três agentes principais: o Modelo Alpha, os Modelos A1 e A2, e o Modelo Ômega. Cada um desses agentes desempenha papéis distintos no processo de resolução de problemas, interagindo iterativamente conforme as necessidades identificadas em cada ciclo. Essa abordagem compartilha similaridades com a estrutura MoA, onde múltiplos agentes colaboram em camadas sucessivas para aprimorar a qualidade das respostas.

2.1. Modelo Alpha

O Modelo Alpha é responsável pela compreensão inicial do problema e pela definição de uma estratégia para sua resolução. Suas funções principais incluem:

1. **Compreensão e Identificação de Objetivos:** Ao receber um problema, Alpha realiza uma análise inicial para entender o contexto, delimitando os objetivos principais e especificando o escopo do problema. Esta função é similar ao papel dos modelos propositores na metodologia MoA, que oferecem respostas de referência úteis aos demais modelos.
2. **Formulação de Hipóteses:** Após a identificação do problema, o modelo Alpha cria duas hipóteses distintas, cada uma representando uma abordagem específica para a solução. Essas hipóteses são então enviadas aos modelos A1 e A2.
3. **Decisão sobre a Necessidade de Pesquisa na Internet:** O modelo Alpha avalia se uma consulta externa é necessária para obter informações adicionais relevantes. Se sim, seleciona um prompt apropriado para realizar essa pesquisa.
4. **Escolha do Modelo Adequado:** Com base na natureza do problema (matemática, lógica, análise textual, etc.), Alpha escolhe o modelo mais adequado para cada tarefa a partir de uma lista de modelos de API disponíveis. Essa escolha cuidadosa é essencial, de modo similar ao critério de seleção de agentes em cada camada da metodologia MoA, que considera tanto o desempenho quanto a diversidade dos modelos.
5. **Comunicação com o Usuário:** Alpha gera uma saída inicial para o usuário, descrevendo brevemente o plano de ação e a abordagem proposta.

2.2. Modelos A1 e A2

Os Modelos A1 e A2 são responsáveis por desenvolver as hipóteses formuladas pelo Modelo Alpha. Esses agentes trabalham de forma independente e paralela, permitindo a exploração de diferentes abordagens para o mesmo problema. Suas funções principais incluem:

1. **Recebimento de Pesquisa Externa:** Se Alpha decide que uma pesquisa externa é necessária, os modelos A1 e A2 recebem e incorporam as informações obtidas para desenvolver suas respectivas soluções.
2. **Recordação dos Objetivos:** Cada modelo mantém o objetivo inicial em mente e concentra-se em alcançá-lo, usando a hipótese designada.
3. **Desenvolvimento da Hipótese via "Chain of Thought":** A1 e A2 utilizam o método "chain of thought" para desenvolver suas hipóteses, o que significa que constroem suas respostas de maneira estruturada e progressiva, garantindo uma maior profundidade de análise e exploração. Esta abordagem é similar ao processo de refinamento contínuo descrito na metodologia MoA, onde agentes sucessivos recebem saídas intermediárias e as aprimoram.
4. **Comunicação com o Usuário:** Ao final do desenvolvimento da hipótese, A1 e A2 geram uma saída breve que resume suas conclusões e a abordagem seguida.

2.3. Modelo Ômega

O Modelo Ômega é a entidade que coordena a decisão final sobre a adequação das soluções desenvolvidas pelos Modelos A1 e A2. Suas funções incluem:

1. **Avaliação das Soluções Recebidas:** Ômega recebe as respostas geradas por Alpha, A1 e A2 e as testa para confirmar se os objetivos foram efetivamente alcançados. Esse papel assemelha-se ao do modelo agregador descrito na metodologia MoA, que sintetiza as respostas de outros modelos para gerar uma saída de alta qualidade.
2. **Comparação de Abordagens e Decisão de Caminho Futuro:** Se ambas as abordagens de A1 e A2 falham em atingir os objetivos, Ômega elabora um plano para o próximo ciclo de iteração. Este plano pode incluir ajustes nos modelos A1 e A2 ou, se necessário, uma mudança drástica na estratégia, criando novas hipóteses para um novo ciclo. Essa dinâmica de ajuste é

fundamental na abordagem MoA, onde a iteração contínua leva a um refinamento progressivo das respostas.

3. **Iteração ou Finalização:** Se Ômega conclui que o objetivo foi atingido, finaliza o processo e gera a resposta final para o usuário. Caso contrário, decide se o processo deve continuar, definindo novas abordagens para A1 e A2.
4. **Comunicação com o Usuário:** Quando Ômega decide continuar a iteração, imprime uma mensagem breve explicando o que será tentado a seguir. Se a solução foi encontrada, ele imprime a resposta final para o usuário.

3. Processo Iterativo e Cooperativo

O sistema é projetado para operar de maneira iterativa e cooperativa, onde os modelos não atuam isoladamente, mas como componentes de um ciclo de melhoria contínua. O modelo Alpha desempenha o papel de estrategista inicial, os modelos A1 e A2 desenvolvem as soluções, e Ômega avalia e decide sobre o progresso. Essa interação permite que o sistema evolua em resposta a problemas complexos, refinando suas abordagens até que uma solução satisfatória seja atingida ou até que se determine que a solução não é possível. Esse processo é análogo ao descrito na metodologia MoA, onde a colaboração entre múltiplos agentes, cada um contribuindo com suas habilidades específicas, leva a uma melhoria significativa na qualidade das respostas.

4. Comparação com Mixture-of-Agents (MoA)

A metodologia Mixture-of-Agents (MoA) é uma abordagem recente que utiliza múltiplos LLMs em camadas sucessivas para gerar respostas de alta qualidade, onde agentes colaborativos refinam respostas de maneira iterativa e organizada. No entanto, o sistema aqui descrito apresenta características distintas e inovações em relação à estrutura MoA, particularmente pela implementação de uma resolução baseada em cadeia de pensamento, a iteração dinâmica e a capacidade de consulta à internet.

Uma diferença central entre o sistema proposto e a metodologia MoA é que o sistema aqui descrito aplica o método “chain of thought” (cadeia de pensamento) como base de resolução. A cada execução dos modelos A1 e A2, não se espera que esses agentes resolvam o problema integralmente; ao contrário, eles abordam apenas uma parte do problema, desenvolvendo raciocínios intermediários que serão

analisados, refinados ou expandidos nas iterações seguintes. Isso permite que os agentes avancem gradualmente, com análises passo a passo, garantindo que o entendimento e a solução final sejam construídos progressivamente, com base em etapas controladas e verificadas pelo modelo Ômega. Essa estrutura de raciocínio modular contrasta com o MoA, onde cada camada de agentes colabora para aprimorar uma resposta integral em camadas definidas, sem fracionar a resolução do problema em pequenas partes consecutivas.

Outra inovação significativa do sistema é a possibilidade de consulta à internet, o que permite que os modelos acessem informações externas conforme a necessidade identificada pelo modelo Alpha. A capacidade de pesquisa em fontes externas amplia a abrangência e a precisão das respostas ao agregar dados contextuais ou especializados, adaptando-se a problemas que exigem atualização de informações em tempo real ou dados específicos. No MoA, por outro lado, as camadas de agentes operam exclusivamente com informações internas, limitadas ao conhecimento já disponível nos modelos de linguagem.

Além disso, o sistema proposto adota uma iteração dinâmica e adaptativa, onde o próprio modelo Ômega decide sobre a continuidade ou finalização do processo com base nos resultados obtidos. Em outras palavras, o número de iterações não é predefinido, mas determinado pelo próprio sistema, que avalia constantemente se os objetivos propostos foram atingidos. Esse processo de iteração controlado pelo Ômega proporciona flexibilidade e ajusta o esforço de resolução às especificidades de cada problema. Em contraste, a estrutura MoA é projetada com um número de camadas e iterações pré-estabelecido, independentemente da complexidade do problema ou das conclusões atingidas até um determinado ponto.

Essas características tornam o sistema uma extensão do conceito MoA, adaptando-o para uma abordagem de resolução mais modular, flexível e informada. Assim, a combinação de raciocínio em cadeia, iteração auto-regulada e acesso a informações externas fortalece a capacidade do sistema em lidar com problemas complexos e dinâmicos, oferecendo uma solução mais robusta e alinhada às necessidades de adaptação e precisão do contexto real.

5. Importância do "Chain of Thought" no Sistema

O método "chain of thought" desempenha um papel central na estrutura do sistema multiagente, servindo como ferramenta essencial para a construção de soluções completas e coerentes. Em particular, o uso do raciocínio usando cadeia de pensamento permite que cada agente — especialmente os modelos A1 e A2 — desenvolva raciocínios intermediários de forma organizada e sequencial. A técnica refere-se à prática de gerar uma série de passos intermediários que conduzem à resposta final, o que se mostra útil em tarefas que exigem múltiplas etapas de análise ou resolução, como problemas matemáticos e raciocínio de senso comum.

No sistema proposto, esta técnica é usada por A1 e A2 para estruturar hipóteses, facilitando a avaliação posterior pelo modelo Ômega e garantindo clareza e precisão no processo de resolução de problemas. Essa técnica alinha-se à natureza iterativa e colaborativa da arquitetura multiagente descrita, permitindo que cada ciclo de iteração seja mais preciso e orientado ao objetivo final.

6. Importância do In-Context Few-Shot Learning no Sistema

Outra técnica central para o desempenho do sistema é o "in-context few-shot

learning via prompting", que permite aos modelos aprender a partir de poucos exemplos apresentados diretamente no prompt. Esse método é eficaz em sistemas que requerem generalização rápida e adaptação a tarefas variadas. O "in-context few-shot learning" consiste em fornecer exemplos específicos no próprio prompt, permitindo ao modelo entender padrões e contextos desejados, mesmo que não tenha sido treinado especificamente para a tarefa.

Este método permite que os modelos, especialmente A1 e A2, desenvolvam abordagens eficazes para uma ampla gama de problemas, usando poucos exemplos como orientação, proporcionando maior flexibilidade e capacidade de generalização ao sistema.

7. Considerações

A arquitetura proposta proporciona um sistema robusto e flexível, capaz de resolver problemas complexos de forma iterativa e colaborativa. O uso de modelos com funções definidas cria uma dinâmica em que a análise, a criatividade e a avaliação crítica se complementam. O modelo Ômega, com a capacidade de decidir entre continuar ou finalizar o processo, assegura que o sistema não fique preso em ciclos

de tentativas infrutíferas, enquanto os modelos Alpha, A1 e A2 trabalham para explorar abordagens diversas.

Essa abordagem alinha-se com a metodologia Mixture-of-Agents, que demonstrou uma melhoria significativa na qualidade das respostas ao utilizar múltiplos agentes colaborativos. Além disso, a estrutura permite expansão e aprimoramento, incluindo mais camadas de modelos e integração de diferentes tecnologias para aumentar a capacidade de resolução de problemas.

APÊNDICE 5
Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 31 de out. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

Victor Emanuel da Silva Monteiro

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Para esse Gate, foi implementado o sistema de resolução de problemas de forma iterativa, usando base em cadeia de pensamento, com múltiplos LLMs que opera em quatro etapas distintas e colaborativas:

1. **Entrada e Análise do Problema**: O sistema começa com a entrada de um problema. Em seguida, um módulo de análise inicial (Alpha) compreende o problema, define objetivos claros e gera duas hipóteses para explorar possíveis soluções.
2. **Execução de Hipóteses em Paralelo**: As hipóteses são encaminhadas para dois módulos de execução (A1 e A2) que testam abordagens diferentes em paralelo. Esses módulos trabalham de forma independente, detalhando as etapas e documentando os resultados e raciocínios para cada hipótese.
3. **Avaliação dos Resultados**: Após a execução, um módulo de avaliação (Ômega) compara as respostas de ambos os executores. Esse módulo analisa se os objetivos foram cumpridos, identifica eventuais falhas ou pontos de melhoria e decide os próximos passos, podendo sugerir uma nova iteração.
4. **Processo Iterativo até a Conclusão**: Caso nenhuma das hipóteses inicial atinja completamente os objetivos, o sistema se ajusta e reitera o processo com base nas conclusões da avaliação. Essa iteração contínua permite que o sistema refine as soluções até encontrar uma resposta satisfatória.
5. **Geração da Solução Final**: Quando o objetivo é alcançado, o sistema consolida os resultados e gera uma resposta final detalhada, entregando a solução ao usuário com transparência sobre o processo de raciocínio.

Código desenvolvido nesse gate:
Sistema baseado em cadeia de pensamento com LLMs para Resolução Iterativa de Problemas.ipynb

Proposta completa: Versão 1: Desenvolvimento de um Sistema Multiagente baseado em LLMs para …

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

- **Implementar pesquisa na internet**
- **Implementar múltiplos modelos com seleção automatizada**

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

Sistema baseado em cadeia de pensamento com LLMs para Resolução Iterativa de Problemas.ipynb

Visão geral do sistema atual

Screenshot 2024-10-31 at 13.14.48.png

- **Alpha (primeiro agente)**
  - **No código:** Representado pela classe AlphaAgent.
  - **Função:** Recebe o problema como entrada (input) e:
    - Compreende o problema.
    - Identifica objetivos específicos.
    - Cria duas hipóteses diferentes com o uso de pensamento em cadeia (chain of thought).
  - **Processo:** Executa a análise inicial e retorna um objeto AlphaResponse com as hipóteses e objetivos definidos.
- **A1 e A2 (agentes executores)**
  - **No código:** Representados pela classe ExecutorAgent, com instâncias a1 e a2.
  - **Função:** Recebem uma das hipóteses geradas por Alpha e:
    - Lembram o objetivo a ser alcançado.
    - Desenvolvem e testam a hipótese recebida através de passos detalhados, usando chain of thought.
  - **Processo:** Cada agente executa uma série de etapas para testar sua hipótese e gera um objeto ExecutorResponse com o resultado de cada passo, incluindo raciocínio e perguntas feitas.
- **Omega (agente avaliador)**

- **No código:** Representado pela classe OmegaAgent.
- **Função:** Recebe as respostas de Alpha, A1 e A2 e:
  - Avalia se o objetivo foi alcançado e compara as abordagens de A1 e A2.
  - Decide se a iteração deve continuar ou se o processo está concluído.
  - Planeja o próximo passo ou gera uma resposta final para o usuário se o objetivo for alcançado.
- **Processo:** Omega avalia os resultados em um objeto OmegaResponse, onde documenta o raciocínio, a conclusão, e decide os próximos passos.

· **Ciclo de Iteração (do Alpha ao Ômega)**

- **No código:** Implementado na classe MultiAgentSystem.
- **Função:** Coordena as iterações entre Alpha, A1, A2 e Omega até que o problema seja resolvido ou o número máximo de iterações seja alcançado.
- **Processo:** Cada iteração envolve Alpha criando hipóteses, A1 e A2 testando-as, e Omega avaliando e decidindo a próxima ação. Se necessário, o ciclo continua com Omega enviando novas instruções a A1 e A2.

Implementação do código

```
# Instalação de pacotes
!pip install mistralai
%pip install -qU langchain-openai

# Importações de pacotes e módulos
import json
import time
import os
```

```
from typing import List, Dict, Any, Optional, Tuple
from dataclasses import dataclass
from pydantic import BaseModel, Field
from rich import print as rprint
from rich.console import Console
from rich.panel import Panel
from rich.table import Table
from mistralai import Mistral
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
```

```
Collecting mistralai
  Downloading mistralai-1.1.0-py3-none-any.whl.metadata (23 kB)
Requirement already satisfied: eval-type-backport<0.3.0,>=0.2.0 in /usr/local/lib/python3.10/dist-packages (from mistralai) (0.2.0)
Requirement already satisfied: httpx<0.28.0,>=0.27.0 in /usr/local/lib/python3.10/dist-packages (from mistralai) (0.27.2)
Collecting jsonpath-python<2.0.0,>=1.0.6 (from mistralai)
  Downloading jsonpath_python-1.0.6-py3-none-any.whl.metadata (12 kB)
Requirement already satisfied: pydantic<3.0.0,>=2.9.0 in /usr/local/lib/python3.10/dist-packages (from mistralai) (2.9.2)
Requirement already satisfied: python-dateutil==2.8.2 in /usr/local/lib/python3.10/dist-packages (from mistralai) (2.8.2)
Collecting typing-inspect<0.10.0,>=0.9.0 (from mistralai)
  Downloading typing_inspect-0.9.0-py3-none-any.whl.metadata (1.5 kB)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil==2.8.2->mistralai) (1.16.0)
Requirement already satisfied: anyio in /usr/local/lib/python3.10/dist-packages (from httpx<0.28.0,>=0.27.0->mistralai) (3.7.1)
Requirement already satisfied: certifi in /usr/local/lib/python3.10/dist-packages (from httpx<0.28.0,>=0.27.0->mistralai) (2024.8.30)
Requirement already satisfied: httpcore==1.* in /usr/local/lib/python3.10/dist-packages (from
```

httpx<0.28.0,>=0.27.0->mistralai) (1.0.6)
Requirement already satisfied: idna in /usr/local/lib/python3.10/dist-packages (from httpx<0.28.0,>=0.27.0->mistralai) (3.10)
Requirement already satisfied: sniffio in /usr/local/lib/python3.10/dist-packages (from httpx<0.28.0,>=0.27.0->mistralai) (1.3.1)
Requirement already satisfied: h11<0.15,>=0.13 in /usr/local/lib/python3.10/dist-packages (from httpcore==1.*->httpx<0.28.0,>=0.27.0->mistralai) (0.14.0)
Requirement already satisfied: annotated-types>=0.6.0 in /usr/local/lib/python3.10/dist-packages (from pydantic<3.0.0,>=2.9.0->mistralai) (0.7.0)
Requirement already satisfied: pydantic-core==2.23.4 in /usr/local/lib/python3.10/dist-packages (from pydantic<3.0.0,>=2.9.0->mistralai) (2.23.4)
Requirement already satisfied: typing-extensions>=4.6.1 in /usr/local/lib/python3.10/dist-packages (from pydantic<3.0.0,>=2.9.0->mistralai) (4.12.2)
Collecting mypy-extensions>=0.3.0 (from typing-inspect<0.10.0,>=0.9.0->mistralai)
  Downloading mypy_extensions-1.0.0-py3-none-any.whl.metadata (1.1 kB)
Requirement already satisfied: exceptiongroup in /usr/local/lib/python3.10/dist-packages (from anyio->httpx<0.28.0,>=0.27.0->mistralai) (1.2.2)
Downloading mistralai-1.1.0-py3-none-any.whl (229 kB)

229.7/229.7 kB 12.6 MB/s eta 0:00:00
ypy_extensions-1.0.0-py3-none-any.whl (4.7 kB)
Installing collected packages: mypy-extensions, jsonpath-python, typing-inspect, mistralai
Successfully installed jsonpath-python-1.0.6 mistralai-1.1.0 mypy-extensions-1.0.0 typing-inspect-0.9.0

50.4/50.4 kB 2.8 MB/s eta 0:00:00

1.2/1.2 MB 31.3 MB/s eta 0:00:00

```
console = Console()

# Estrutura para processos de raciocínio
```

```
class Step(BaseModel):
    step_number: int = Field(description="O número da etapa em sequência")
    description: str = Field(description="Descrição do que fazer nesta etapa")
    reasoning: str = Field(description="Raciocínio por trás desta etapa")

class Hypothesis(BaseModel):
    hypothesis: str = Field(description="Descrição da hipótese")
    steps: List[Step] = Field(description="Etapas para testar esta hipótese")

class AlphaResponse(BaseModel):
    analysis: str = Field(description="Análise detalhada do problema")
    objectives: str = Field(description="Declaração clara dos objetivos")
    hypotheses: List[Hypothesis] = Field(description="Lista de hipóteses a serem testadas")

class ExecutorStep(BaseModel):
    step_number: int = Field(description="O número da etapa em sequência")
    action: str = Field(description="Ação realizada nesta etapa")
    result: str = Field(description="Resultado da ação")
    reasoning: str = Field(description="Raciocínio por trás da ação e resultado")
    questions_asked: List[str] = Field(description="Perguntas feitas durante esta etapa")
    answers_found: List[str] = Field(description="Respostas para as perguntas feitas")

class ExecutorResponse(BaseModel):
    summary: str = Field(description="Resumo breve da tentativa de execução")
    reasoning_process: str = Field(description="Processo de raciocínio detalhado, etapa por etapa")
    detailed_steps: List[ExecutorStep] = Field(description="Etapas detalhadas da execução")
    conclusion: str = Field(description="Conclusão da tentativa")
    next_actions: str = Field(description="Próximas etapas sugeridas")

class OmegaResponse(BaseModel):
```

```
    evaluation: str = Field(description="Avaliação do progresso atual")
    reasoning_process: str = Field(description="Processo de raciocínio detalhado, etapa por etapa")
    objective_achieved: bool = Field(description="Se o objetivo foi alcançado")
    final_response: Optional[str] = Field(description="Solução final se o objetivo foi alcançado")
    next_step: Optional[str] = Field(description="Próximas etapas se o objetivo não foi alcançado")
    reasoning: str = Field(description="Raciocínio detalhado para a avaliação")
    hypothesis_to_continue: Optional[Hypothesis] = Field(description="Hipótese para continuar")

class Agent:
    def __init__(self, agent_id: str, llm: ChatOpenAI):
    self.agent_id = agent_id
    self.llm = llm

    def get_system_prompt(self) -> str:
    return f"You are {self.agent_id}."

class AlphaAgent(Agent):
    def __init__(self, llm: ChatOpenAI):
    super().__init__("Alpha", llm)
    self.structured_llm = llm.with_structured_output(AlphaResponse)

    def analyze(self, problem: str) -> Optional[AlphaResponse]:
    prompt = ChatPromptTemplate.from_messages([
    ("system", self.get_system_prompt() + """
You analyze problems and develop hypotheses to solve them using chain of thought and thinking step by step. Your tasks are:
- Comprehend the problem thoroughly
- Identify clear and specific objectives
- Create two distinct hypotheses with detailed reasoning
- For each hypothesis, outline one detailed first step"""),
```

```
        ("human", "Problem: {problem}")
        ])

        chain = prompt | self.structured_llm
        try:
        return chain.invoke({"problem": problem})
        except Exception as e:
        console.print(f"[red]Erro na análise de Alpha: {e}[/red]")
        return None

class ExecutorAgent(Agent):
        def __init__(self, agent_id: str, llm: ChatOpenAI):
        super().__init__(agent_id, llm)
        self.structured_llm = llm.with_structured_output(ExecutorResponse)

        def execute(self, objective: str, hypothesis: Hypothesis) -> Optional[ExecutorResponse]:
        prompt = ChatPromptTemplate.from_messages([
        ("system", self.get_system_prompt() + """
You execute steps to test hypotheses using chain of thought STEP BY STEP. Your tasks are:
- Follow the given objective and step by step meticulously
- Provide detailed reasoning for each step by asking yourself questions and answering them
- Document your thought process by:
  1. Asking yourself relevant questions about the current step
  2. Exploring possible approaches and their implications
  3. Explaining why you chose a particular approach
  4. Documenting any assumptions or constraints
- Document and write results clearly
- Whenever you have found part of the solution, write directly to the user what the partial or total solution you found was, in addition to explaining how you arrived at it. Write the partial or full answer
- Suggest next actions based on results

For each step, structure your reasoning as:
```

```
1. What am I trying to achieve in this step?
2. What information do I need?
3. What approaches could work here?
4. Why is my chosen approach the best?
5. What potential obstacles might I face?
6. How can I verify my results?"""),
        ("human", """Objective: {objective}
Hypothesis: {hypothesis}
Steps: {steps}""")
        ])

        steps_text = "\n".join([
        f"{step.step_number}. {step.description}\n   Reasoning: {step.reasoning}"
        for step in hypothesis.steps
        ])

        chain = prompt | self.structured_llm
        try:
        return chain.invoke({
        "objective": objective,
        "hypothesis": hypothesis.hypothesis,
        "steps": steps_text
        })
        except Exception as e:
        console.print(f"[red]Erro na execução de {self.agent_id}: {e}[/red]")
        return None

class OmegaAgent(Agent):
        def __init__(self, llm: ChatOpenAI):
        super().__init__("Omega", llm)
        self.structured_llm = llm.with_structured_output(OmegaResponse)
```

```
    def evaluate(self, history: str, problem: str, a1_response: ExecutorResponse, a2_response: ExecutorResponse) -> Optional[OmegaResponse]:
        prompt = ChatPromptTemplate.from_messages([
        ("system", self.get_system_prompt() + """
```

You evaluate the execution results and determine the next steps. Your tasks are:

- Review the responses from the executors through a detailed reasoning process:
  1. What specific results did each executor achieve?
  2. How do these results align with the original objective?
  3. What are the strengths and weaknesses of each approach?
  4. Are there any inconsistencies or gaps in the reasoning?
  5. What evidence supports each conclusion?
- Assess if the objectives were achieved 100% according to the original problem
- Provide a final response that answers the objective or recommends the next steps
- Critically analyze each executor's approach and results; ensure they met the objective by reviewing what they did. Do not trust that it is resolved just because they claim to have reached the correct answer; review and look for possible errors yourself
- If an error is identified, you have the authority to request a step to be repeated, think of another alternative and decide first next step, or continue the attempts
- Decide whether to continue, modify the approach, or conclude the problem-solving process
- If the models can
- If not successful, generate a new hypothesis or strategy

Important:

- If you believe only one model is right you can ask both models to repeat one task to make sure both will agree and get the answer right
- For theoretical questions, strict rigor is not necessary.
- Document your reasoning process by asking yourself key questions:
  1. What specific criteria am I using to evaluate success?
  2. How reliable are the results from each executor?
  3. What potential biases might affect my evaluation?
  4. What evidence would convince me to change my evaluation?

```
- If the models are too repetitive, then wrap up and inform the user of any intermediate conclusions reached, but that the expected result was not achieved.
- Provide clear reasoning STEP BY STEP for your decision
- If the problem is solved, generate a comprehensive final response"""),
            ("human", """Original Problem: {problem}
History: {history}
A1 Response: {a1_response}
A2 Response: {a2_response}""")
        ])

        chain = prompt | self.structured_llm
        try:
        return chain.invoke({
        "problem": problem,
        "history": history,
        "a1_response": a1_response.model_dump_json(),
        "a2_response": a2_response.model_dump_json()
        })
        except Exception as e:
        console.print(f"[red]Erro na avaliação de Omega: {e}[/red]")
        return None


class MultiAgentSystem:
        def __init__(self, api_key: str, model: str):
        os.environ["OPENAI_API_KEY"] = api_key
        self.llm = ChatOpenAI(model=model)
        self.history: List[Dict[str, Any]] = []

        self.alpha = AlphaAgent(self.llm)
        self.a1 = ExecutorAgent("A1", self.llm)
```

```
self.a2 = ExecutorAgent("A2", self.llm)
self.omega = OmegaAgent(self.llm)

self.initial_hypotheses: Optional[List[Hypothesis]] = None
self.original_problem: Optional[str] = None

def build_history_string(self) -> str:
history = ""
for i, entry in enumerate(self.history, start=1):
history += f"Iteration {i}:\n"
history += f"A1: {entry['a1'].summary}\n"
history += f"A2: {entry['a2'].summary}\n"
history += f"Omega Evaluation: {entry['omega'].evaluation}\n\n"
return history

def display_iteration_results(self, iteration: int, a1_response: ExecutorResponse,
                              a2_response: ExecutorResponse, omega_response: OmegaResponse):
table = Table(title=f"Resultados da Iteração {iteration}")
table.add_column("Agente", style="cyan")
table.add_column("Resultado", style="magenta")

# Resultados de A1
table.add_row("Resumo de A1", a1_response.summary)
table.add_row("Processo de Raciocínio de A1", a1_response.reasoning_process)
table.add_row("Conclusão de A1", a1_response.conclusion)

# Resultados de A2
table.add_row("Resumo de A2", a2_response.summary)
table.add_row("Processo de Raciocínio de A2", a2_response.reasoning_process)
table.add_row("Conclusão de A2", a2_response.conclusion)
```

```
# Resultados de Omega
table.add_row("Avaliação de Omega", omega_response.evaluation)
table.add_row("Processo de Raciocínio de Omega", omega_response.reasoning_process)

console.print(table)

if omega_response.next_step:
console.print(f"[yellow]Próxima Etapa: {omega_response.next_step}[/yellow]")

def run_iteration(self, hypotheses: List[Hypothesis], iteration: int) -> bool:
console.print(f"\n[bold cyan]--- Iteração {iteration} ---[/bold cyan]")

# Executar ambas as hipóteses
a1_response = self.a1.execute(
self.original_problem,
hypotheses[0]
)

a2_response = self.a2.execute(
self.original_problem,
hypotheses[1]
)

if not a1_response or not a2_response:
console.print("[red]Falha na execução para um ou ambos os agentes.[/red]")
return False

# Avaliar resultados
omega_response = self.omega.evaluate(
self.build_history_string(),
self.original_problem,
```

```
a1_response,
a2_response
)

if not omega_response:
console.print("[red]Falha na avaliação de Omega.[/red]")
return False

# Exibir resultados da iteração
self.display_iteration_results(iteration, a1_response, a2_response, omega_response)

# Verificar se o objetivo foi alcançado
if omega_response.objective_achieved:
console.print(Panel(
omega_response.final_response or "Solução encontrada, mas sem resposta final fornecida.",
title="[bold green]Solução Final[/bold green]",
border_style="green"
))
return True

# Adicionar ao histórico
self.history.append({
"iteration": iteration,
"a1": a1_response,
"a2": a2_response,
"omega": omega_response
})

return False

def solve_problem(self, problem: str, max_iterations: int = 10) -> None:
```

```
console.print(f"\n[bold green]Iniciando a resolução do problema:[/bold green] {problem}\n")

# Armazenar problema original
self.original_problem = problem

# Obter análise inicial e hipóteses de Alpha (apenas uma vez)
alpha_response = self.alpha.analyze(problem)
if not alpha_response:
console.print("[red]Erro: Alpha não gerou hipóteses.[/red]")
return

# Armazenar hipóteses iniciais
self.initial_hypotheses = alpha_response.hypotheses

console.print(f"[yellow]Análise Inicial de Alpha:[/yellow]\n{alpha_response.analysis}")
console.print(f"[yellow]Objetivos Iniciais:[/yellow]\n{alpha_response.objectives}")

# Executar iterações
iteration = 1
current_hypotheses = self.initial_hypotheses

while iteration <= max_iterations:
if self.run_iteration(current_hypotheses, iteration):
console.print("\n[bold green]Problema resolvido com sucesso![/bold green]")
break

# Se não resolvido, Omega decide os próximos passos
last_omega_response = self.history[-1]['omega']
if last_omega_response.hypothesis_to_continue:
# Se Omega sugere uma nova hipótese, use-a
current_hypotheses = [last_omega_response.hypothesis_to_continue,
```

```
                    self.initial_hypotheses[1]]

        iteration += 1
        time.sleep(1)

        if iteration > max_iterations:
        console.print("\n[bold red]Número máximo de iterações atingido sem encontrar uma solução.[/bold red]")

def main():
        from getpass import getpass

        API_KEY = getpass("Digite sua chave API do OpenAI: ").strip()
        if not API_KEY:
        console.print("[red]Erro: A chave API do OpenAI é necessária.[/red]")
        return

        MODEL = "gpt-4o-mini"  # Usando o modelo mais recente GPT-4o-mini
        system = MultiAgentSystem(API_KEY, MODEL)

        problem = """ Explique como funciona um paradoxo"""
        system.solve_problem(problem)

if __name__ == "__main__":
        main()
```

Digite sua chave API do OpenAI: ··········

Iniciando a resolução do problema: Explique como funciona um paradoxo

Análise Inicial de Alpha:
Um paradoxo é uma declaração ou proposição que parece contradizer-se ou ir contra a intuição, mas que pode, de
fato, conter uma verdade subjacente. Os paradoxos são frequentemente usados na filosofia, matemática e lógica para
explorar conceitos complexos e desafiadores. Eles geralmente revelam uma falha em nosso entendimento ou nas
premissas que usamos para chegar a uma conclusão.

Objetivos Iniciais:
Explicar o conceito de paradoxo de forma clara e acessível, incluindo exemplos e aplicações em diferentes campos.

--- Iteração 1 ---

Resultados da Iteração 1

Agente	Resultado
Resumo de A1	Investigação de exemplos clássicos de paradoxos, como o Paradoxo de Zenão e o Paradoxo do Mentiroso.
Processo de Raciocínio de A1	1. O que estou tentando alcançar neste passo? Quero identificar alguns exemplos clássicos de paradoxos que demonstram como eles desafiam suposições e complexificam conceitos. 2. Que informações eu preciso? Preciso de uma

		compreensão básica dos paradoxos, suas definições e exemplos clássicos, como
		o Paradoxo de Zenão e o Paradoxo do Mentiroso. 3. Que abordagens poderiam
		funcionar aqui? Poderia pesquisar em livros de filosofia, artigos acadêmicos
		ou fontes online confiáveis sobre lógica e paradoxos. 4. Por que a abordagem
	escolhida é a melhor? A pesquisa em fontes confiáveis me permitirá acessar
		uma explicação abrangente e bem fundamentada sobre os paradoxos. 5. Que
		obstáculos potenciais posso enfrentar? As informações podem ser complexas ou
		confusas, e posso encontrar interpretações diferentes dos paradoxos. 6. Como
		posso verificar meus resultados? Posso comparar as informações obtidas com
		várias fontes para garantir a precisão e a consistência.
Conclusão de A1	A investigação de exemplos clássicos de paradoxos, como o Paradoxo de Zenão e
		o Paradoxo do Mentiroso, ilustra como os paradoxos desafiam suposições e
		revelam a complexidade de conceitos como movimento e verdade.
Resumo de A2	Análise das premissas de um paradoxo clássico para entender suas
		inconsistências.
Processo de Raciocínio de A2	Neste passo, vou focar em um paradoxo específico, o paradoxo do mentiroso,
		que diz: "Esta frase é falsa." Vamos analisar as premissas envolvidas. O

		objetivo é entender se há alguma falha na lógica que nos leva ao paradoxo. Ao identificar essas falhas, poderemos reavaliar a situação e ver se conseguimos encontrar uma solução ou explicação para o paradoxo.
Conclusão de A2	A análise das premissas do paradoxo do mentiroso revela que a autorreferência cria uma situação onde as definições convencionais de verdade e falsidade não se aplicam. A teoria de Tarski oferece uma maneira de lidar com essa complexidade, sugerindo que tais afirmações não podem ser verdadeiras ou falsas na mesma estrutura.
Avaliação de Omega	Os executores A1 e A2 abordaram o conceito de paradoxo de maneiras complementares. A1 focou em exemplos clássicos de paradoxos, enquanto A2 analisou especificamente o paradoxo do mentiroso, explorando suas premissas e possíveis soluções.
Processo de Raciocínio de Omega	1. O que estou tentando alcançar neste passo? Avaliar se as investigações de A1 e A2 explicam adequadamente como funciona um paradoxo. 2. Que informações são necessárias? Precisamos garantir que ambos os executores apresentem uma compreensão clara do conceito de paradoxo e suas implicações. 3. Quais abordagens funcionaram? A abordagem de A1 trouxe exemplos práticos,

enquanto |
| A2 aprofundou-se na análise lógica. 4. Quais são as forças e fraquezas? A1 |
| trouxe uma visão abrangente com exemplos, mas não aprofundou a lógica |
| subjacente. A2 fez uma análise lógica sólida, mas se concentrou em um único |
| paradoxo. 5. Existem inconsistências? Ambos os executores não contradizem um |
| ao outro, mas suas abordagens diferem em escopo. 6. Que evidências suportam |
| as conclusões? A1 usou exemplos clássicos, enquanto A2 se baseou em lógica |
| formal e teorias filosóficas. |

Próxima Etapa: Solicitar que A1 e A2 produzam uma síntese conjunta que combine exemplos clássicos de paradoxos com
uma análise lógica, garantindo que a explicação final seja abrangente e clara sobre como funcionam os paradoxos.

--- Iteração 2 ---

Resultados da Iteração 2

| Agente | Resultado |

Resumo de A1	Colaboração entre A1 e A2 para explicar paradoxos.
Processo de Raciocínio de A1	Neste passo, o objetivo é unir os conhecimentos de A1 e A2 para criar uma apresentação sobre paradoxos. A1 traz exemplos práticos e A2 fornece uma análise lógica. Essa sinergia deve resultar em uma explicação mais robusta do conceito de paradoxo.
Conclusão de A1	A colaboração entre A1 e A2 resultou em uma apresentação abrangente que combina exemplos e análises lógicas de paradoxos, facilitando uma compreensão mais robusta do conceito.
Resumo de A2	Analisei as premissas do paradoxo do mentiroso e identifiquei inconsistências que ajudam a esclarecer sua natureza.
Processo de Raciocínio de A2	O paradoxo do mentiroso afirma que "esta frase é falsa". A análise das premissas revela que se a frase é verdadeira, então é falsa, e vice-versa. Isso indica uma inconsistência na própria estrutura da afirmação. Ao reavaliar a premissa de que a frase pode ser verdadeira ou falsa, podemos entender que a classificação de verdade não se aplica da mesma maneira a essa frase, levando à sua natureza paradoxal.
Conclusão de A2	A análise das premissas do paradoxo do mentiroso revela inconsistências que indicam a necessidade de uma nova abordagem para entender a verdade.

	Paradoxos podem, de fato, ser resolvidos através da reavaliação de suas premissas subjacentes.	
Avaliação de Omega	Ambas as respostas de A1 e A2 se complementaram bem, fornecendo uma visão abrangente sobre o conceito de paradoxo. A1 trouxe exemplos práticos e uma definição clara, enquanto A2 focou em uma análise mais profunda de um paradoxo específico, o paradoxo do mentiroso, revelando suas inconsistências.	
Processo de Raciocínio de Omega	1. A1 definiu um paradoxo como uma contradição aparente e reuniu exemplos clássicos, como o Paradoxo de Zenão e o Paradoxo do Mentiroso. Isso ajuda a entender o conceito de maneira geral. 2. A2 aprofundou-se no paradoxo do mentiroso, analisando suas premissas e identificando inconsistências. Essa análise fornece uma compreensão mais refinada de como os paradoxos desafiam a lógica convencional. 3. A colaboração entre A1 e A2 resultou em uma apresentação robusta e multifacetada que explora tanto a definição quanto a lógica dos paradoxos, sendo que ambas as abordagens se sustentam mutuamente.	

Solução Final

Paradoxos são afirmações ou situações que parecem contradizer-se, mas que revelam complexidades na lógica e na linguagem. Exemplos clássicos, como o Paradoxo do Mentiroso, mostram como a auto-referência pode gerar inconsistências. Para entender paradoxos, é crucial avaliar suas premissas e considerar novas abordagens lógicas, como a lógica paraconsistente.

Problema resolvido com sucesso!

Próximos passos

Múltiplos modelos a serem implementados, para o Alpha e o Ômega escolherem os executores conforme a necessidade:

- Ministral 3B: Conhecido como o melhor modelo edge, com disponibilidade de peso, permitindo alto desempenho em operações locais. Conta com um limite de 128 mil tokens e é acessível pela API “ministral-3b-latest” na versão 24.10.
- Ministral 8B: Também com disponibilidade de peso e sob a Mistral Research License, este modelo é reconhecido pelo seu desempenho elevado em relação ao custo. Possui um limite de 128 mil tokens e pode ser acessado pela API “ministral-8b-latest”, versão 24.10.
- Mistral Large: Este é o modelo de alta complexidade da Mistral, ideal para tarefas que exigem raciocínio avançado. A versão mais recente (v2) foi lançada

em julho de 2024. Com um limite de 128 mil tokens, ele é acessível via API "mistral-large-latest", versão 24.07.

- Mistral Small: Um modelo de menor porte, otimizado para uso empresarial. Sua última versão (v2) foi lançada em setembro de 2024, oferecendo um limite de 32 mil tokens. A API correspondente é "mistral-small-latest", versão 24.09.
- Codestral: Lançado em maio de 2024 e sob a Mistral Non-Production License, é um modelo especializado em linguagem de programação, com limite de 32 mil tokens, acessível pela API "codestral-latest", versão 24.05.

Implementar sistema de pesquisa na internet para ter acesso a informações atualizadas

- SepAPI https://serpapi.com/

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 13 de nov. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

Victor Emanuel da Silva Monteiro

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Para esta Entrega semanal, os conceitos e termos do "Sistema Multiagente baseado em LLMs com respostas iterativas" foram reformulados para seguir os padrões conhecidos na literatura, especialmente com base nos conceitos já conhecidos e apresentados nos papers:

- Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.pdf
- Mixture-of-Agents Enhances Large Language Model Capabilities.pdf

O sistema é composto por três agentes principais que colaboram entre si para responder às solicitações dos usuários:

- **Propositor Alpha**: Realiza a análise inicial da solicitação do usuário, define os objetivos e formula propostas de raciocínio para serem exploradas a fim de gerar a resposta.
- **Executores E1 e E2**: Executam as propostas de raciocínio desenvolvidas para encontrar possíveis respostas a fim de atender a solicitação do usuário. Cada Executor realiza o processo, de forma independente, usando a técnica de cadeia de pensamento (*Chain of Thought*) para construir respostas estruturadas e detalhadas seguindo os objetivos definidos pelo Propositor Alpha.
- **Agregador Ômega**: Avalia, compara e sintetiza as respostas dos executores, com autonomia para decidir sobre a continuidade ou a finalização do processo com base nos objetivos definidos pelo Propositor.

Versão 1.2 - Desenvolvimento de um Sistema Multiagente baseado em LLMs para Resolução Iterativ…

Além disso, foram implementadas melhorias no sistema, incluindo:

- **Nova ferramenta de pesquisa**: possibilita o acesso direto às buscas no Google sempre que o modelo identificar necessidade, garantindo respostas mais precisas e fundamentadas.
- **Novos exemplos** de raciocínio em cadeia, para demonstrar ao LLM como responder a solicitação passo a passo

Versão 1.2-Sistema baseado em cadeia de pensamento com LLMs para Resolução Iterativa de Probl…

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

Implementar a escolha automática de qual modelo LLM será usado para resolver a solicitação
- com base nos pontos fortes de cada modelo
- com base no custo

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

Versão 1.2: Sistema Multiagente baseado em LLMs com respostas iterativas

1. Introdução

Este documento detalha a arquitetura e o funcionamento de um sistema multiagente baseado em Modelos de Linguagem Grande (Large Language Models, LLMs) por meio de APIs, com foco na resolução iterativa de solicitações complexas. O sistema integra três categorias de agentes — **Propositor Alpha**, **Executores E1 e E2**, e **Agregador Ômega** — que colaboram para compreender, formular, explorar e avaliar respostas de forma eficiente e flexível. Essa abordagem se inspira na metodologia *Mixture-of-Agents* (MoA), onde a colaboração entre diferentes LLMs permite construir respostas mais robustas e precisas. A arquitetura proposta distribui funções especializadas entre os agentes, otimizando a capacidade de compreensão e execução em áreas distintas conforme a necessidade, alinhando-se ao conceito de colaboratividade entre LLMs.

2. Componentes do Sistema

O sistema é composto por três agentes principais: o **Propositor Alpha**, os **Executores E1 e E2**, e o **Agregador Ômega**. Cada um desses agentes desempenha papéis distintos no processo de resolução de problemas, interagindo iterativamente conforme as necessidades identificadas em cada ciclo. Essa abordagem compartilha similaridades com a estrutura MoA, onde múltiplos agentes colaboram em camadas sucessivas para aprimorar a qualidade das respostas.

2.1. Propositor Alpha

O **Propositor Alpha** é responsável pela compreensão inicial da solicitação e pela definição de uma estratégia para sua resposta. Suas funções principais incluem:

- **Compreensão e Identificação de Objetivos**: Ao receber uma solicitação, Alpha realiza uma análise inicial para entender o contexto, delimitando os objetivos principais e especificando o escopo da resposta. Esta função é similar ao papel dos

modelos propositores na metodologia MoA, que oferecem respostas de referência úteis aos demais agentes.

- **Formulação de Propostas**: Após identificar os objetivos, Alpha cria duas propostas de raciocínio distintas, cada uma representando um caminho específico para a resposta. Essas propostas são então enviadas aos executores E1 e E2 para desenvolvimento independente.

- **Decisão sobre a Necessidade de Pesquisa na Internet**: Alpha avalia se uma consulta externa é necessária para obter informações adicionais relevantes. Caso positivo, ele seleciona um prompt apropriado para realizar essa pesquisa.

- **Escolha do Modelo Adequado**: Com base na natureza da solicitação de resposta (matemática, lógica, análise textual, etc.), Alpha escolhe os modelos mais adequados para cada tarefa a partir de uma lista de APIs disponíveis. Essa escolha cuidadosa se assemelha ao critério de seleção de agentes em cada camada da metodologia MoA, que considera tanto o desempenho quanto a diversidade dos modelos.

- **Comunicação com o Usuário**: Alpha gera uma saída inicial para o usuário, descrevendo brevemente o plano de ação e a abordagem proposta.

2.2. Executores E1 e E2

Os **Executores E1 e E2** são responsáveis por desenvolver as propostas formuladas pelo Propositor Alpha. Esses agentes trabalham de forma independente e paralela, permitindo a exploração de diferentes abordagens para a mesma solicitação. Suas funções principais incluem:

- **Recebimento de Pesquisa Externa**: Se Alpha decide que uma pesquisa externa é necessária, E1 e E2 recebem e incorporam as informações obtidas para desenvolver suas respectivas soluções.

- **Recordação dos Objetivos**: Cada executor mantém o objetivo inicial em mente e concentra-se em alcançá-lo, com base na proposta designada.

- **Desenvolvimento de Propostas via "Cadeia de Pensamento"**: E1 e E2 utilizam o método de *cadeia de pensamento* para desenvolver suas soluções. Isso significa que constroem suas respostas de maneira estruturada e progressiva,

detalhando cada etapa do raciocínio. Esse método permite uma análise profunda e uma exploração rigorosa, similar ao processo de refinamento contínuo descrito na metodologia MoA, onde agentes sucessivos recebem saídas intermediárias e as aprimoram. Cada executor documenta seus passos, reflexões e resultados intermediários, explorando abordagens alternativas para alcançar os objetivos estabelecidos.

- **Comunicação com o Usuário**: Ao final do desenvolvimento de suas cadeias de pensamento, E1 e E2 geram uma saída breve que resume suas conclusões e o raciocínio seguido.

2.3. Agregador Ômega

O **Agregador Ômega** é a entidade que coordena a decisão final sobre a adequação das soluções desenvolvidas pelos executores E1 e E2. Suas funções incluem:

- **Avaliação das Soluções Recebidas**: Ômega recebe as respostas geradas por E1 e E2 e testa se os objetivos foram efetivamente alcançados. Esse papel assemelha-se ao do modelo agregador descrito na metodologia MoA, que sintetiza as respostas de outros agentes para gerar uma saída de alta qualidade.

- **Comparação de Abordagens e Decisão de Caminho Futuro**: Se as abordagens de E1 e E2 falham em atingir os objetivos, Ômega propõe um plano para o próximo ciclo de iteração. Esse plano pode incluir ajustes nos executores ou, se necessário, uma mudança drástica na estratégia, criando novas propostas para um novo ciclo. Essa dinâmica de ajuste é fundamental na abordagem MoA, onde a iteração contínua leva a um refinamento progressivo das respostas.

- **Iteração ou Finalização**: Se Ômega conclui que o objetivo foi atingido, finaliza o processo e gera a resposta final para o usuário. Caso contrário, decide se o processo deve continuar, definindo novas abordagens para E1 e E2.

- **Comunicação com o Usuário**: Quando Ômega decide continuar a iteração, imprime uma mensagem breve explicando os próximos passos. Se a solução foi encontrada, ele imprime a resposta final para o usuário.

3. Processo Iterativo e Cooperativo

O sistema é projetado para operar de maneira iterativa e cooperativa, onde os agentes não atuam isoladamente, mas como componentes de um ciclo de melhoria contínua. O Propositor Alpha atua como estrategista inicial, os Executores E1 e E2 desenvolvem as respostas, e o Agregador Ômega avalia e decide sobre o progresso. Essa interação permite que o sistema evolua em resposta a problemas complexos, refinando suas abordagens até que uma solução satisfatória seja atingida ou até que se determine que a solução não é possível. Esse processo é análogo ao descrito na metodologia MoA, onde a colaboração entre múltiplos agentes, cada um contribuindo com suas habilidades específicas, leva a uma melhoria significativa na qualidade das respostas.

4. Comparação com Mixture-of-Agents (MoA)

A metodologia *Mixture-of-Agents* (MoA) é uma abordagem recente que utiliza múltiplos LLMs em camadas sucessivas para gerar respostas de alta qualidade, onde agentes colaborativos refinam respostas de maneira iterativa e organizada. No entanto, o sistema aqui descrito apresenta características distintas e inovações em relação à estrutura MoA, particularmente pela implementação de uma resolução baseada em cadeia de pensamento, a iteração dinâmica e a capacidade de consulta à internet.

Uma diferença central é que o sistema aplica o método *cadeia de pensamento* como base de resolução. A cada execução dos Executores E1 e E2, não se espera que esses agentes resolvam o problema integralmente; ao contrário, eles abordam partes do problema, desenvolvendo raciocínios intermediários que serão analisados e refinados nas iterações seguintes. Isso permite que os agentes avancem gradualmente, com análises passo a passo, garantindo que o entendimento e a solução final sejam construídos progressivamente.

Além disso, o sistema oferece a possibilidade de consulta à internet, permitindo que os agentes acessem informações externas, conforme a necessidade identificada pelo Propositor Alpha. Em contraste, o MoA trabalha exclusivamente com informações internas dos modelos, limitadas ao conhecimento disponível em suas bases.

A arquitetura também adota uma iteração dinâmica, onde o Agregador Ômega decide sobre a continuidade ou finalização do processo com base nos resultados

obtidos, proporcionando flexibilidade e ajuste às necessidades específicas de cada problema.

5. Importância do “Chain of Thought” no Sistema

O método *chain of thought* desempenha um papel central na estrutura do sistema multiagente, servindo como ferramenta essencial para a construção de soluções completas e coerentes. Em particular, E1 e E2 utilizam essa técnica para estruturar e detalhar suas propostas de solução, facilitando a avaliação pelo Agregador Ômega e garantindo clareza no processo de resolução de problemas. Esse método se alinha à natureza iterativa e colaborativa da arquitetura, permitindo que cada ciclo de iteração se aproxime mais do objetivo final.

6. Importância do “In-Context Few-Shot Learning” no Sistema

Outra técnica essencial é o *in-context few-shot learning via prompting*, que permite aos agentes aprender a partir de poucos exemplos apresentados diretamente no prompt. Esse método é eficaz em sistemas que requerem generalização rápida e adaptação a tarefas variadas, permitindo ao sistema desenvolver abordagens flexíveis e eficazes para uma ampla gama de problemas.

7. Considerações

A arquitetura proposta proporciona um sistema robusto e flexível, capaz de resolver problemas complexos de forma iterativa e colaborativa. Com o Ômega (Agregador) garantindo a continuidade ou a finalização dos ciclos, e os modelos Alpha (Propositor), E1 e E2 (Executores) explorando abordagens diversas, a metodologia se alinha com o MoA, demonstrando como agentes colaborativos podem aprimorar a qualidade das respostas.

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 13 de nov. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

Victor Emanuel da Silva Monteiro

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Nesta Semana, foram realizadas melhorias no **Sistema Multiagente baseado em LLMs com Respostas Iterativas** em relação à implementação de uma escolha automática sobre quais modelos de linguagem utilizar para responder a solicitação do usuário.

Esse mecanismo permite que o sistema escolha, de forma autônoma, o modelo LLM mais adequado para cada tipo de tarefa (dentre uma lista de opções pré-configuradas), com base na análise inicial da mensagem do usuário e da complexidade envolvida.

Versão 1.3-Sistema Multiagente baseado em LLMs com respostas iterativas.ipynb

Implementação de escolha automática de modelos

- **Identificação de Tarefas**: O sistema identifica o tipo de tarefa (como código, matemática, análise lógica sobre um assunto) para definir as necessidades específicas da solicitação, usando o modelo GPT 4o mini.
- **Regras de Seleção de Modelos**: Com base na solicitação do usuário, o sistema escolhe automaticamente o modelo mais apropriado:
  - Para **tarefas de código**, é escolhido o modelo Codestral, especializado em programação.
  - Para **tarefas simples ou criativas**, utiliza o modelo Ministral 8B, que oferece baixo custo gerando respostas de menor complexidade.
  - Para **tarefas de lógica ou matemática**, opta pelo modelo Mistral Large, que possui um desempenho avançado para análises de lógica ou matemática.
  - Para **tarefas gerais**, utiliza o modelo Mistral Small com 22B de parâmetros, que equilibra bom desempenho e eficiência para uma variedade de solicitações.

Flexibilidade e Adaptação: Caso o sistema detecte a necessidade de trocar o modelo ao longo das iterações, o Agregador Ômega realiza a modificação, garantindo a melhor resposta possível para o problema em questão.

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

- Verificar como o sistema se comporta com assuntos diversos
- Melhorar os prompts
- Planejar uma forma de avaliação

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

Relembrando estrutura do Sistema Multiagente baseado em LLMs com Respostas Iterativas:

- Propositor Alpha (GPT 4o mini):
  - Realiza a análise inicial da solicitação do usuário;
  - Define os objetivos;
  - Formula propostas de raciocínio para gerar a resposta;
  - Escolhe qual LLM usar;
  - Decide se existe a necessidade de pesquisa na internet.
- Executores E1 e E2 (Codestral, Ministral 8B, Mistral Small ou Mistral Large):
  - Executam as propostas de raciocínio desenvolvidas passo a passo para encontrar possíveis respostas a fim de atender a solicitação do usuário.
- Agregador Ômega (GPT 4o mini):
  - Avalia, compara e sintetiza as respostas dos executores;
  - Tem autonomia para decidir sobre a continuidade ou a finalização do processo com base nos objetivos do Alpha;
  - Escolhe qual LLM usar em caso de nova iteração;
  - Decide se existe a necessidade de pesquisa na internet em caso de nova iteração.

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

Versão 1.3: Sistema Multiagente baseado em LLMs com respostas iterativas

Visão geral do sistema atual

- Propositor Alpha (GPT 4o mini):
- Realiza a análise inicial da solicitação do usuário;
- Define os objetivos;
- Formula propostas de raciocínio para gerar a resposta;
- Escolhe qual LLM usar;
- Decide se existe a necessidade de pesquisa na internet.
- Executores E1 e E2 (Codestral, Ministral 8B, Mistral Small ou Mistral Large):
- Executam as propostas de raciocínio desenvolvidas passo a passo para encontrar possíveis respostas a fim de atender a solicitação do usuário.
- Agregador Ômega (GPT 4o mini):
- Avalia, compara e sintetiza as respostas dos executores;
- Tem autonomia para decidir sobre a continuidade ou a finalização do processo com base nos objetivos do Alpha;
- Escolhe qual LLM usar em caso de nova iteração;
- Decide se existe a necessidade de pesquisa na internet em caso de nova iteração.

```
!pip install -U langchain-community
!pip install google-search-results
from serpapi import GoogleSearch

!pip install mistralai
%pip install -qU langchain-openai
!pip install -q google
!pip install -q serpapi
!sudo apt-get update


# Importações de pacotes e módulos
import json
import time
import os
import requests
from typing import List, Dict, Any, Optional, Tuple
from dataclasses import dataclass
from pydantic import BaseModel, Field, field_validator, validator, ConfigDict
from rich import print as rprint
from rich.console import Console
from rich.panel import Panel
from rich.table import Table
from mistralai import Mistral
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI  # Import correto
from langchain.prompts import ChatPromptTemplate
from langchain.output_parsers import PydanticOutputParser
from langchain.utilities import SerpAPIWrapper
from langchain.chains import LLMChain
```

```
from serpapi import GoogleSearch
from google.colab import userdata
```

```
Collecting langchain-community
  Downloading langchain_community-0.3.7-py3-none-any.whl.metadata (2.9 kB)
Requirement already satisfied: PyYAML>=5.3 in /usr/local/lib/python3.10/dist-packages (from langchain-community) (6.0.2)
Collecting SQLAlchemy<2.0.36,>=1.4 (from langchain-community)
  Downloading SQLAlchemy-2.0.35-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (9.6 kB)
Requirement already satisfied: aiohttp<4.0.0,>=3.8.3 in /usr/local/lib/python3.10/dist-packages (from langchain-community) (3.10.10)
Collecting dataclasses-json<0.7,>=0.5.7 (from langchain-community)
  Downloading dataclasses_json-0.6.7-py3-none-any.whl.metadata (25 kB)
Collecting httpx-sse<0.5.0,>=0.4.0 (from langchain-community)
  Downloading httpx_sse-0.4.0-py3-none-any.whl.metadata (9.0 kB)
Requirement already satisfied: langchain<0.4.0,>=0.3.7 in /usr/local/lib/python3.10/dist-packages (from langchain-community) (0.3.7)
Requirement already satisfied: langchain-core<0.4.0,>=0.3.17 in /usr/local/lib/python3.10/dist-packages (from langchain-community) (0.3.17)
Requirement already satisfied: langsmith<0.2.0,>=0.1.125 in /usr/local/lib/python3.10/dist-packages (from langchain-community) (0.1.142)
Requirement already satisfied: numpy<2,>=1 in /usr/local/lib/python3.10/dist-packages (from langchain-community) (1.26.4)
Collecting pydantic-settings<3.0.0,>=2.4.0 (from langchain-community)
  Downloading pydantic_settings-2.6.1-py3-none-any.whl.metadata (3.5 kB)
Requirement already satisfied: requests<3,>=2 in /usr/local/lib/python3.10/dist-packages (from langchain-community) (2.32.3)
Requirement already satisfied: tenacity!=8.4.0,<10,>=8.1.0 in /usr/local/lib/python3.10/dist-packages (from langchain-community) (9.0.0)
Requirement already satisfied: aiohappyeyeballs>=2.3.0 in /usr/local/lib/python3.10/dist-packages
```

(from aiohttp<4.0.0,>=3.8.3->langchain-community) (2.4.3)
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/python3.10/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain-community) (1.3.1)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain-community) (24.2.0)
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain-community) (1.5.0)
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.10/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain-community) (6.1.0)
Requirement already satisfied: yarl<2.0,>=1.12.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain-community) (1.17.1)
Requirement already satisfied: async-timeout<5.0,>=4.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain-community) (4.0.3)
Collecting marshmallow<4.0.0,>=3.18.0 (from dataclasses-json<0.7,>=0.5.7->langchain-community)
  Downloading marshmallow-3.23.1-py3-none-any.whl.metadata (7.5 kB)
Requirement already satisfied: typing-inspect<1,>=0.4.0 in /usr/local/lib/python3.10/dist-packages (from dataclasses-json<0.7,>=0.5.7->langchain-community) (0.9.0)
Requirement already satisfied: langchain-text-splitters<0.4.0,>=0.3.0 in /usr/local/lib/python3.10/dist-packages (from langchain<0.4.0,>=0.3.7->langchain-community) (0.3.2)
Requirement already satisfied: pydantic<3.0.0,>=2.7.4 in /usr/local/lib/python3.10/dist-packages (from langchain<0.4.0,>=0.3.7->langchain-community) (2.9.2)
Requirement already satisfied: jsonpatch<2.0,>=1.33 in /usr/local/lib/python3.10/dist-packages (from langchain-core<0.4.0,>=0.3.17->langchain-community) (1.33)
Requirement already satisfied: packaging<25,>=23.2 in /usr/local/lib/python3.10/dist-packages (from langchain-core<0.4.0,>=0.3.17->langchain-community) (24.2)
Requirement already satisfied: typing-extensions>=4.7 in /usr/local/lib/python3.10/dist-packages (from langchain-core<0.4.0,>=0.3.17->langchain-community) (4.12.2)
Requirement already satisfied: httpx<1,>=0.23.0 in /usr/local/lib/python3.10/dist-packages (from langsmith<0.2.0,>=0.1.125->langchain-community) (0.27.2)
Requirement already satisfied: orjson<4.0.0,>=3.9.14 in /usr/local/lib/python3.10/dist-packages (from langsmith<0.2.0,>=0.1.125->langchain-community) (3.10.11)

Requirement already satisfied: requests-toolbelt<2.0.0,>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from langsmith<0.2.0,>=0.1.125->langchain-community) (1.0.0)

Collecting python-dotenv>=0.21.0 (from pydantic-settings<3.0.0,>=2.4.0->langchain-community)

Downloading python_dotenv-1.0.1-py3-none-any.whl.metadata (23 kB)

Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests<3,>=2->langchain-community) (3.4.0)

Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests<3,>=2->langchain-community) (3.10)

Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests<3,>=2->langchain-community) (2.2.3)

Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests<3,>=2->langchain-community) (2024.8.30)

Requirement already satisfied: greenlet!=0.4.17 in /usr/local/lib/python3.10/dist-packages (from SQLAlchemy<2.0.36,>=1.4->langchain-community) (3.1.1)

Requirement already satisfied: anyio in /usr/local/lib/python3.10/dist-packages (from httpx<1,>=0.23.0->langsmith<0.2.0,>=0.1.125->langchain-community) (3.7.1)

Requirement already satisfied: httpcore==1.* in /usr/local/lib/python3.10/dist-packages (from httpx<1,>=0.23.0->langsmith<0.2.0,>=0.1.125->langchain-community) (1.0.6)

Requirement already satisfied: sniffio in /usr/local/lib/python3.10/dist-packages (from httpx<1,>=0.23.0->langsmith<0.2.0,>=0.1.125->langchain-community) (1.3.1)

Requirement already satisfied: h11<0.15,>=0.13 in /usr/local/lib/python3.10/dist-packages (from httpcore==1.*->httpx<1,>=0.23.0->langsmith<0.2.0,>=0.1.125->langchain-community) (0.14.0)

Requirement already satisfied: jsonpointer>=1.9 in /usr/local/lib/python3.10/dist-packages (from jsonpatch<2.0,>=1.33->langchain-core<0.4.0,>=0.3.17->langchain-community) (3.0.0)

Requirement already satisfied: annotated-types>=0.6.0 in /usr/local/lib/python3.10/dist-packages (from pydantic<3.0.0,>=2.7.4->langchain<0.4.0,>=0.3.7->langchain-community) (0.7.0)

Requirement already satisfied: pydantic-core==2.23.4 in /usr/local/lib/python3.10/dist-packages (from pydantic<3.0.0,>=2.7.4->langchain<0.4.0,>=0.3.7->langchain-community) (2.23.4)

Requirement already satisfied: mypy-extensions>=0.3.0 in /usr/local/lib/python3.10/dist-packages (from typing-inspect<1,>=0.4.0->dataclasses-json<0.7,>=0.5.7->langchain-community) (1.0.0)

Requirement already satisfied: propcache>=0.2.0 in /usr/local/lib/python3.10/dist-packages (from yarl<2.0,>=1.12.0->aiohttp<4.0.0,>=3.8.3->langchain-community) (0.2.0)
Requirement already satisfied: exceptiongroup in /usr/local/lib/python3.10/dist-packages (from anyio->httpx<1,>=0.23.0->langsmith<0.2.0,>=0.1.125->langchain-community) (1.2.2)
Downloading langchain_community-0.3.7-py3-none-any.whl (2.4 MB)

2.4/2.4 MB 27.0 MB/s eta 0:00:00
y-2.0.35-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (3.1 MB)

3.1/3.1 MB 72.4 MB/s eta 0:00:00
arshmallow-3.23.1-py3-none-any.whl (49 kB)

49.5/49.5 kB 3.0 MB/s eta 0:00:00
y, python-dotenv, marshmallow, httpx-sse, dataclasses-json, pydantic-settings, langchain-community
  Attempting uninstall: SQLAlchemy
    Found existing installation: SQLAlchemy 2.0.36
    Uninstalling SQLAlchemy-2.0.36:
    Successfully uninstalled SQLAlchemy-2.0.36
Successfully installed SQLAlchemy-2.0.35 dataclasses-json-0.6.7 httpx-sse-0.4.0 langchain-community-0.3.7 marshmallow-3.23.1 pydantic-settings-2.6.1 python-dotenv-1.0.1

```
console = Console()

class EnvironmentConfig:
    @staticmethod
    def setup_environment():
    """Configura as variáveis de ambiente necessárias utilizando Secrets do Colab."""
    try:
    os.environ["OPENAI_API_KEY"] = userdata.get("OPENAI_API_KEY")
    os.environ["SERPAPI_API_KEY"] = userdata.get("SERPAPI_API_KEY")
    os.environ["MISTRAL_API_KEY"] = userdata.get("MISTRAL_API_KEY")
```

```
        # Verificação se as variáveis estão disponíveis
        required_keys = ["OPENAI_API_KEY", "SERPAPI_API_KEY", "MISTRAL_API_KEY"]
        missing_keys = [key for key in required_keys if not os.getenv(key)]

        if missing_keys:
        raise ValueError(f"Variáveis de ambiente não configuradas: {', '.join(missing_keys)}")

        return True
        except Exception as e:
        console.print(f"[red]Erro ao configurar variáveis de ambiente: {e}[/red]")
        return False
from typing import List, Optional, Dict, Any
from pydantic import BaseModel, Field
from enum import Enum
from typing import Optional, Dict
from mistralai import Mistral
import os

class TaskType(Enum):
        CODE = "code"
        MATH = "math"
        LOGIC = "logic"
        CREATIVE = "creative"
        ANALYSIS = "analysis"
        SIMPLE = "simple"

class ModelSelector:
        MODELS = {
        "gpt-4o-mini": {"description": "Modelo principal para análise e coordenação"},
        "ministral-3b-latest": {"description": "Modelo base para tarefas gerais"},
```

```
"ministral-8b-latest": {"description": "Modelo para alto desempenho e análise"},
"mistral-large-latest": {"description": "Modelo para raciocínio complexo"},
"mistral-small-latest": {"description": "Modelo para tarefas mais simples"},
"codestral-latest": {"description": "Modelo especializado em código"}
}

@staticmethod
def get_available_models() -> str:
"""Retorna uma descrição formatada dos modelos disponíveis para o Alpha."""
return "\n".join([
f"- {model}: {info['description']}"
for model, info in ModelSelector.MODELS.items()
])

@staticmethod
def select_model(problem: str, alpha_analysis: str) -> str:
"""Seleciona o modelo mais apropriado baseado na análise da tarefa."""
complexity_scores = ModelSelector.analyze_task_complexity(problem, alpha_analysis)

# Determina o tipo principal da tarefa
main_task_type = max(complexity_scores.items(), key=lambda x: x[1])[0]

# Se a tarefa é principalmente código
if main_task_type == TaskType.CODE.value:
return "codestral-latest"

# Se a tarefa é simples ou criativa
if main_task_type in [TaskType.SIMPLE.value, TaskType.CREATIVE.value]:
return "mistral-small-latest"

# Se é uma tarefa de análise ou matemática
```

```
        if main_task_type in [TaskType.ANALYSIS.value, TaskType.MATH.value]:
        return "mistral-8b-latest"

        # Se é uma tarefa de lógica complexa
        if main_task_type == TaskType.LOGIC.value and
complexity_scores[TaskType.LOGIC.value] > 0.6:
        return "mistral-large-latest"

        # Para tarefas gerais de complexidade média
        if sum(complexity_scores.values()) / len(complexity_scores) > 0.3:
        return "mistral-8b-latest"

        # Default para tarefas mais simples
        return "mistral-3b-latest"

class ModelManager:
        def __init__(self):
        # Verifica as chaves de API
        self.openai_api_key = os.environ.get("OPENAI_API_KEY")
        self.mistral_api_key = os.environ.get("MISTRAL_API_KEY")

        if not self.openai_api_key:
        raise ValueError("OPENAI_API_KEY não encontrada nas variáveis de ambiente")
        if not self.mistral_api_key:
        raise ValueError("MISTRAL_API_KEY não encontrada nas variáveis de ambiente")

        # Inicializa o cliente Mistral
        self.mistral_client = Mistral(api_key=self.mistral_api_key)

        def complete(self, messages: list, model: Optional[str] = None) -> str:
        """Executa a completion usando o modelo especificado."""
```

```
use_model = model or "gpt-4o-mini"

if use_model == "gpt-4o-mini":
# Usar OpenAI
return self.complete_with_openai(messages)
else:
# Usar Mistral diretamente
try:
formatted_messages = []
for msg in messages:
        if isinstance(msg, dict):
        formatted_messages.append(msg)
        else:
        # Se for uma tupla (role, content)
        formatted_messages.append({
        "role": msg[0],
        "content": msg[1]
        })

# Corrigido para usar chat.complete em vez de chat diretamente
response = self.mistral_client.chat.complete(
        model=use_model,
        messages=formatted_messages
)
return response.choices[0].message.content
except Exception as e:
console.print(f"[red]Erro ao usar modelo Mistral {use_model}: {e}[/red]")
# Fallback para OpenAI em caso de erro
console.print("[yellow]Fazendo fallback para OpenAI...[/yellow]")
return self.complete_with_openai(messages)
```

```
    def complete_with_openai(self, messages: list) -> str:
    """Executa a completion usando OpenAI."""
    try:
    llm = ChatOpenAI(
    model="gpt-4o-mini",  # ou o modelo OpenAI desejado
            temperature=0,
    api_key=self.openai_api_key
    )

    # Formatar mensagens se necessário
    formatted_messages = []
    for msg in messages:
    if isinstance(msg, dict):
                formatted_messages.append(msg)
    else:
                # Se for uma tupla (role, content)
                formatted_messages.append({
                "role": msg[0],
                "content": msg[1]
            })

    response = llm.invoke(formatted_messages)
    return response.content
    except Exception as e:
    console.print(f"[red]Erro ao usar OpenAI: {e}[/red]")
    raise

class Step(BaseModel):
    step_number: int = Field(description="O número da etapa em sequência")
    description: str = Field(description="Descrição do que fazer nesta etapa")
    reasoning: str = Field(description="Raciocínio por trás desta etapa")
```

```
class Hypothesis(BaseModel):
    hypothesis: str = Field(description="Descrição clara e detalhada da hipótese")
    steps: List[Step] = Field(description="Lista de etapas para testar esta hipótese")

class AlphaResponse(BaseModel):
    model_config = ConfigDict(protected_namespaces=())  # Resolve o warning do Pydantic

    analysis: str = Field(description="Análise detalhada do problema")
    objectives: str = Field(description="Declaração clara e específica dos objetivos")
    hypotheses: List[Hypothesis] = Field(description="Lista de exatamente duas hipóteses a serem testadas")
    selected_model: str = Field(description="Modelo selecionado para E2")
    model_reasoning: str = Field(description="Justificativa para a seleção do modelo")

    @field_validator('hypotheses')
    @classmethod
    def validate_hypotheses(cls, v: List[Hypothesis]) -> List[Hypothesis]:
    if len(v) != 2:
    raise ValueError('Devem ser fornecidas exatamente duas hipóteses')
    return v

class ExecutorStep(BaseModel):
    step_number: int = Field(description="O número da etapa em sequência")
    action: str = Field(description="Ação realizada nesta etapa")
    result: str = Field(description="Resultado da ação")
    reasoning: str = Field(description="Raciocínio por trás da ação e resultado")
    insights: str = Field(description="Insights descobertos nesta etapa")
    patterns_identified: str = Field(  # Campo obrigatório adicionado
    description="Padrões identificados nesta etapa",
    default=""  # Valor padrão vazio para evitar erros
    )
```

```
    partial_solution: Optional[str] = Field(
    description="Solução parcial encontrada nesta etapa",
    default=None
    )

class ExecutorResponse(BaseModel):
    summary: str = Field(description="Resumo breve da tentativa de execução")
    reasoning_process: str = Field(description="Processo de raciocínio detalhado")
    detailed_steps: List[ExecutorStep] = Field(description="Etapas detalhadas da execução")
    patterns_found: str = Field(  # Agora é string simples
    description="Padrões descobertos até o momento",
    default=""  # Valor padrão vazio
    )
    decoded_segments: str = Field(  # Agora é string simples
    description="Segmentos já decodificados",
    default=""  # Valor padrão vazio
    )
    conclusion: str = Field(description="Conclusão da tentativa")
    next_actions: str = Field(description="Próximas etapas sugeridas")
    analysis_state: str = Field(
    description="Estado atual da análise",
    default=""  # Valor padrão vazio
    )

class SearchDecision(BaseModel):
    needs_search: bool = Field(description="Se é necessário realizar uma pesquisa")
    search_query: Optional[str] = Field(description="A consulta de pesquisa a ser realizada")
    reasoning: str = Field(description="Raciocínio por trás da decisão de pesquisa")

class OmegaResponse(BaseModel):
    model_config = ConfigDict(protected_namespaces=())  # Resolve o warning do Pydantic
```

```
    evaluation: str = Field(description="Avaliação do progresso atual")
    reasoning_process: str = Field(description="Processo de raciocínio detalhado")
    discovered_patterns: str = Field(description="Padrões descobertos e confirmados")
    decoded_content: str = Field(description="Conteúdo decodificado até o momento")
    progress_summary: str = Field(description="Resumo do progresso atual")
    objective_achieved: bool = Field(description="Se o objetivo foi alcançado")
    final_response: Optional[str] = Field(description="Solução final se o objetivo foi alcançado")
    next_step: Optional[str] = Field(description="Próximas etapas se o objetivo não foi alcançado")
    accumulated_knowledge: str = Field(description="Conhecimento acumulado até o momento")
    hypothesis_to_continue: Optional[Hypothesis] = Field(description="Hipótese para continuar")
    search_decision: SearchDecision = Field(description="Decisão sobre necessidade de pesquisa")
    selected_model: Optional[str] = Field(description="Modelo recomendado para próxima iteração", default=None)
    selection_reasoning: Optional[str] = Field(description="Justificativa para a seleção do modelo", default=None)


class Agent:
    def __init__(self, agent_id: str, model: str = "gpt-4o-mini"):
    self.agent_id = agent_id
    self.model = model
    self.model_manager = ModelManager()  # Cada agente tem seu próprio ModelManager

    def get_completion(self, messages: list) -> str:
    """Método para obter completion usando o ModelManager."""
    return self.model_manager.complete(messages, self.model)

class PropositorAlpha(Agent):
```

```
        def __init__(self, model: str = "gpt-4o-mini"):
        super().__init__("Propositor Alpha", model)
        self.parser = PydanticOutputParser(pydantic_object=AlphaResponse)

        # Define o template do sistema com ênfase em soluções concretas
        self.system_template = """You are the Propositor Alpha, an agent that analyzes problems and generates two different approaches to solve them through step-by-step reasoning.

You must also select the most appropriate model for the E2 executor based on the problem characteristics.

Available models:
{models}

For problems involving code, ALWAYS select "codestral-latest" as the model, as it is specialized in code-related tasks.

For analytical and decoding problems, generate EXACTLY TWO different approaches and select an appropriate model for E2 (E1 always uses gpt-4o-mini).

Your response should include:
1. Analysis of the problem
2. Selection of the most appropriate model for E2 with clear reasoning
3. Two different hypotheses for solving the problem

For code-related problems:
- Each hypothesis MUST include the EXACT code or commands that need to be written/executed
- Include specific file modifications if needed
- Show before/after examples when relevant
- Specify exact package versions if needed
- Always provide concrete, actionable steps that can be directly copied and used
```

{format_instructions}

REMEMBER:
1. You MUST provide EXACTLY TWO hypotheses
2. Each hypothesis MUST have at least THREE steps
3. Steps MUST be concrete and actionable, with exact commands or code
4. Each step MUST include clear reasoning
5. Both hypotheses MUST provide different approaches
6. You MUST select and justify a model for E2
7. For code problems, ALWAYS select codestral-latest
8. Focus on providing concrete, copy-pasteable solutions

Analyze this problem step by step and select the appropriate model: {problem}"""

```
def analyze(self, problem: str) -> Optional[AlphaResponse]:
try:
# Preparar as instruções e conteúdo
format_instructions = self.parser.get_format_instructions()
system_content = self.system_template.format(
models=ModelSelector.get_available_models(),
format_instructions=format_instructions,
problem=problem
)

# Criar as mensagens
messages = [
{"role": "system", "content": system_content},
{"role": "user", "content": problem}
]
```

```
        # Obter a resposta do modelo
        response = self.get_completion(messages)

        # Parsear a resposta
        parsed_response = self.parser.parse(response)
        return parsed_response

        except Exception as e:
        console.print(f"[red]Erro na análise do Propositor Alpha: {e}[/red]")
        if hasattr(e, '__traceback__'):
        import traceback
        console.print(f"[red]Traceback: {''.join(traceback.format_tb(e.__traceback__))}")
        return None


class ExecutorAgent(Agent):
        def __init__(self, agent_id: str, model: str = "gpt-4o-mini"):
        super().__init__(agent_id, model)
        self.parser = PydanticOutputParser(pydantic_object=ExecutorResponse)

        # Define o template do sistema
        self.system_template = """You are {agent_id}, an executor agent that follows a step-by-step
approach to solve analytical and decoding problems.
You have access to the complete history of previous iterations to build upon earlier discoveries.

{format_instructions}

Important guidelines:
1. For EACH step you take, you MUST include:
  - Clear action description
  - Specific result observed
  - Detailed reasoning
```

- Insights gained
- Patterns identified (even if none, state "No new patterns identified in this step")
- Any partial solutions found

2. When documenting patterns:
- Be explicit about what was found
- Use clear, descriptive strings
- Include pattern type and examples
- Link to previous patterns if relevant

3. For the overall response, ensure:
- All steps are properly documented
- Patterns are summarized clearly
- Decoded segments are explicitly stated
- Analysis state is clearly described

4. Build upon previous iterations:
- Reference earlier findings
- Show how new discoveries relate to past ones
- Maintain consistency in pattern descriptions

Remember: Every step must include ALL required fields, especially patterns_identified!"""

self.user_template = """Original Problem: {objective}
Current Hypothesis: {hypothesis}

PREVIOUS ITERATIONS HISTORY:
{complete_history}

Current Progress: {current_progress}
Previous Patterns: {patterns_found}

```
Decoded So Far: {decoded_segments}"""

    def execute(self, objective: str, hypothesis: Hypothesis,
    complete_history: str,
    current_progress: str = "",
    patterns_found: str = "",
    decoded_segments: str = "") -> Optional[ExecutorResponse]:
    try:
    format_instructions = self.parser.get_format_instructions()

    # Formatar o conteúdo do sistema
    system_content = self.system_template.format(
    agent_id=self.agent_id,
    format_instructions=format_instructions
    )

    # Formatar o conteúdo do usuário
    user_content = self.user_template.format(
    objective=objective,
    hypothesis=hypothesis.model_dump_json(),
    complete_history=complete_history,
    current_progress=current_progress,
    patterns_found=patterns_found,
    decoded_segments=decoded_segments
    )

    messages = [
    {"role": "system", "content": system_content},
    {"role": "user", "content": user_content}
    ]
```

```
            response = self.get_completion(messages)
            return self.parser.parse(response)
            except Exception as e:
            console.print(f"[red]Erro na execução do {self.agent_id}: {e}[/red]")
           return None

class AgregadorOmega(Agent):
            def __init__(self, model: str = "gpt-4o-mini"):
            super().__init__("Agregador Omega", model)
            self.parser = PydanticOutputParser(pydantic_object=OmegaResponse)

            self.system_template = """You are the Agregador Omega, responsible for evaluating progress
and making decisions about next steps.

{format_instructions}

Important guidelines:
1. Evaluate the logical progression of reasoning in both executor responses
2. If a clear pattern or solution is found, set objective_achieved = true
3. For decoding problems:
  - Verify pattern consistency across examples
  - Check if the decoded message makes sense
  - Validate transformations are reversible
4. Look for:
  - Successful pattern identifications
  - Inconsistencies in decoding
  - Missing or unclear steps
5. When providing a new hypothesis:
  - Build upon successful pattern matches
  - Address any gaps in the decoding
  - Focus on extending working approaches
```

```
6. Consider model performance:
  - Evaluate if the current model is effective
  - Recommend model changes if needed
  - Provide clear reasoning for model recommendations

For search decisions:
1. Default to NO search for decoding problems
2. Only request searches for verifying decoded content meaning
3. Explain clearly why any search would be necessary"""

        self.user_template = """Original Problem: {problem}
Execution History: {history}
E1 Response: {e1_response}
E2 Response: {e2_response}"""

        def evaluate(self, history: str, problem: str, e1_response: ExecutorResponse,
        e2_response: ExecutorResponse) -> Optional[OmegaResponse]:
        try:
        # Preparar as instruções e conteúdo
        format_instructions = self.parser.get_format_instructions()
        system_content = self.system_template.format(
        format_instructions=format_instructions
        )

        user_content = self.user_template.format(
        problem=problem,
        history=history,
        e1_response=e1_response.model_dump_json(),
        e2_response=e2_response.model_dump_json()
        )
```

```
        # Criar as mensagens
        messages = [
        {"role": "system", "content": system_content},
        {"role": "user", "content": user_content}
        ]

        # Obter a resposta do modelo
        response = self.get_completion(messages)

        # Parsear a resposta
        return self.parser.parse(response)

         except Exception as e:
        console.print(f"[red]Erro na avaliação do Agregador Omega: {e}[/red]")
        if hasattr(e, '__traceback__'):
        import traceback
        console.print(f"[red]Traceback: {''.join(traceback.format_tb(e.__traceback__))}")
        return None

class MultiAgentSystem:
        def __init__(self, model: str = "gpt-4o-mini"):
        if not EnvironmentConfig.setup_environment():
        raise ValueError("Failed to setup environment")

        self.search_engine = SerpAPIWrapper()
        self.history: List[Dict[str, Any]] = []
        self.current_progress: str = ""
        self.patterns_found: str = ""
        self.decoded_segments: str = ""

        # Inicialização do gerenciador de modelos
        self.model_manager = ModelManager()
```

```
# Alpha e Omega sempre usam gpt-4o-mini
self.propositor_alpha = PropositorAlpha("gpt-4o-mini")
self.agregador_omega = AgregadorOmega("gpt-4o-mini")

# E1 e E2 serão inicializados depois
self.e1 = None
self.e2 = None

self.initial_hypotheses: Optional[List[Hypothesis]] = None
self.original_problem: Optional[str] = None

def initialize_executors(self, problem: str, alpha_response: AlphaResponse):
"""Inicializa os executores com os modelos apropriados baseado na análise do Alpha."""
# E1 sempre usa gpt-4o-mini
self.e1 = ExecutorAgent("E1", "gpt-4o-mini")
# E2 usa o modelo recomendado pelo Alpha
selected_model = alpha_response.selected_model or "mistral-3b-latest"
console.print(f"[yellow]Modelo selecionado para E2: {selected_model}[/yellow]")
console.print(f"[yellow]Razão: {alpha_response.model_reasoning}[/yellow]")
self.e2 = ExecutorAgent("E2", selected_model)

def build_history_string(self) -> str:
history = ""
for i, entry in enumerate(self.history, start=1):
history += f"\n=== Iteração {i} ===\n"
if 'e1' in entry:
history += f"\nE1 Findings (Model: gpt-4o-mini):\n"
history += f"Summary: {entry['e1'].summary}\n"
history += f"Reasoning: {entry['e1'].reasoning_process}\n"
for step in entry['e1'].detailed_steps:
```

```
            history += f"Step {step.step_number}:\n"
            history += f"- Action: {step.action}\n"
            history += f"- Result: {step.result}\n"
            history += f"- Insights: {step.insights}\n"
            history += f"- Patterns: {step.patterns_identified}\n"
        history += f"Conclusion: {entry['e1'].conclusion}\n"

        if 'e2' in entry:
        history += f"\nE2 Findings (Model: {entry['e2_model']}):\n"
        history += f"Summary: {entry['e2'].summary}\n"
        history += f"Reasoning: {entry['e2'].reasoning_process}\n"
        for step in entry['e2'].detailed_steps:
            history += f"Step {step.step_number}:\n"
            history += f"- Action: {step.action}\n"
            history += f"- Result: {step.result}\n"
            history += f"- Insights: {step.insights}\n"
            history += f"- Patterns: {step.patterns_identified}\n"
        history += f"Conclusion: {entry['e2'].conclusion}\n"

        if 'omega' in entry:
        history += f"\nOmega Evaluation:\n"
        history += f"Evaluation: {entry['omega'].evaluation}\n"
        history += f"Discovered Patterns: {entry['omega'].discovered_patterns}\n"
        history += f"Decoded Content: {entry['omega'].decoded_content}\n"
        history += f"Progress Summary: {entry['omega'].progress_summary}\n"
        if entry['omega'].selected_model:
            history += f"Model Recommendation: {entry['omega'].selected_model}\n"
            history += f"Model Reasoning: {entry['omega'].selection_reasoning}\n"

        history += "\nState at End of Iteration:\n"
        history += f"Patterns Found: {entry.get('patterns', '')}\n"
```

```
        history += f"Decoded Segments: {entry.get('decoded', '')}\n"
        history += "=" * 50 + "\n"

        return history

    def display_iteration_results(self, iteration: int, e1_response: ExecutorResponse,
                                  e2_response: ExecutorResponse, omega_response: OmegaResponse,
                                  search_results: Optional[str] = None):
        table = Table(title=f"Resultados da Iteração {iteration}")
        table.add_column("Agente", style="cyan")
        table.add_column("Resultado", style="magenta")

        if search_results:
            table.add_row("Resultados da Pesquisa", search_results)

        # E1 Results
        table.add_row("E1 - Modelo", "gpt-4o-mini")
        table.add_row("E1 - Resumo", e1_response.summary)
        table.add_row("E1 - Raciocínio", e1_response.reasoning_process)
        for step in e1_response.detailed_steps:
            table.add_row(
                f"E1 - Passo {step.step_number}",
                f"Ação: {step.action}\nResultado: {step.result}\nInsights: {step.insights}\nPadrões: {step.patterns_identified}"
            )
        table.add_row("E1 - Padrões", e1_response.patterns_found)
        table.add_row("E1 - Conclusão", e1_response.conclusion)

        # E2 Results
        table.add_row("E2 - Modelo", self.e2.model)
        table.add_row("E2 - Resumo", e2_response.summary)
```

```
        table.add_row("E2 - Raciocínio", e2_response.reasoning_process)
        for step in e2_response.detailed_steps:
        table.add_row(
        f"E2 - Passo {step.step_number}",
        f"Ação: {step.action}\nResultado: {step.result}\nInsights: {step.insights}\nPadrões:
{step.patterns_identified}"
        )
        table.add_row("E2 - Padrões", e2_response.patterns_found)
        table.add_row("E2 - Conclusão", e2_response.conclusion)

        # Omega Results
        table.add_row("Omega - Avaliação", omega_response.evaluation)
        table.add_row("Omega - Raciocínio", omega_response.reasoning_process)
        table.add_row("Omega - Padrões", omega_response.discovered_patterns)
        table.add_row("Omega - Progresso", omega_response.progress_summary)

        # Verificar se há recomendação de modelo usando os atributos corretos
        if omega_response.selected_model and omega_response.selection_reasoning:
        table.add_row("Omega - Recomendação de Modelo",
                f"Modelo: {omega_response.selected_model}\nRazão:
{omega_response.selection_reasoning}")

        # Estado Atual
        table.add_row("Estado Atual - Padrões", self.patterns_found)
        table.add_row("Estado Atual - Decodificado", self.decoded_segments)
        table.add_row("Estado Atual - Progresso", self.current_progress)

        console.print(table)

        if omega_response.next_step:
            console.print(f"[yellow]Próxima Etapa: {omega_response.next_step}[/yellow]")
```

```
    def update_state(self, omega_response: OmegaResponse) -> None:
    """Atualiza o estado do sistema com base na resposta do Omega."""
    if omega_response.discovered_patterns.strip():
    self.patterns_found = omega_response.discovered_patterns

    if omega_response.decoded_content.strip():
    self.decoded_segments = omega_response.decoded_content

    if omega_response.progress_summary.strip():
    self.current_progress = omega_response.progress_summary

    def run_iteration(self, hypotheses: List[Hypothesis], iteration: int) -> bool:
    console.print(f"\n[bold cyan]--- Iteração {iteration} ---[/bold cyan]")

    complete_history = self.build_history_string()
    search_results = None

    # Se não for primeira iteração, verifica se Omega recomendou troca de modelo
    if self.history and iteration > 1:
    last_omega = self.history[-1]['omega']
        if last_omega.selected_model and last_omega.selected_model != self.e2.model:
    console.print(f"[yellow]Alterando modelo do E2 para:
{last_omega.selected_model}[/yellow]")
    console.print(f"[yellow]Razão: {last_omega.selection_reasoning}[/yellow]")
    self.e2 = ExecutorAgent("E2", last_omega.selected_model)

    while True:
    # Executar E1
    e1_response = self.e1.execute(
    self.original_problem,
```

```
hypotheses[0],
complete_history,
self.current_progress,
self.patterns_found,
self.decoded_segments
)

# Executar E2
e2_response = self.e2.execute(
self.original_problem,
hypotheses[1],
complete_history,
self.current_progress,
self.patterns_found,
self.decoded_segments
)

if not e1_response or not e2_response:
console.print("[red]Falha na execução dos agentes.[/red]")
return False

# Avaliar resultados com Omega
omega_response = self.agregador_omega.evaluate(
complete_history,
self.original_problem,
e1_response,
e2_response
)

if not omega_response:
console.print("[red]Falha na avaliação do Agregador Omega.[/red]")
```

```
            return False

                # Atualizar o estado do sistema
            self.update_state(omega_response)

            # Verificar se precisa fazer uma pesquisa
            if omega_response.search_decision.needs_search and
omega_response.search_decision.search_query:
            console.print(f"[yellow]Realizando pesquisa:
{omega_response.search_decision.search_query}[/yellow]")
            try:
                    search_results =
self.search_engine.run(omega_response.search_decision.search_query)
                    continue  # Repetir a iteração com os resultados da pesquisa
            except Exception as e:
                    console.print(f"[red]Erro na pesquisa: {e}[/red]")
                    search_results = "Error performing search"

            # Mostrar resultados da iteração
            self.display_iteration_results(iteration, e1_response, e2_response, omega_response,
search_results)

            # Verificar se o objetivo foi alcançado
            if omega_response.objective_achieved:
            console.print(Panel(
                    omega_response.final_response or "Solução encontrada, mas sem resposta final
fornecida.",
                    title="[bold green]Solução Final[/bold green]",
                    border_style="green"
            ))
            return True
```

```
# Adicionar ao histórico
self.history.append({
"iteration": iteration,
"e1": e1_response,
"e2": e2_response,
"e1_model": "gpt-4o-mini",
"e2_model": self.e2.model,
"omega": omega_response,
"patterns": self.patterns_found,
"decoded": self.decoded_segments
})

break  # Sair do loop se não precisar de mais pesquisas

return False

def solve_problem(self, problem: str, max_iterations: int = 5) -> None:
console.print(f"\n[bold green]Iniciando a resolução do problema:[/bold green] {problem}\n")

self.original_problem = problem

  # Obter análise inicial do Propositor Alpha
alpha_response = self.propositor_alpha.analyze(problem)
if not alpha_response:
console.print("[red]Erro: Propositor Alpha não gerou hipóteses.[/red]")
return

# Inicializar executores com modelos escolhidos pelo Alpha
self.initialize_executors(problem, alpha_response)
self.initial_hypotheses = alpha_response.hypotheses
```

```
        console.print(f"[yellow]Análise Inicial:[/yellow]\n{alpha_response.analysis}")
        console.print(f"[yellow]Objetivos:[/yellow]\n{alpha_response.objectives}")

        # Loop principal de iterações
        iteration = 1
        current_hypotheses = self.initial_hypotheses

        while iteration <= max_iterations:
        if self.run_iteration(current_hypotheses, iteration):
        console.print("\n[bold green]Problema resolvido com sucesso![/bold green]")
        break

        # Se não resolvido, verificar próximos passos do Omega
        if self.history:
        last_omega_response = self.history[-1]['omega']
        if last_omega_response.hypothesis_to_continue:
                current_hypotheses = [
                last_omega_response.hypothesis_to_continue,
                self.initial_hypotheses[1]
                ]

        iteration += 1
        time.sleep(1)  # Pequena pausa entre iterações

        if iteration > max_iterations:
        console.print("\n[bold red]Número máximo de iterações atingido sem encontrar uma solução.[/bold red]")

def main():
        try:
```

```
        system = MultiAgentSystem(model="gpt-4o-mini")

        # Exemplo de uso
        problem = """como resolver esse warning
        <ipython-input-16-31635a91617c>:85: LangChainDeprecationWarning: The class `ChatOpenAI` was deprecated in LangChain 0.0.10 and will be removed in 1.0. An updated version of the class exists in the :class:`~langchain-openai package and should be used instead. To use it run `pip install -U :class:`~langchain-openai` and import as `from :class:`~langchain_openai import ChatOpenAI``.
  self.llm = ChatOpenAI(model=model, temperature=0)"""
        system.solve_problem(problem)

        except Exception as e:
        console.print(f"[red]Erro durante a execução: {e}[/red]")

if __name__ == "__main__":
        main()
```

```
Iniciando a resolução do problema: como resolver esse warning
        <ipython-input-16-31635a91617c>:85: LangChainDeprecationWarning: The class `ChatOpenAI` was deprecated in
LangChain 0.0.10 and will be removed in 1.0. An updated version of the class exists in the
:class:`~langchain-openai package and should be used instead. To use it run `pip install -U
:class:`~langchain-openai` and import as `from :class:`~langchain_openai import ChatOpenAI``.
  self.llm = ChatOpenAI(model=model, temperature=0)


Modelo selecionado para E2: codestral-latest

Razão: O problema envolve a atualização de código e a resolução de um aviso de depreciação, o que requer um foco em
```

tarefas de codificação. O modelo 'codestral-latest' é especializado em tarefas de código, tornando-o a escolha mais
apropriada para este problema.

Análise Inicial:
O problema é um aviso de depreciação relacionado ao uso da classe `ChatOpenAI` da biblioteca LangChain. A classe
foi descontinuada na versão 0.0.10 e será removida na versão 1.0. O aviso sugere que uma nova versão da classe está
disponível no pacote `langchain-openai`, e que o código deve ser atualizado para evitar problemas futuros.

Objetivos:
Atualizar o código para usar a nova classe `ChatOpenAI` do pacote `langchain-openai` e eliminar o aviso de
depreciação.

--- Iteração 1 ---

Resultados da Iteração 1

Agente	Resultado
E1 - Modelo	gpt-4o-mini
E1 - Resumo	O aviso de depreciação foi abordado através da atualização do pacote e da modificação das importações no código.
E1 - Raciocínio	O aviso de depreciação indica que a classe `ChatOpenAI` não deve mais ser

		utilizada e que uma nova versão está disponível no pacote `langchain-openai`.
		Para resolver isso, é necessário instalar o novo pacote, modificar as importações
		e atualizar a instância da classe no código.
E1 - Passo 1		Ação: Instalar o pacote `langchain-openai` usando pip.
		Resultado: O pacote foi instalado com sucesso, garantindo que a versão mais
		recente esteja disponível.
		Insights: A instalação do pacote é um passo crucial para garantir que o código
		funcione corretamente com as atualizações.
		Padrões: No new patterns identified in this step.
E1 - Passo 2		Ação: Modificar a importação da classe `ChatOpenAI` no código.
		Resultado: A importação foi alterada para `from langchain_openai import
		ChatOpenAI`.
		Insights: Modificar as importações é uma prática comum ao atualizar bibliotecas,
		garantindo que o código esteja alinhado com as versões mais recentes.
		Padrões: No new patterns identified in this step.
E1 - Passo 3		Ação: Substituir a instância da classe `ChatOpenAI` no código existente.
		Resultado: A instância da classe foi atualizada para usar a nova implementação,
		eliminando o aviso de depreciação.
		Insights: A atualização da instância é essencial para garantir que o código

	funcione corretamente e esteja livre de avisos.	
	Padrões: No new patterns identified in this step.	
E1 - Padrões	No new patterns discovered until now.	
E1 - Conclusão	O aviso de depreciação foi resolvido com sucesso ao atualizar o pacote e	
	modificar as importações e instâncias no código.	
E2 - Modelo	codestral-latest	
E2 - Resumo	The warning issue has been successfully addressed by updating the package,	
	creating a wrapper class, and modifying the existing code.	
E2 - Raciocínio	The warning message was decoded, and a solution was implemented using a	
	step-by-step approach. The existing code was modified to use the new ChatOpenAI	
	class without any errors.	
E2 - Passo 1	Ação: Installing the updated langchain-openai package using pip.	
	Resultado: The package is successfully installed, and the updated version	
	contains the new ChatOpenAI class.	
	Insights: The new ChatOpenAI class is a replacement for the deprecated version,	
	and it should be used to avoid future deprecation warnings.	
	Padrões: No new patterns identified in this step.	
E2 - Passo 2	Ação: Creating a wrapper for the ChatOpenAI class.	
	Resultado: The wrapper class is successfully created, and it encapsulates the	
	ChatOpenAI class.	
	Insights: Using a wrapper allows the existing code to be modified to use the new	
	ChatOpenAI class without significant changes.	

		Padrões: No new patterns identified in this step.
E2 - Passo 3	Ação: Modifying the existing code to use the wrapper.	
	Resultado: The code is successfully modified, and it uses the new wrapper class	
	without any errors.	
	Insights: The wrapper class provides a seamless transition to the new implementation without requiring major code changes.	
	Padrões: No new patterns identified in this step.	
E2 - Padrões	No new patterns were identified in this process.	
E2 - Conclusão	The warning issue has been successfully addressed. The existing code can now utilize the new ChatOpenAI class without any errors.	
Omega - Avaliação	O aviso de depreciação foi resolvido com sucesso ao atualizar o pacote e modificar as importações e instâncias no código.	
Omega - Raciocínio	Ambas as respostas abordaram a depreciação da classe `ChatOpenAI` e implementaram soluções eficazes. A primeira resposta focou na atualização do pacote e na modificação das importações, enquanto a segunda introduziu uma classe wrapper para manter a compatibilidade com o código existente.	
Omega - Padrões	Nenhum novo padrão foi identificado durante o processo.	
Omega - Progresso	As etapas necessárias para resolver o aviso de depreciação foram executadas com sucesso, e o código foi atualizado para utilizar a nova classe `ChatOpenAI`.	
Estado Atual - Padrões	Nenhum novo padrão foi identificado durante o processo.	

Estado Atual - Decodificado	O aviso de depreciação foi identificado e abordado através da atualização do
	pacote e do código.
Estado Atual - Progresso	As etapas necessárias para resolver o aviso de depreciação foram executadas com
	sucesso, e o código foi atualizado para utilizar a nova classe `ChatOpenAI`.

Próxima Etapa: Testar o código atualizado para garantir que não haja mais avisos de depreciação e que a
funcionalidade permaneça intacta.

Solução Final

| O aviso de depreciação foi resolvido com sucesso. O código atualizado não apresenta mais avisos e mantém a |
| funcionalidade. |

Problema resolvido com sucesso!

APÊNDICE 6
Termo de Aceite de Entrega

Objetivo deste documento
Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 27 de nov. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

Victor Emanuel da Silva Monteiro

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Nesta Semana, foram realizados testes com diferentes assuntos para identificar pontos de melhoria no Sistema Multiagente Baseado em LLMs com Respostas Iterativas. Durante os testes, foi observado que, em algumas etapas do processo de cadeia de raciocínio, havia perda de informações, o que comprometia a qualidade e a rastreabilidade dos resultados.

Implementação de Correção da Perda de Informações no Processo:

Foi identificado que a perda de informações ocorria durante a execução de algumas etapas do raciocínio, dificultando a validação e o controle dos resultados.

Para solucionar esse problema, foi desenvolvido um espaço dedicado onde cada modelo registra de forma detalhada:

- **Ação realizada:** O que foi feito.
- **Resultados obtidos:** Os resultados alcançados.
- **Observações relevantes:** Comentários ou insights sobre a execução.
- **Status do passo:** Indicação clara de sucesso ou falha na etapa.

Essa estruturação melhorou a rastreabilidade e garantiu maior consistência nas respostas ao usuário.

Planejamento para Avaliação de Desempenho
Além das ações corretivas, foi identificado o dataset *Reveal: A Benchmark for Verifiers of Reasoning Chains*, criado pelo Google, como uma ferramenta para avaliação do sistema.

- Este dataset é especialmente projetado para testar modelos que utilizam *Chain of Thought*, apresentando questões que exigem raciocínios detalhados e sequenciais e permite testar a capacidade do sistema desenvolvido.

Atualizações do código:
Versão 1.4.4-Sistema Multiagente baseado em LLMs com respostas iterativas.ipynb

Atualização da documentação: Versão 1.4 - Desenvolvimento de um Sistema Multiagente baseado e…
Paper de dataset benchmark do Google:
A Chain-of-Thought Is as Strong as Its Weakest Link- A Benchmark for Verifiers of Reasoning Chains…

Planejamento: **[descrever o que pretende fazer para realizar a próxima ENTREGA]**

- Continuar a implementação da avaliação do sistema

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

LEONARDO ANTÔNIO ALVES: Go!

Versão Resumo 1.4.4: Sistema Multiagente baseado em LLMs com respostas iterativas

1. Introdução

Este documento apresenta uma análise detalhada sobre o funcionamento de um sistema multiagente baseado em Modelos de Linguagem Grande (LLMs), projetado para processar solicitações complexas por meio de uma abordagem iterativa e colaborativa. A estrutura do sistema é composta por agentes que desempenham papéis específicos, organizados em uma arquitetura que permite explorar diferentes caminhos para alcançar resultados satisfatórios. A inspiração para essa solução deriva do conceito de Mixture-of-Agents (MoA), que promove a integração e a colaboração entre múltiplos modelos especializados, ampliando as possibilidades de análise e execução.

A ideia central do sistema é delegar funções distintas a diferentes agentes — Propositor Alpha, Executores E1 e E2, e Agregador Ômega — que, em conjunto, desenvolvem soluções de maneira estruturada. O Propositor Alpha atua como estrategista, os Executores elaboram soluções específicas e detalhadas, enquanto o Agregador avalia e sintetiza as melhores respostas. Essa divisão de responsabilidades é fundamental para garantir que o sistema funcione de maneira eficiente, mesmo em cenários de alta complexidade.

Além disso, o sistema adota o método Cadeia de Pensamento (chain of thought) como fundamento principal para a construção das soluções, permitindo uma análise detalhada e incremental das etapas envolvidas. Cada agente contribui com informações e processos especializados, formando um ciclo contínuo de iteração e refinamento. Isso assegura que, mesmo diante de problemas difíceis, o sistema possa evoluir progressivamente até chegar a uma resposta satisfatória ou determinar que o objetivo não pode ser alcançado com os recursos disponíveis.

2. Componentes do Sistema

O sistema é composto por três agentes principais — o Propositor Alpha, os Executores E1 e E2, e o Agregador Ômega. Cada um desses agentes desempenha funções distintas e complementares, permitindo que o sistema opere de forma iterativa e eficiente. Essa colaboração é inspirada no conceito de MoA, onde diferentes modelos trabalham em conjunto para aprimorar as respostas por meio de sucessivas iterações.

A modularidade é um dos principais pontos fortes dessa abordagem. Cada agente é projetado para desempenhar tarefas específicas, desde a análise inicial até a avaliação final. Isso garante que o sistema possa se adaptar a diferentes tipos de solicitações, distribuindo as responsabilidades de forma lógica e eficiente. Por exemplo, enquanto o Propositor Alpha é responsável pela estratégia inicial, os Executores trabalham em abordagens detalhadas e o Agregador avalia as respostas, promovendo um ciclo de aprendizado contínuo.

Outra característica importante é a integração entre os agentes. Os dados e resultados são compartilhados em um formato padronizado, facilitando a análise e o rastreamento de cada etapa do processo. Além disso, o sistema adota práticas de registro extensivo, garantindo que todas as ações e decisões sejam documentadas para posterior avaliação ou auditoria. Esse nível de detalhamento aumenta a transparência e a confiabilidade do sistema.

2.1. Propositor Alpha

O Propositor Alpha desempenha um papel estratégico no sistema, sendo responsável por definir a abordagem inicial para lidar com a solicitação recebida. Sua principal função é decompor o problema em partes menores e gerenciáveis, estabelecendo uma sequência clara de passos que os executores deverão seguir. Essa análise inicial é crucial para assegurar que o sistema avance de maneira estruturada e eficiente.

Além de formular o plano de execução, o Alpha é responsável por selecionar os modelos mais adequados para cada tarefa, com base na complexidade e no tipo de solicitação. Essa escolha é feita com base em critérios como a necessidade de raciocínio lógico, capacidade de análise ou geração criativa. O Alpha utiliza uma lista de modelos disponíveis, priorizando aqueles que melhor atendem às exigências

específicas de cada etapa. O raciocínio por trás dessa seleção é documentado, permitindo que outras partes do sistema entendam a lógica das decisões tomadas.
Outro aspecto importante do Propositor Alpha é sua capacidade de identificar a necessidade de buscar informações externas. Caso seja detectado que o conhecimento atual do sistema é insuficiente para lidar com a solicitação, o Alpha pode ativar o mecanismo de busca. Ele gera consultas bem estruturadas para recuperar informações relevantes da internet e compartilha esses dados com os executores. Essa funcionalidade torna o sistema mais adaptável e capaz de lidar com problemas que vão além do conhecimento pré-treinado dos modelos.

2.2. Executores E1 e E2

Os Executores E1 e E2 são agentes especializados que trabalham paralelamente para implementar as etapas definidas pelo Propositor Alpha. Sua principal tarefa é desenvolver soluções detalhadas utilizando o método de cadeia de pensamento, onde cada passo do raciocínio é documentado de forma estruturada. Essa abordagem facilita tanto a avaliação das etapas pelo Agregador Ômega quanto a continuidade do processo em ciclos subsequentes.
Uma das melhorias recentes do sistema é a capacidade de os executores utilizarem diferentes modelos, dependendo da natureza da etapa que estão processando. Enquanto E1 utiliza um modelo fixo de alta capacidade, E2 pode alternar entre modelos mais leves ou especializados, conforme recomendado pelo Alpha. Isso permite que os executores adaptem sua abordagem às exigências específicas de cada tarefa, otimizando os recursos disponíveis e aumentando a eficiência do sistema.
Além disso, os executores agora registram detalhadamente todas as suas ações, resultados e observações. Esse registro inclui informações sobre o que foi feito, os resultados obtidos, se a etapa foi bem-sucedida e quaisquer insights relevantes que possam orientar iterações futuras. Essa documentação é essencial para garantir a rastreabilidade do processo, permitindo que o Agregador Ômega avalie as respostas de maneira informada e que o sistema, como um todo, aprenda com as etapas anteriores.

2.3. Agregador Ômega

O Agregador Ômega atua como o ponto central de avaliação e síntese dentro do sistema. Ele é responsável por comparar as soluções geradas pelos executores,

identificar a abordagem mais eficaz e decidir sobre os próximos passos do processo. Sua função é garantir que as etapas sejam concluídas com sucesso ou propor ajustes necessários para alcançar os objetivos estabelecidos.

Uma das principais funções do Ômega é a avaliação detalhada dos resultados. Ele utiliza critérios objetivos para determinar qual executor produziu a solução mais adequada, considerando fatores como a coerência lógica, a completude e a aderência aos objetivos da etapa. Além disso, o Ômega mantém um histórico completo de todas as iterações, registrando as decisões tomadas e as justificativas para cada escolha. Esse histórico é usado não apenas para fins de auditoria, mas também como base para melhorar o desempenho do sistema em iterações futuras.

Outro aspecto fundamental é a construção da solução final. O Ômega combina os melhores resultados de cada etapa para formar uma resposta completa e coerente. Ele também verifica se todos os requisitos foram atendidos e, caso identifique lacunas, propõe novos ciclos de iteração. Essa capacidade de adaptação dinâmica é crucial para lidar com problemas complexos que exigem múltiplos níveis de refinamento.

3. Processo Iterativo e Busca Externa

O sistema é projetado para operar de maneira iterativa, onde cada ciclo de execução permite refinar a abordagem e avançar em direção à conclusão. Essa dinâmica começa com a análise inicial do Alpha, que define um plano de execução. Os executores então implementam as etapas, e o Ômega avalia os resultados, decidindo se o processo deve continuar ou ser encerrado.

Uma das funcionalidades mais avançadas do sistema é sua capacidade de realizar buscas externas para complementar o conhecimento disponível. Quando o Alpha detecta que informações adicionais são necessárias, ele ativa um mecanismo de busca baseado em APIs como SERPAPI. Esse mecanismo permite recuperar dados atualizados e específicos da internet, que são então processados e integrados ao raciocínio dos executores. Essa funcionalidade amplia significativamente o alcance do sistema, permitindo que ele lide com problemas além do escopo de seu treinamento inicial.

Após cada ciclo, o Ômega utiliza as informações disponíveis para ajustar o plano de execução, garantindo que o sistema continue a avançar de forma eficaz. Esse processo iterativo, combinado com a possibilidade de buscar informações externas, torna o sistema altamente adaptável e robusto, capaz de lidar com uma ampla variedade de solicitações de forma eficiente e detalhada.

Conclusão

Com as atualizações implementadas, o sistema se destaca como uma ferramenta poderosa para lidar com problemas complexos de maneira estruturada e colaborativa. A integração de agentes especializados, o uso de chain of thought e a capacidade de busca externa são exemplos de como a arquitetura foi projetada para maximizar eficiência e eficácia. A abordagem iterativa garante que as soluções sejam refinadas progressivamente, promovendo um nível de qualidade e confiabilidade difícil de alcançar por métodos tradicionais.

4. Escolha Automática do Modelo

Uma das inovações mais significativas implementadas no sistema é a capacidade de selecionar automaticamente o modelo mais adequado para cada tarefa, com base em critérios como complexidade, tipo de solicitação e requisitos específicos da etapa. Essa funcionalidade foi desenvolvida para otimizar o desempenho do sistema, garantindo que os recursos computacionais sejam utilizados de maneira eficiente e que as respostas sejam mais precisas e contextualmente relevantes.

O processo de escolha automática começa com uma análise detalhada da tarefa, realizada pelo Propositor Alpha. Com base em informações extraídas do contexto da solicitação e nos objetivos definidos para a etapa, o Alpha utiliza um mecanismo interno para classificar a complexidade e determinar o tipo predominante da tarefa. Entre os tipos de tarefas identificados estão análise, lógica, matemática, criativa, simples ou relacionada a código. Cada tipo de tarefa é então associado a um modelo específico que melhor atende às suas necessidades. Por exemplo, tarefas de lógica complexa podem ser delegadas a modelos robustos como o *mistral-large-latest*, enquanto tarefas simples podem ser processadas por modelos mais leves, como o *mistral-small-latest*.

Essa capacidade de seleção é gerenciada pelo componente ModelSelector, que centraliza as informações sobre os modelos disponíveis, suas descrições e características. Ele calcula uma pontuação de complexidade com base nos requisitos da tarefa e mapeia essas informações para o modelo mais apropriado. Além disso, o sistema garante flexibilidade ao permitir que modelos especializados sejam priorizados em casos onde sua performance seja mais relevante, como o *codestral-latest* para tarefas de programação. Essa abordagem evita o uso excessivo de modelos mais robustos para tarefas triviais, otimizando os tempos de execução e reduzindo custos computacionais.

O Agregador Ômega também se beneficia dessa funcionalidade ao avaliar se o modelo inicialmente selecionado foi eficaz na execução da etapa. Caso um modelo

não produza resultados satisfatórios, o Ômega pode recomendar alterações no plano de execução ou sugerir o uso de outro modelo em ciclos subsequentes. Isso demonstra a adaptabilidade do sistema, que combina automação e avaliação crítica para refinar suas escolhas ao longo do processo.

A escolha automática do modelo não apenas aumenta a eficiência operacional do sistema, mas também fortalece sua capacidade de lidar com solicitações diversificadas. Essa funcionalidade é especialmente útil em cenários onde a natureza da tarefa varia amplamente entre as etapas, garantindo que o modelo correto seja empregado em cada situação. Assim, o sistema entrega soluções com maior qualidade, consistência e alinhamento aos requisitos específicos de cada solicitação.

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 4 de dez. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

Victor Emanuel da Silva Monteiro

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Nesta Semana:

- O dataset *Reveal: A Benchmark for Verifiers of Reasoning Chains* foi explorado com o objetivo de extrair perguntas para testar no sistema desenvolvido.
- Verificado que o benchmark não apresenta respostas padrão esperadas como corretas, apenas as geradas pelos modelos submetidos.
- Foram selecionadas apenas as perguntas acompanhadas de respostas avaliadas como totalmente corretas pelos anotadores do dataset do Google.
- Resultado de 90,32% de acerto
- Foi implementada uma interface gráfica

Assistente Virtual Baseado em Multiagentes Colaborativos: Sistema Iterativo com Cadeia de Pensamento e Mixture of Agents

- Propositor Alpha:
  - Realiza a análise inicial da solicitação do usuário;
  - Define os objetivos;
  - Formula propostas de raciocínio para gerar a resposta;
  - Escolhe qual LLM usar;
  - Decide se existe a necessidade de pesquisa na internet.
- Executores E1 e E2:
  - Executam as propostas de raciocínio desenvolvidas passo a passo para encontrar possíveis respostas a fim de atender a solicitação do usuário.
- Agregador Ômega:
  - Avalia, compara e sintetiza as respostas dos executores;
  - Tem autonomia para decidir sobre a continuidade ou a finalização do processo com base nos objetivos do Alpha;

- Escolhe qual LLM usar em caso de nova iteração, caso a iteração atual esteja com dificuldade usando o LLM mais básico;
- Decide se existe a necessidade de pesquisa na internet em caso de nova iteração.

Atualizações do código:
Versão 1.4.7-Sistema Multiagente baseado em LLMs com respostas iterativas.ipynb

Atualização da documentação:
Versão 1.7 - Desenvolvimento de um Sistema Multiagente baseado em LLMs para Resolução Iterativ…

Planejamento: **[descrever o que pretende fazer para realizar a próxima ENTREGA]**

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

LEONARDO ANTÔNIO ALVES: Go!

Versão 1.7: Sistema Multiagente baseado em LLMs com respostas iterativas

1. Introdução

Este documento apresenta uma análise detalhada sobre o funcionamento de um sistema multiagente baseado em Modelos de Linguagem Grande (LLMs), projetado para processar solicitações complexas por meio de uma abordagem iterativa e colaborativa. A estrutura do sistema é composta por agentes que desempenham papéis específicos, organizados em uma arquitetura que permite explorar diferentes caminhos para alcançar resultados satisfatórios. A inspiração para essa solução deriva do conceito de Mixture-of-Agents (MoA), que promove a integração e a colaboração entre múltiplos modelos especializados, ampliando as possibilidades de análise e execução.

A ideia central do sistema é delegar funções distintas a diferentes agentes — Propositor Alpha, Executores E1 e E2, e Agregador Ômega — que, em conjunto, desenvolvem soluções de maneira estruturada. O Propositor Alpha atua como estrategista, os Executores elaboram soluções específicas e detalhadas, enquanto o Agregador avalia e sintetiza as melhores respostas. Essa divisão de responsabilidades é fundamental para garantir que o sistema funcione de maneira eficiente, mesmo em cenários de alta complexidade.
Além disso, o sistema adota o método Cadeia de Pensamento (chain of thought) como fundamento principal para a construção das soluções, permitindo uma análise detalhada e incremental das etapas envolvidas. Cada agente contribui com informações e processos especializados, formando um ciclo contínuo de iteração e refinamento. Isso assegura que, mesmo diante de problemas difíceis, o sistema possa evoluir progressivamente até chegar a uma resposta satisfatória ou determinar que o objetivo não pode ser alcançado com os recursos disponíveis.

2. Componentes do Sistema

O sistema é composto por três agentes principais — o Propositor Alpha, os Executores E1 e E2, e o Agregador Ômega. Cada um desses agentes desempenha funções distintas e complementares, permitindo que o sistema opere de forma iterativa e eficiente. Essa colaboração é inspirada no conceito de MoA, onde diferentes modelos trabalham em conjunto para aprimorar as respostas por meio de sucessivas iterações.
A modularidade é um dos principais pontos fortes dessa abordagem. Cada agente é projetado para desempenhar tarefas específicas, desde a análise inicial até a avaliação final. Isso garante que o sistema possa se adaptar a diferentes tipos de solicitações, distribuindo as responsabilidades de forma lógica e eficiente. Por exemplo, enquanto o Propositor Alpha é responsável pela estratégia inicial, os Executores trabalham em abordagens detalhadas e o Agregador avalia as respostas, promovendo um ciclo de aprendizado contínuo.
Outra característica importante é a integração entre os agentes. Os dados e resultados são compartilhados em um formato padronizado, facilitando a análise e o rastreamento de cada etapa do processo. Além disso, o sistema adota práticas de registro extensivo, garantindo que todas as ações e decisões sejam documentadas para posterior avaliação ou auditoria. Esse nível de detalhamento aumenta a transparência e a confiabilidade do sistema.

2.1. Propositor Alpha

O Propositor Alpha desempenha um papel estratégico no sistema, sendo responsável por definir a abordagem inicial para lidar com a solicitação recebida. Sua principal função é decompor o problema em partes menores e gerenciáveis, estabelecendo uma sequência clara de passos que os executores deverão seguir. Essa análise inicial é crucial para assegurar que o sistema avance de maneira estruturada e eficiente.

Além de formular o plano de execução, o Alpha é responsável por selecionar os modelos mais adequados para cada tarefa, com base na complexidade e no tipo de solicitação. Essa escolha é feita com base em critérios como a necessidade de raciocínio lógico, capacidade de análise ou geração criativa. O Alpha utiliza uma lista de modelos disponíveis, priorizando aqueles que melhor atendem às exigências específicas de cada etapa. O raciocínio por trás dessa seleção é documentado, permitindo que outras partes do sistema entendam a lógica das decisões tomadas.

Outro aspecto importante do Propositor Alpha é sua capacidade de identificar a necessidade de buscar informações externas. Caso seja detectado que o conhecimento atual do sistema é insuficiente para lidar com a solicitação, o Alpha pode ativar o mecanismo de busca. Ele gera consultas bem estruturadas para recuperar informações relevantes da internet e compartilha esses dados com os executores. Essa funcionalidade torna o sistema mais adaptável e capaz de lidar com problemas que vão além do conhecimento pré-treinado dos modelos.

2.2. Executores E1 e E2

Os Executores E1 e E2 são agentes especializados que trabalham paralelamente para implementar as etapas definidas pelo Propositor Alpha. Sua principal tarefa é desenvolver soluções detalhadas utilizando o método de cadeia de pensamento, onde cada passo do raciocínio é documentado de forma estruturada. Essa abordagem facilita tanto a avaliação das etapas pelo Agregador Ômega quanto a continuidade do processo em ciclos subsequentes.

Uma das melhorias recentes do sistema é a capacidade de os executores utilizarem diferentes modelos, dependendo da natureza da etapa que estão processando. Enquanto E1 utiliza um modelo fixo de alta capacidade, E2 pode alternar entre modelos mais leves ou especializados, conforme recomendado pelo Alpha. Isso

permite que os executores adaptem sua abordagem às exigências específicas de cada tarefa, otimizando os recursos disponíveis e aumentando a eficiência do sistema.
Além disso, os executores agora registram detalhadamente todas as suas ações, resultados e observações. Esse registro inclui informações sobre o que foi feito, os resultados obtidos, se a etapa foi bem-sucedida e quaisquer insights relevantes que possam orientar iterações futuras. Essa documentação é essencial para garantir a rastreabilidade do processo, permitindo que o Agregador Ômega avalie as respostas de maneira informada e que o sistema, como um todo, aprenda com as etapas anteriores.

2.3. Agregador Ômega

O Agregador Ômega atua como o ponto central de avaliação e síntese dentro do sistema. Ele é responsável por comparar as soluções geradas pelos executores, identificar a abordagem mais eficaz e decidir sobre os próximos passos do processo. Sua função é garantir que as etapas sejam concluídas com sucesso ou propor ajustes necessários para alcançar os objetivos estabelecidos.
Uma das principais funções do Ômega é a avaliação detalhada dos resultados. Ele utiliza critérios objetivos para determinar qual executor produziu a solução mais adequada, considerando fatores como a coerência lógica, a completude e a aderência aos objetivos da etapa. Além disso, o Ômega mantém um histórico completo de todas as iterações, registrando as decisões tomadas e as justificativas para cada escolha. Esse histórico é usado não apenas para fins de auditoria, mas também como base para melhorar o desempenho do sistema em iterações futuras.
Outro aspecto fundamental é a construção da solução final. O Ômega combina os melhores resultados de cada etapa para formar uma resposta completa e coerente. Ele também verifica se todos os requisitos foram atendidos e, caso identifique lacunas, propõe novos ciclos de iteração. Essa capacidade de adaptação dinâmica é crucial para lidar com problemas complexos que exigem múltiplos níveis de refinamento.

3. Processo Iterativo e Busca Externa

O sistema é projetado para operar de maneira iterativa, onde cada ciclo de execução permite refinar a abordagem e avançar em direção à conclusão. Essa dinâmica começa com a análise inicial do Alpha, que define um plano de execução. Os executores então implementam as etapas, e o Ômega avalia os resultados, decidindo se o processo deve continuar ou ser encerrado.

Uma das funcionalidades mais avançadas do sistema é sua capacidade de realizar buscas externas para complementar o conhecimento disponível. Quando o Alpha detecta que informações adicionais são necessárias, ele ativa um mecanismo de busca baseado em APIs como SERPAPI. Esse mecanismo permite recuperar dados atualizados e específicos da internet, que são então processados e integrados ao raciocínio dos executores. Essa funcionalidade amplia significativamente o alcance do sistema, permitindo que ele lide com problemas além do escopo de seu treinamento inicial.

Após cada ciclo, o Ômega utiliza as informações disponíveis para ajustar o plano de execução, garantindo que o sistema continue a avançar de forma eficaz. Esse processo iterativo, combinado com a possibilidade de buscar informações externas, torna o sistema altamente adaptável e robusto, capaz de lidar com uma ampla variedade de solicitações de forma eficiente e detalhada.

Conclusão

Com as atualizações implementadas, o sistema se destaca como uma ferramenta poderosa para lidar com problemas complexos de maneira estruturada e colaborativa. A integração de agentes especializados, o uso de chain of thought e a capacidade de busca externa são exemplos de como a arquitetura foi projetada para maximizar eficiência e eficácia. A abordagem iterativa garante que as soluções sejam refinadas progressivamente, promovendo um nível de qualidade e confiabilidade difícil de alcançar por métodos tradicionais.

4. Escolha Automática do Modelo

Uma das inovações mais significativas implementadas no sistema é a capacidade de selecionar automaticamente o modelo mais adequado para cada tarefa, com base em critérios como complexidade, tipo de solicitação e requisitos específicos da etapa. Essa funcionalidade foi desenvolvida para otimizar o desempenho do

sistema, garantindo que os recursos computacionais sejam utilizados de maneira eficiente e que as respostas sejam mais precisas e contextualmente relevantes.

O processo de escolha automática começa com uma análise detalhada da tarefa, realizada pelo Propositor Alpha. Com base em informações extraídas do contexto da solicitação e nos objetivos definidos para a etapa, o Alpha utiliza um mecanismo interno para classificar a complexidade e determinar o tipo predominante da tarefa. Entre os tipos de tarefas identificados estão análise, lógica, matemática, criativa, simples ou relacionada a código. Cada tipo de tarefa é então associado a um modelo específico que melhor atende às suas necessidades. Por exemplo, tarefas de lógica complexa podem ser delegadas a modelos robustos como o *mistral-large-latest*, enquanto tarefas simples podem ser processadas por modelos mais leves, como o *mistral-small-latest*.

Essa capacidade de seleção é gerenciada pelo componente ModelSelector, que centraliza as informações sobre os modelos disponíveis, suas descrições e características. Ele calcula uma pontuação de complexidade com base nos requisitos da tarefa e mapeia essas informações para o modelo mais apropriado. Além disso, o sistema garante flexibilidade ao permitir que modelos especializados sejam priorizados em casos onde sua performance seja mais relevante. Essa abordagem evita o uso excessivo de modelos mais robustos para tarefas triviais, otimizando os tempos de execução e reduzindo custos computacionais.

O Agregador Ômega também se beneficia dessa funcionalidade ao avaliar se o modelo inicialmente selecionado foi eficaz na execução da etapa. Caso um modelo não produza resultados satisfatórios, o Ômega pode recomendar alterações no plano de execução ou sugerir o uso de outro modelo em ciclos subsequentes. Isso demonstra a adaptabilidade do sistema, que combina automação e avaliação crítica para refinar suas escolhas ao longo do processo.

A escolha automática do modelo não apenas aumenta a eficiência operacional do sistema, mas também fortalece sua capacidade de lidar com solicitações diversificadas. Essa funcionalidade é especialmente útil em cenários onde a natureza da tarefa varia amplamente entre as etapas, garantindo que o modelo correto seja empregado em cada situação. Assim, o sistema entrega soluções com maior qualidade, consistência e alinhamento aos requisitos específicos de cada solicitação.

4. Cadeia de pensamento

A Cadeia de Pensamento (*Chain of Thought*) implementada é um elemento fundamental no funcionamento do sistema, sendo responsável por organizar e conduzir o raciocínio de maneira estruturada e eficiente. Essa abordagem permite que requisições complexas sejam decompostas em etapas menores e logicamente encadeadas, o que facilita a análise e execução de cada parte do processo.

Uma das principais vantagens dessa metodologia é a estruturação do raciocínio, que garante que os agentes sigam um fluxo lógico e sequencial. Isso é particularmente relevante para cenários multifacetados, onde soluções diretas podem ser insuficientes ou suscetíveis a erros. Além disso, a Cadeia de Pensamento promove transparência e rastreabilidade, já que cada etapa é documentada detalhadamente, incluindo validações claras. Esse nível de detalhamento permite que os agentes, como o Agregador Ômega, revisem as decisões e resultados, corrigindo possíveis erros ou refinando as abordagens adotadas.

Nesse sistema, um diferencial significativo da arquitetura do sistema é que cada modelo de linguagem (LLM) executa apenas um passo do processo de maneira focada, em vez de tentar resolver o problema completo ou iterar continuamente sobre o que já produziu. Essa divisão de responsabilidades entre os agentes impede que os modelos gerem inconsistências ao revisar suas próprias respostas, um problema comum em abordagens que dependem de iterações repetidas sobre um resultado inicial. Ao isolar cada etapa e atribuí-la a um agente específico, o sistema garante que o raciocínio permaneça estruturado e que cada parte seja avaliada e ajustada de forma independente, evitando desvios cumulativos ou perda de contexto.

Outro benefício essencial é a colaboração entre agentes, pois a Cadeia de Pensamento facilita a integração de informações e resultados, permitindo que cada agente contribua de forma complementar ao processo. Por exemplo, enquanto os Executores elaboram soluções específicas para cada etapa, o Agregador avalia os resultados no contexto geral, garantindo coerência e aderência aos objetivos. Essa interação contínua entre agentes aumenta a robustez e a confiabilidade do sistema.

A Cadeia de Pensamento também desempenha um papel crítico na redução de erros, uma vez que a divisão das tarefas em etapas menores e a validação constante minimizam a acumulação de problemas. Cada passo é avaliado com critérios objetivos, o que assegura que inconsistências sejam identificadas e corrigidas antes que o processo avance. Além disso, a metodologia é particularmente útil em cenários complexos, pois permite que o sistema se adapte

dinamicamente às exigências da tarefa, refinando soluções parciais, ajustando o plano de execução e buscando informações externas sempre que necessário.
Por fim, essa abordagem promove o aprendizado contínuo do sistema. Ao registrar insights e padrões durante o processo, o sistema aprimora sua capacidade de lidar com solicitações futuras, tornando-se mais eficiente e eficaz. Dessa forma, a Cadeia de Pensamento, combinada com a estratégia de dividir responsabilidades entre os LLMs, não apenas aumenta a precisão das soluções, mas também fortalece a adaptabilidade e a confiabilidade do sistema em lidar com problemas desafiadores.

5. Mixture of Agents

O sistema desenvolvido aqui também adota uma abordagem baseada no conceito de *Mixture of Agents* (MoA), que utiliza múltiplos modelos de linguagem especializados e interconectados para realizar tarefas de alta complexidade de forma colaborativa. Esse conceito baseia-se na premissa de que diferentes agentes, com capacidades distintas, podem contribuir de maneira complementar, maximizando a eficácia da solução e garantindo maior precisão.

No contexto do sistema, cada agente desempenha um papel específico dentro de uma arquitetura modular. O Propositor Alpha é o responsável por criar estratégias e estruturar a execução, decompondo a solicitação inicial em etapas menores e gerenciáveis. Os Executores E1 e E2, por sua vez, são especializados em implementar as etapas planejadas, cada um utilizando modelos de linguagem adequados para a tarefa designada. Finalmente, o Agregador Ômega atua como o avaliador e integrador dos resultados, comparando, refinando e sintetizando as contribuições dos executores.

Essa divisão clara de responsabilidades é um dos principais diferenciais do sistema. O MoA garante que cada agente concentre seus esforços em uma etapa ou função específica, sem a necessidade de abarcar toda a complexidade do processo. Isso elimina redundâncias, evita decisões arbitrárias e promove um fluxo de trabalho lógico e coerente. Por exemplo, enquanto o Propositor Alpha analisa a melhor estratégia para abordar a tarefa, os Executores podem operar de forma independente, garantindo que cada etapa seja realizada com o máximo de eficiência. O Agregador, por sua vez, age como um filtro crítico, identificando as soluções mais adequadas e propondo ajustes ou refinamentos, quando necessário.

Outro aspecto importante do MoA no sistema é a capacidade de cada agente utilizar modelos de linguagem especializados para diferentes tipos de tarefas. Essa escolha estratégica é realizada pelo Propositor Alpha, que aloca os recursos computacionais de maneira otimizada, selecionando o modelo mais adequado para cada etapa com base em critérios como complexidade e requisitos específicos. Isso não apenas melhora a precisão do sistema, mas também reduz custos computacionais e acelera o processo.

A colaboração entre agentes é facilitada por uma troca de informações contínua e estruturada. Os resultados gerados por cada executor são documentados em um formato padronizado, permitindo que o Agregador avalie cada etapa de forma transparente. Esse mecanismo de integração assegura que o sistema funcione como uma entidade coesa, mesmo sendo composto por múltiplos agentes com habilidades distintas.

Por fim, o MoA utilizado no sistema não só melhora a eficiência e a precisão, mas também amplia sua capacidade de adaptação a diferentes contextos. Essa flexibilidade é alcançada graças à disponibilidade de diversos modelos especializados, incluindo gpt-4o-mini, grok-beta, gpt-4o-2024-11-20, gemini-1.5-flash e mistral-large-latest, que integram as operações do sistema e contribuem para sua capacidade de oferecer soluções robustas e eficientes em diversos cenários.