

Grandes Modelos de Linguagem

Autoria:

Sávio Salvarino Teles de Oliveira
Daniel Fazzioni
David O'Neil Campos Ferreira

Organizadores:

Taciana Novo Kudo
Renata Dutra Braga
Deborah Silva Alves Fernandes
Cristiane Bastos Rocha Ferreira
Arlindo Rodrigues Galvão Filho



Universidade Federal de Goiás

Reitora

Angelita Pereira de Lima

Vice-Reitor

Jesiel Freitas Carvalho

Diretora do Cegraf UFG

Maria Lucia Kons

Conselho Editorial da Coleção Formação no AKCIT

Anderson da Silva Soares

Arlindo Rodrigues Galvão Filho

Deborah Silva Alves Fernandes

Juliana Pereira de Souza Zinader

Renata Dutra Braga

Taciana Novo Kudo

Telma Woerle de Lima Soares

Equipe de produção:

Amanda Souza Vitor

Ana Laura Sene Amâncio Zara

Ana Luísa Silva Gonçalves

Caio Barbosa Dias

Daiane Souza Vitor

Dandra Alves de Souza

Davi Oliveira Gomes

Guilherme Correia Dutra

Iuri Vaz Miranda

Isadora Yasmim da Silva

Júlia de Souza Nascimento

Layane Grazielle Souza Dias

Luciana Dantas Soares Alves

Luis Felipe Ferreira Silva

Luiza de Oliveira Costa

Luma Wanderley de Oliveira

Pedro Vitor Silveira Fajardo

Suse Barbosa Castilho

Vinícius Pereira Espíndola

Wagner Wilson Furtado

Wanderley de Souza Alencar

Grandes Modelos de Linguagem

Autoria:

Sávio Salvarino Teles de Oliveira
Daniel Fazzioni
David O'Neil Campos Ferreira

Organizadores:

Taciana Novo Kudo
Renata Dutra Braga
Deborah Silva Alves Fernandes
Cristiane Bastos Rocha Ferreira
Arlindo Rodrigues Galvão Filho

Cegraf UFG
2024

© Cegraf UFG, 2024

© Taciana Novo Kudo

Renata Dutra Braga

Deborah Silva Alves Fernandes

Cristiane Bastos Rocha Ferreira

Arlindo Rodrigues Galvão Filho

© Universidade Federal de Goiás, 2024

© AKCIT, 2024

Revisão Técnica

Deborah Silva Alves Fernandes

Rafael Teixeira Sousa

Revisão Editorial

Ana Laura de Sene Amâncio Zara Brisolla

Capa

Iuri Vaz Miranda

Editoração Eletrônica

Layane Grazielle Souza Dias

Luma Wanderley de Oliveira



Esta obra é disponibilizada nos termos da Licença Creative Commons – Atribuição – Não Comercial – Compartilhamento pela mesma licença 4.0 Internacional. É permitida a reprodução parcial ou total desta obra, desde que citada a fonte.

<https://doi.org/10.5216/OLI.gra.ebook.978-85-495-1096-9/2024>

Dados Internacionais de Catalogação na Publicação (CIP) (Câmara Brasileira do Livro, SP, Brasil)

Oliveira, Sávio Salvarino Teles de
Grandes modelos de linguagem [livro eletrônico] /
Sávio Salvarino Teles de Oliveira, Daniel Fazzioni,
David O'Neil Campos Ferreira ; organizadores Taciana
Novo Kudo...[et al.]. -- Goiânia, GO : Cegraf UFG,
2024.

PDF

Outros organizadores: Renata Dutra Braga,
Deborah Silva Alves Fernandes, Cristiane Bastos
Rocha Ferreira, Arlindo Rodrigues Galvão Filho.
Bibliografia.

ISBN 978-85-495-1096-9

1. Aprendizagem de máquina 2. Inteligência
artificial 3. Linguagem de programação (Computadores)
4. Linguagem e línguas I. Fazzioni, Daniel.
II. Ferreira, David O'Neil Campos.
III. Kudo, Taciana Novo. IV. Braga, Renata Dutra.
V. Fernandes, Deborah Silva Alves. VI. Ferreira,
Cristiane Bastos Rocha. VII. Galvão Filho, Arlindo
Rodrigues. VIII. Título.

25-257993

CDD-005.133

Índices para catálogo sistemático:

1. Linguagem de programação : Computadores :
Processamento de dados 005.133

Grandes Modelos de Linguagem

Instituições responsáveis

Universidade Federal de Goiás (UFG)

Centro de Competência Embrapii em Tecnologias Imersivas, denominado AKCIT (Advanced Knowledge Center for Immersive Technologies)

Centro de Excelência em Inteligência Artificial (CEIA)

Instituições financiadoras

Empresa Brasileira de Pesquisa e Inovação Industrial (Embrapii)

Governo do Estado de Goiás

Empresas parceiras do AKCIT

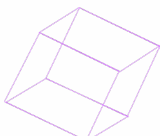
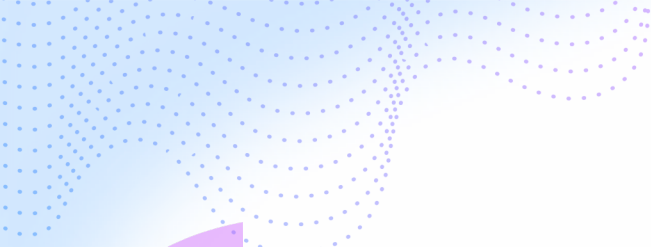
Apoio

Universidade Federal de Goiás (UFG)

Pró-Reitoria de Pesquisa e Inovação (PRPI-UFG)

Instituto de Informática (INF-UFG)





Lista de Abreviaturas e Siglas

APIs	<i>Application Programming Interface</i> - Interface de Programação de Aplicações
AWS	<i>Amazon Web Services</i> - Serviços <i>Web</i> da Amazon
BERT	<i>Bidirecional Encoder Representations from Transformers</i> - Representações de Codificador Bidirecional de Transformadores
CNN	<i>Convolutional Neural Networks</i> - Redes Neurais Convolucionais
CPU	<i>Central Processing Unit</i> - Unidade Central de Processamento
ELMo	<i>Embeddings from Language Model</i> - Incorporações do Modelo de Linguagem
FLOPs	<i>Floating Point Operations per Second</i> - Operações de Ponto Flutuante por Segundo
FTaaS	<i>Fine-Tuning-as-a-Service</i> - Ajuste Fino como Serviço
GPT	<i>Generative Pre-trained Transformer</i> - Transformador Generativo Pré-treinado
GPU	<i>Graphics Processing Units</i> - Unidades de Processamento Gráfico
HMM	<i>Hidden Markov Models</i> - Modelos Ocultos de Markov
IAG	Inteligência Artificial Generativa
ICL	<i>In-Context Learning</i> - Aprendizado In-contexto
LLMs	<i>Large Language Models</i> - Grandes Modelos de Linguagem
LMs	<i>Language Models</i> - Modelos de Linguagem
LoRA	<i>Low-Rank Adaptation</i> - Adaptação de Baixa Dimensão
LSTM	<i>Long Short-Term Memory</i> - Memória de Longo e Curto Prazo
MLN	<i>Neural Language Models</i> - Modelos de Linguagem Neurais



MLP

Probabilistic Language Models - Modelos de Linguagem Probabilísticos

MLOps

Machine Learning Operations - Operações de Aprendizado de Máquina

MP

Multilayer Perceptron - Perceptron Multicamadas

PLN

Processamento de Linguagem Natural

RAG

Retrieval Augmentation Generation - Recuperação e Geração com Aumento

RAM

Random Access Memory - Memória de Acesso Aleatório

RNN

Recurrent Neural Networks - Redes Neurais Recorrentes

TPUs

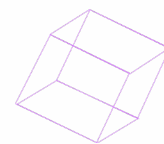
Tensor Processing Units - Unidades de Processamento Tensor

UFG

Universidade Federal de Goiás

VRAM

Video Random Access Memory - Memória de Acesso Aleatório para Vídeo

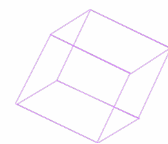


Lista de Equações, Figuras e Tabelas

Equação 1 - Memória necessária para rodar um <i>Large Language Model</i>	233
Equação 2 - Exemplo do Llama 70B com precisão de 16 <i>bits</i>	233
Equação 3 - Quantização do Llama 70B com precisão de 4 <i>bits</i>	234
Figura 1 - Mapa mental que ilustra a evolução e a taxonomia dos modelos de linguagem	18
Figura 2 - Evolução da capacidade de resolução de tarefas em quatro gerações de Modelos de Linguagem	20
Figura 3 - Comparação entre a abordagem usual de treinamento de modelos	24
Figura 4 - Tendências dos números cumulativos de artigos do arXiv que contêm as frases-chave " <i>language model</i> " (desde junho de 2018) (a) e " <i>large language model</i> " (desde outubro de 2019) (b)	27
Figura 5 - Tarefa de completar textos utilizando grandes modelos de linguagem baseados em arquitetura <i>Transformer</i>	28
Figura 6 - Exemplo de aplicação do parâmetro Top-k ($k=2$) na geração de texto	32
Figura 7 - Exemplo de aplicação do parâmetro top-p ($p=0,95$) na geração de texto	34
Figura 8 - Impacto da temperatura na distribuição de probabilidades em modelos de linguagem	37
Figura 9 - Distribuição de probabilidade em função da temperatura	39
Figura 10 - Treinamento de um grande modelo de linguagem como <i>Transformer</i>	41
Figura 11 - Ciclo de <i>design</i> e aplicação de IA Generativa	51
Figura 12 - Exemplo de <i>Prompt</i>	56
Figura 13 - Exemplo de <i>prompt</i> sem formatação	58
Figura 14 - Exemplo de <i>prompt</i> com formatação	58

Figura 15 - <i>Prompt</i> incorreto	60
Figura 16 - <i>Prompt</i> correto	60
Figura 17 - Exemplos de <i>Prompts</i> correto e incorreto	61
Figura 18 - Avaliação da qualidade de saída do sistema	62
Figura 19 - Principais componentes da estratégia de testes em engenharia de <i>prompts</i>	64
Figura 20 - Criação de um <i>prompt</i> para geração de resumos de textos longos	67
Figura 21 - Desenvolvimento de um assistente virtual para responder perguntas complexas	68
Figura 22 - Prompt Engineering	75
Figura 23 - Ilustração do aprendizado em contexto, onde <i>Large Language Models</i> usam <i>prompts</i> com exemplos para fazer previsões	114
Figura 24 - Cenário de exemplo de <i>few-shot learning</i>	115
Figura 25 - Estágios de desenvolvimento de um <i>Large Language Model</i>	117
Figura 26 - Ajuste fino de um grande modelo de linguagem utilizando bases de dados privadas (conjunto de dados de um problema real)	120
Figura 27 - Reparametrização utilizando a Técnica <i>Low-Rank Adaptation</i> (LoRA)	122
Figura 28 - Comparativo da quantidade de parâmetros treináveis por camada no <i>fine-tuning</i> do Llama 3.1 na versão com 3 bilhões de parâmetros, utilizando <i>Low-Rank Adaptation</i> (LoRA) com rank 128	123
Figura 29 - Vantagens e desafios do <i>fine-tuning</i>	124
Figura 30 - Exemplo de Cadeia de Raciocínio em Modelos de Linguagem	130
Figura 31 - <i>Feature-based vs Fine Tuning</i>	142
Figura 32 - Histograma dos caracteres do dataset	154
Figura 33 - Loss ao longo do treinamento	164
Figura 34 - Visão geral de como funciona o <i>Retrieval Augmentation Generation</i> (RAG) com três etapas	181

Figura 35 - Exemplo de cenário com uso de <i>Retrieval Augmentation Generation</i> (RAG)	182
Figura 36 - Comparação entre os três paradigmas do <i>Retrieval Augmentation Generation</i> (RAG)	184
Figura 37 - Pipeline RAG Básico	189
Figura 38 - Arquitetura simplificada de memória da <i>Central Processing Units</i> (CPU) e <i>Graphics Processing Units</i> (GPU)	228
Figura 39 - Quatro abordagens para colocar <i>Large Language Models</i> em produção	230
Figura 40 - Matriz de otimização da acurácia de <i>Large Language Models</i>	237
Figura 41 - Boas práticas para garantir que o <i>Large Language Model</i> funcione de maneira eficaz	239
Tabela 1 - Comparativo entre respostas do modelo <i>Base</i> com o modelo <i>Instruct</i>	118



Sumário

Apresentação	14
Unidade I - Modelos de Linguagem e Grandes Modelos de Linguagem	15
1.1 Modelos de Linguagem	16
1.1.1 Conexão entre os Modelos de Linguagem	17
1.1.2 Modelos de Linguagem Probabilísticos	20
1.1.2.1 Aplicações	21
1.1.3 Modelos de Linguagem Neurais	22
1.1.3.1 Aplicações	23
1.1.4 Modelos de Linguagem Pré-Treinados	23
1.1.4.1 Aplicações	25
1.2 Grandes Modelos de Linguagem	26
1.2.1 Grandes Modelos de Linguagem com Transformers	27
1.2.2 Geração de Texto por Amostragem em Arquiteturas Transformer	30
1.2.2.1 Amostragem Top-k	31
1.2.2.2 Amostragem Nucleus ou Top-p	33
1.2.2.3 Temperatura	35
1.3 Treinamento de Grandes Modelos de Linguagem	40
1.3.1 Relação entre Número de Tokens , Tamanho do Modelo e Custo de Treinamento	43
1.3.2 Corpora de Treinamento para Grandes Modelos de Linguagem	45
1.4 Desafios dos Grandes Modelos de Linguagem	46
Unidade II - Engenharia de Prompts	50
2.1 Introdução à Engenharia de Prompts	51
2.2 Princípios de Engenharia de Prompts	55

2.2.1 Forneça a Direção	55
2.2.2 Especifique o Formato	56
2.2.3 Forneça Exemplos	59
2.2.3.1 Diversidade de Casos	59
2.2.3.2 Detalhe e Clareza	60
2.2.4 Avalie a Qualidade	61
2.2.5 Faça a Divisão do Trabalho	66
2.2.5.1 Exemplos de Aplicação da Divisão do Trabalho	66
2.2.5.2 Por que Dividir o Trabalho Ajuda a IA?	68
Notebook Colab	69

Unidade III - In-context Learning e Fine-tuning em Large Language Models 112

3.1 In-context Learning 113

3.1.1 Desafios e Oportunidades do *In-context Learning* 116

3.2 Fine-tuning em Large Language Models 117

3.2.1 Métodos de *Fine-tuning* 121

3.2.2 Vantagens e Desvantagens do *Fine-tuning* 123

Notebook Colab 124

Unidade IV - Retrieval Augmentation Generation 179

4.1 Como Funciona o Retrieval Augmentation Generation? 180

4.2 Arquiteturas Retrieval Augmentation Generation 183

4.2.1 Arquitetura *Naive* 184

4.2.2 Arquitetura Avançada 185

4.2.3 Arquitetura Modular 185

4.3 Retrieval Augmented Generation vs. Fine-tuning 186

4.4 Benefícios do Retrieval-Augmented Generation 187

4.5 Desafios e Limitações	187
---------------------------	-----

Notebook Colab	188
----------------	-----

Unidade V - Integração e Escalabilidade de Soluções com <i>Large Language Models</i>	225
--	-----

5.1 Escalabilidade de Soluções <i>Large Language Models</i>	226
---	-----

5.2 Capacidade Computacional para Adoção de <i>Large Language Models</i> nas Empresas	227
---	-----

5.2.1 Métricas de Capacidade Computacional para <i>Large Language Models</i>	227
--	-----

5.2.2 Implantação do <i>Large Language Model</i>	229
--	-----

5.2.3 Requisitos de GPU para Implantação de um <i>Large Language Models</i>	
---	--

em uma Infraestrutura Local	232
-----------------------------	-----

5.3 Otimização de Latência e Acurácia	234
---------------------------------------	-----

5.3.1 Otimização de Latência dos <i>Large Language Models</i>	234
---	-----

5.3.2 Otimização da Acurácia dos <i>Large Language Models</i>	236
---	-----

5.4 Integração de Soluções <i>Large Language Models</i>	238
---	-----

5.5 Disponibilizar Soluções <i>Large Language Models</i> para Usuários Finais	239
---	-----

5.6 Estudos de Caso e Exemplos Práticos	240
---	-----

5.6.1 Chatbots para Atendimento ao Cliente	240
---	-----

5.6.2 Personalização de Conteúdo com <i>Large Language Models</i>	241
---	-----

5.6.3 Tradução Automática	242
---------------------------	-----

Unidade VI - Encerramento	245
---------------------------	-----

6.1 Sugestões para Ação Futura e Leitura Adicional	247
--	-----

6.2 Leitura Adicional	248
-----------------------	-----

6.3 Reflexões Finais	248
----------------------	-----

Referências	249
-------------	-----

Apresentação

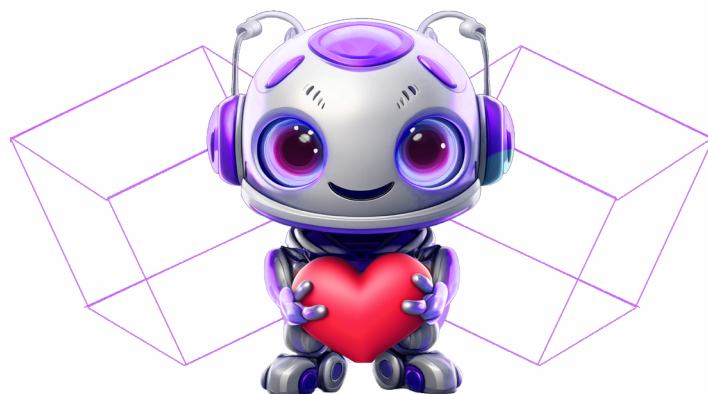
Prezado(a) Participante,

Seja bem-vindo(a) ao Microcurso Grandes Modelos de Linguagem. Este Microcurso faz parte da Coleção Formação e Capacitação do Centro de Competências Imersivas, uma parceria entre a Embrapii e a Universidade Federal de Goiás (UFG).

A oferta deste Microcurso é motivada pela crescente importância dos Grandes Modelos de Linguagem (*Large Language Models* [LLMs]) no cenário da Inteligência Artificial (IA) e suas diversas aplicações em áreas como Processamento de Linguagem Natural (PLN), desenvolvimento de chatbots, tradução automática e muito mais.

O objetivo deste Microcurso é fornecer a você uma base sobre o funcionamento dos LLMs, suas capacidades e limitações. Abordaremos desde os conceitos básicos, como arquiteturas de modelos e técnicas de treinamento, até tópicos mais avançados, como a construção de *prompts* eficazes, a integração de LLMs em sistemas existentes e as questões de escalabilidade.

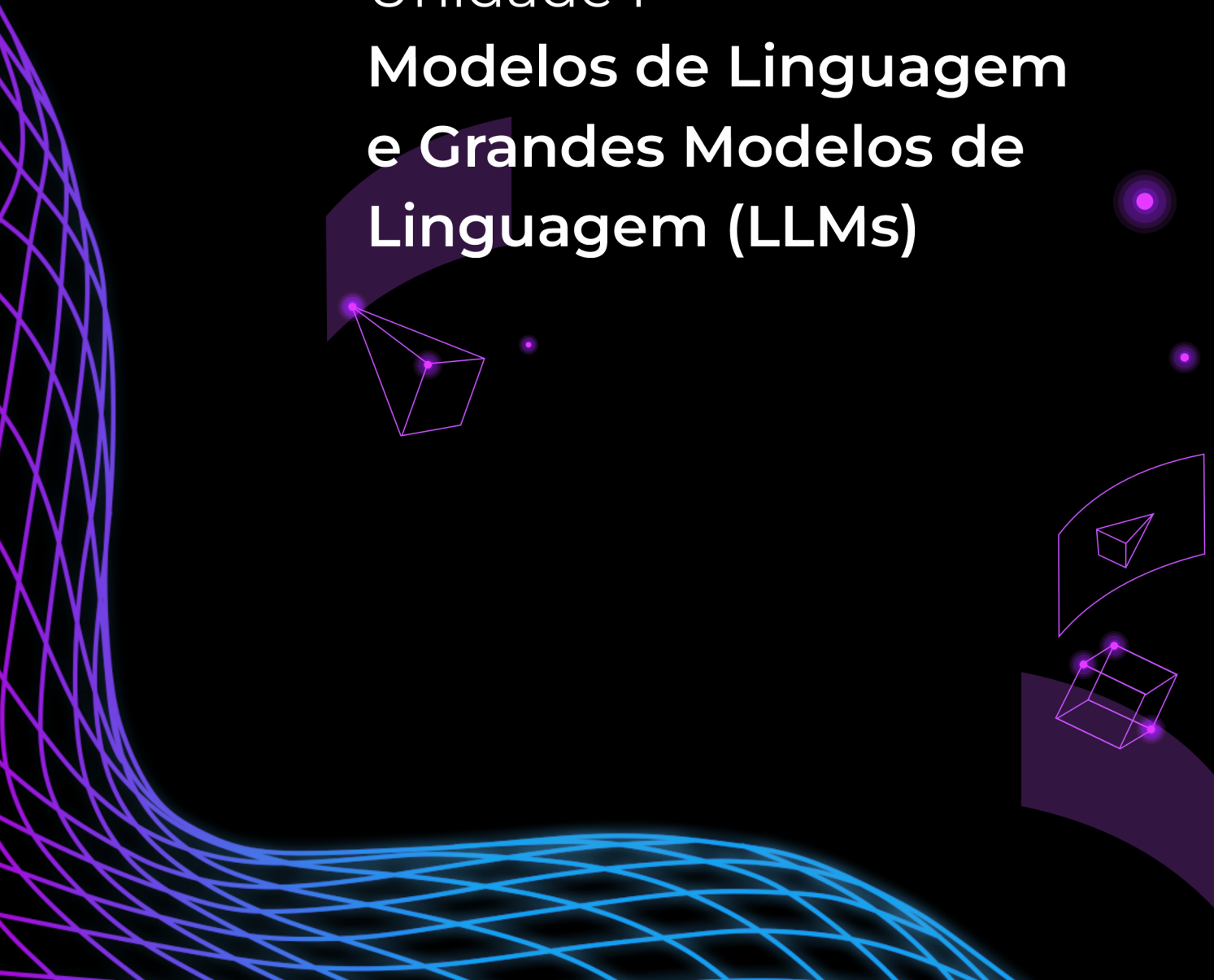
Você terá a oportunidade de aprender com exemplos práticos, exercícios e *notebooks* interativos, que irão auxiliar a dominar as ferramentas e técnicas para utilizar LLMs em seus próprios projetos.

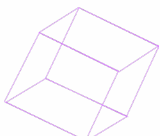
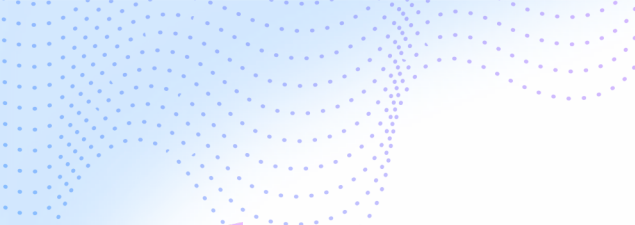


Desejamos um excelente estudo!

Unidade I

Modelos de Linguagem e Grandes Modelos de Linguagem (LLMs)





Unidade I - Modelos de Linguagem e Grandes Modelos de Linguagem



1.1 Modelos de Linguagem

A tarefa de prever o futuro é muito complicada. Mas e prever algo que pareça mais fácil, como as próximas palavras que alguém dirá? Qual palavra, por exemplo, provavelmente seguirá a frase "Por favor, entreguem seus trabalhos de ..."? Talvez, a maioria de vocês concluiu que uma palavra muito provável é "casa", mas não "geladeira".

Nesta seção, formalizamos essa intuição apresentando modelos que atribuem uma probabilidade a cada próxima palavra possível. Modelos que atribuem probabilidades a palavras ou sequências de palavras futuras são chamados de modelos de linguagem ou *Language Models* (LMs). Por que desejaríamos prever as próximas palavras? Os LLMs são treinados apenas prevendo palavras! Como veremos ao longo deste livro, eles aprendem muito sobre linguagem apenas sendo treinados para prever as próximas palavras a partir de palavras vizinhas.

Os LLMs também podem calcular a probabilidade de uma frase inteira. Por exemplo, a frase "O gato deitou-se sobre o tapete macio" possui uma probabilidade muito maior de aparecer em um texto do que a mesma sequência de palavras em uma ordem diferente: "Macio tapete o gato deitou-se sobre".

Por que importa qual a probabilidade de uma frase ou quão provável é a próxima palavra? Em muitas aplicações de PLN, podemos usar a probabilidade como uma forma de escolher uma frase ou palavra melhor em vez de uma menos apropriada. Por exemplo, um sistema de tradução automática que precisa escolher entre "O gato sentou sobre o tapete" e "O gato assentou sobre o tapete". A primeira frase, sendo mais comum e natural em português, terá uma probabilidade maior, guiando o sistema para a tradução correta. Da mesma forma, em um corretor ortográfico, a palavra "cachorro" terá uma probabilidade muito maior do que "caxorro", permitindo a correção automática do erro. A análise de probabilidade também é aplicada em assistentes virtuais, prevendo a próxima palavra mais provável em uma frase e oferecendo sugestões ao usuário durante a digitação.

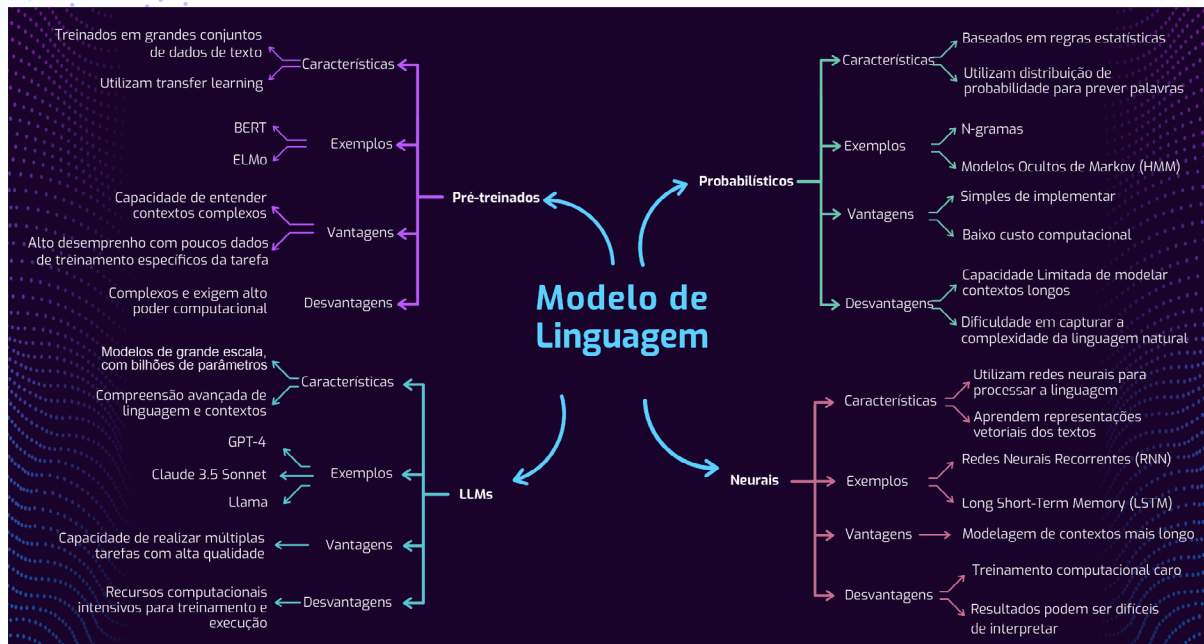
Em termos mais técnicos, um LLM é um algoritmo treinado em um conjunto massivo de dados textuais para prever e gerar texto. Ele pode ser visto como um super-leitor que, após ler uma grande quantidade de documentos textuais, pode adivinhar a próxima palavra de uma frase ou até escrever um texto inteiro com base em algumas palavras-chave. Mas o que torna os LLMs tão importantes? É a capacidade de comunicar e gerar linguagem similar à humana. No nosso dia a dia, os LMs já estão presentes em:

- » **Assistentes virtuais:** Siri®, Alexa® e Google Assistente® utilizam LMs para entender comandos de voz e responder às perguntas dos usuários.
- » **Tradutores automáticos:** plataformas como o Google Tradutor® usam LMs para traduzir textos de forma cada vez mais precisa e natural.
- » **Preenchimento automático e sugestões de texto:** ao escrever um email ou mensagem, os LMs predizem a próxima palavra, corrigem erros ortográficos e até sugerem respostas completas.
- » **Geração de conteúdo:** *blogs* e *sites* podem ser, em parte, escritos por LMs, que criam textos criativos e informativos.
- » **Análise de sentimentos:** empresas utilizam LMs para analisarem comentários e menções nas redes sociais, identificando a opinião do público sobre seus produtos e serviços.

1.1.1 Conexão entre os Modelos de Linguagem

Para melhor compreender a evolução e a diversidade desses modelos, na Figura 1 é apresentado um mapa mental que funciona como um guia visual. Nele, exploramos desde os modelos mais básicos, como os N-gramas, até o estado da arte com os LLMs, passando pelos modelos probabilísticos e modelos pré-treinados, destacando suas características distintivas e funcionalidades. No mapa mental, são organizados diferentes tipos de LMs de forma hierárquica, classificando-os em quatro principais categorias: modelos probabilísticos, modelos neurais, modelos pré-treinados e LLMs.

Figura 1 - Mapa mental que ilustra a evolução e a taxonomia dos modelos de linguagem



Fonte: autoria própria.

Modelos de Linguagem Probabilísticos (MLPs) são caracterizados por serem baseados em regras estatísticas, utilizando distribuições de probabilidades para prever palavras em uma sequência de texto. Exemplos incluem N-gramas e Modelos Ocultos de Markov (HMM). Esses modelos têm como vantagem a simplicidade de implementação e baixo custo computacional, mas possuem limitações na modelagem de contextos longos e na captura da complexidade da linguagem natural.

Modelos Neurais utilizam redes neurais para processar a linguagem, aprendendo representações vetoriais dos textos. Alguns exemplos são as Redes Neurais Recorrentes (RNN) e *Long Short-Term Memory* (LSTM). Esses modelos têm a capacidade de modelar contextos mais longos, o que melhora sua capacidade de entender a linguagem. Entretanto, seu treinamento é computacionalmente caro e os resultados podem ser difíceis de interpretar.

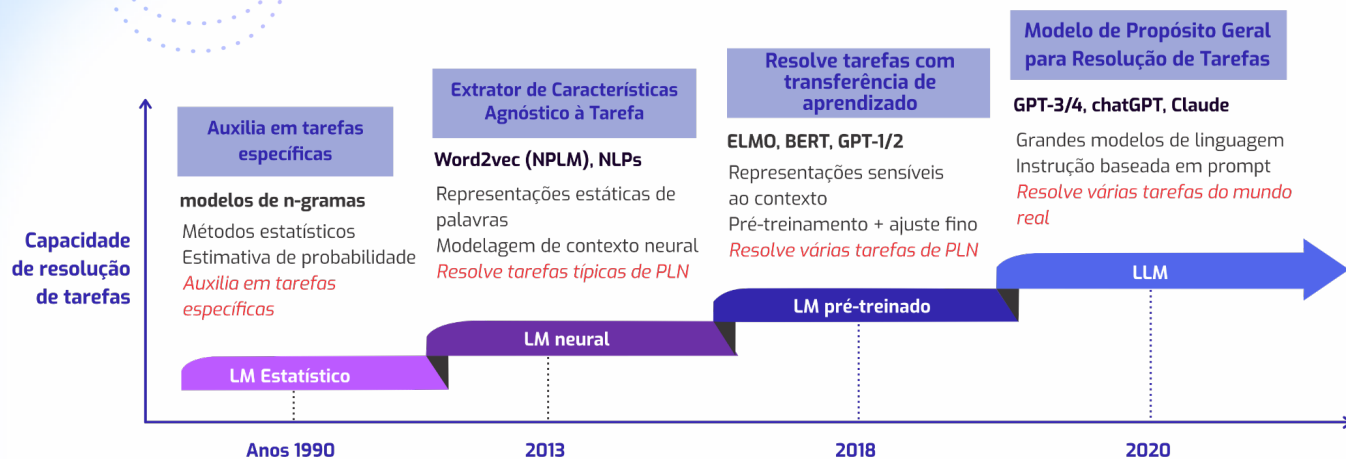
Modelos Pré-treinados são treinados em grandes conjuntos de dados de texto e utilizam técnicas de transferência de aprendizado (*transfer learning*). Exemplos incluem *Bidirecional Encoder Representations from Transformers* (BERT) e *Embeddings from Language Model* (ELMo), que são conhecidos por conseguir processar contextos complexos e pelo alto desempenho em tarefas específicas com poucos dados de treinamento adicionais. Esses modelos, entretanto, são complexos e exigem alto poder computacional.

LLMs são caracterizados por sua escala massiva, com bilhões de parâmetros, o que lhes confere uma compreensão avançada de linguagem e contextos. O GPT-4, Claude 3.5, Mistral e Llama são exemplos de famílias de modelos que são classificados como LLMs. Esses modelos têm a capacidade de realizar múltiplas tarefas com alta qualidade, como geração de texto, tradução automática e sumarização de texto. Contudo, sua desvantagem reside no alto custo computacional necessário tanto para o treinamento quanto para a execução do modelo em produção. Um estudo revelou que o ChatGPT pode consumir 1 litro de água a cada 100 palavras geradas pela ferramenta. A água é utilizada como um mecanismo para resfriar os supercomputadores que rodam o GPT (Li *et al.*, 2023).

Na Figura 2, é apresentada a evolução dos LMs ao longo do tempo, desde os anos 1990 até os dias atuais. No início, os modelos eram baseados em métodos estatísticos, como os modelos *n-gram*, que serviam como auxiliares em tarefas específicas, realizando estimativas de probabilidade para resolver problemas limitados. Esses modelos possuíam uma capacidade de resolução de tarefas muito limitada, sendo utilizados apenas em tarefas específicas dentro do domínio da linguagem.

Com o avanço da tecnologia, surgiram modelos mais avançados, como o Word2Vec e outros modelos baseados em redes neurais, que foram categorizados como modelos de linguagem neurais (MLN). Esses modelos eram capazes de aprender representações de palavras de maneira estática e realizar modelagem de contexto neural, permitindo a resolução de tarefas típicas de PLN de forma mais generalizada. Mais tarde, em 2018, a introdução dos modelos pré-treinados, como ELMo, BERT, e as primeiras versões do GPT, trouxe uma mudança radical ao permitir a transferência de aprendizado entre diferentes tarefas de PLN, resolvendo uma variedade de tarefas com o uso de pré-treinamento e *fine-tuning*. Finalmente, a evolução dos modelos resultou na criação dos LLMs, como o GPT-4, que são capazes de resolver diversas tarefas do mundo real, utilizando escalabilidade e *prompts* para completar tarefas complexas.

Figura 2 - Evolução da capacidade de resolução de tarefas em quatro gerações de Modelos de Linguagem



Fonte: adaptada de Zhao et al. (2023).

1.1.2 Modelos de Linguagem Probabilísticos

Antes do surgimento das redes neurais profundas e da aprendizagem profunda, o PLN se apoiava em MLP. Suponha que você esteja tentando adivinhar a próxima palavra em uma frase. O que você faria? Provavelmente, se basearia nas palavras que já foram ditas, certo? É exatamente assim que funcionam os MLP. Eles aprendem a prever a próxima palavra em uma sequência, analisando um grande conjunto de textos e identificando padrões de linguagem.

Eles podem ser vistos como detetives da linguagem que investigam textos em busca de pistas sobre quais palavras costumam aparecer juntas e com que frequência. Essas pistas são traduzidas em probabilidades, que indicam a chance de uma determinada palavra aparecer após outra.

Um conceito central nos MLP é a probabilidade de uma sequência de palavras, que determina a chance de uma sequência específica aparecer em um determinado contexto. Por exemplo, um modelo treinado em um *corpus* de notícias seria capaz de determinar que a sequência "O presidente anunciou um novo" tem maior probabilidade de ser seguida pela palavra "decreto" do que pela palavra "sorvete".

Um dos modelos probabilísticos mais simples e eficazes são os N-gramas que calculam a probabilidade de uma palavra ocorrer com base nas $n-1$ palavras anteriores. Modelos populares como unigramas (consideram apenas a palavra atual), bigramas (consideram a palavra atual e a anterior) e trigramas (consideram as duas palavras anteriores e a atual) são amplamente utilizados em diversas tarefas de PLN, como

correção ortográfica e predição de texto (Jurafsky; Martin, 2024). A intuição de modelos N-gramas é que em vez de calcular a probabilidade de uma palavra dado o histórico de palavras anteriores, nós queremos aproximar o histórico utilizando apenas as últimas palavras.

1.1.2.1 Aplicações

Os MLP são interconectados e podem ser combinados para criar sistemas. Por exemplo, modelos N-gramas podem ser usados como base para modelos de tópicos, fornecendo informações contextuais sobre as palavras. LMs podem ser integrados a sistemas de tradução automática, usando sua capacidade de analisar a estrutura sintática para realizar traduções mais precisas. As aplicações dos MLP abrangem diversas áreas, como:

- » **Previsão de texto:** implementados em teclados de *smartphones* e editores de texto para sugerir a próxima palavra ou frase.
- » **Tradução automática:** tradução de textos de uma língua para outra, utilizando LMs para modelar a estrutura das línguas envolvidas.
- » **Análise de sentimentos:** identificar a opinião expressa em um texto, seja ela positiva, negativa ou neutra.
- » **Classificação de documentos:** auxiliam na categorização automática de textos em tópicos predefinidos.

Apesar de sua utilidade, os MLP enfrentam vários desafios. Um dos principais problemas é a maldição da dimensionalidade: à medida que o tamanho do vocabulário e a complexidade das relações entre as palavras aumentam, a quantidade de dados necessários para treinar um modelo preciso cresce exponencialmente. Além disso, os MLP têm dificuldade em capturar o significado de longo alcance e o contexto de uma frase, limitando sua capacidade de lidar com tarefas mais complexas de compreensão da linguagem.

Apesar dessas limitações, os MLPs são a base para o desenvolvimento da área de PLN. As técnicas e os resultados obtidos com esses modelos continuam a ser utilizados em conjunto com as abordagens mais recentes, como as arquiteturas *Transformers*.

1.1.3 Modelos de Linguagem Neurais

Os MLN (Turney; Pantel, 2010; Collobert *et al.*, 2011; Mikolov *et al.*, 2010) definem a probabilidade de sequências de palavras por meio de redes neurais, por exemplo, *perceptron* multicamadas e RNN. Os MLN possuem muitas vantagens sobre os MLP, já que podem lidar com sentenças e documentos muito mais longos, podem generalizar melhor sobre contextos de palavras similares e serem mais precisos na predição de palavras. Por outro lado, MLN são muito mais complexos, mais lentos e precisam de mais recursos computacionais para treinar, além de serem menos interpretáveis do que MLP. Por isso, para algumas tarefas menores, um MLP ainda é a ferramenta certa.

Os MLN foram desenvolvidos para construir uma solução unificada e de ponta a ponta para várias tarefas de PLN (Bengio; Ducharme; Vincent, 2000). Um dos principais modelos, denominado *Word2vec* (Kombrink *et al.*, 2011; Mikolov *et al.*, 2013) foi proposto para construir uma rede neural simplificada para aprender representações distribuídas de palavras, que se mostraram muito eficazes em uma variedade de tarefas de PLN. Esses estudos iniciaram o uso de MLN para aprendizado de representação, tendo um impacto importante no campo de PLN.

Os MLN representam palavras por meio de *embeddings*, o que permite que MLN generalizem melhor que MLP para dados não vistos. Por exemplo, suponha que o LM recebeu esta frase na etapa de treinamento:

Eu tenho que garantir que o gato seja alimentado.

Vamos supor que, na etapa de treinamento, o modelo não recebeu exemplos com as palavras "seja alimentado" depois da palavra "cão". Nosso conjunto de testes possui o prefixo "Eu esqueci de garantir que o cão seja". Qual é a próxima palavra? Um LM N-grama irá prever "alimentado" depois de "que o gato seja", mas não depois de "que o cão seja". Mas um MLN, sabendo que "gato" e "cão" possuem *embeddings* similares, será capaz de generalizar a partir do contexto "gato" para atribuir uma probabilidade alta o suficiente para "alimentado" mesmo depois de ver "cão".

1.1.3.1 Aplicações

Os MLN, como redes recorrentes (RNN) e memória de longo prazo e curto prazo (LSTM), têm sido utilizados em diversas aplicações de PLN no mundo real. Algumas das principais aplicações desses modelos incluem:

1. **Classificação de texto:** as capacidades de processamento de linguagem das RNNs e LSTMs permitem utilizá-las para a classificação de textos em diferentes categorias, como sentimento, tópico, spam etc.
2. **Geração de diálogo:** criação de agentes conversacionais capazes de manter diálogos fluidos com seres humanos.
3. **Deteção de anomalias em texto:** as LSTMs podem ser usadas para detectar anomalias e desvios em textos, como identificar fraudes em documentos financeiros ou detectar conteúdo inadequado em postagens online.
4. **Extração de informações:** podem ser treinados para extrair informações estruturadas, como nomes de entidades, relações e fatos, a partir de textos não estruturados.

As redes neurais são treinadas em grandes conjuntos de dados de texto para aprender padrões, relações semânticas e até mesmo nuances de estilos da linguagem. Mas, os limites dos MLN estão se tornando cada vez maiores com o surgimento de novos modelos, como o *Transformer*, que utiliza mecanismos de atenção para capturar relacionamentos de longo prazo entre palavras, superando algumas das limitações de RNNs e LSTMs.

1.1.4 Modelos de Linguagem Pré-Treinados

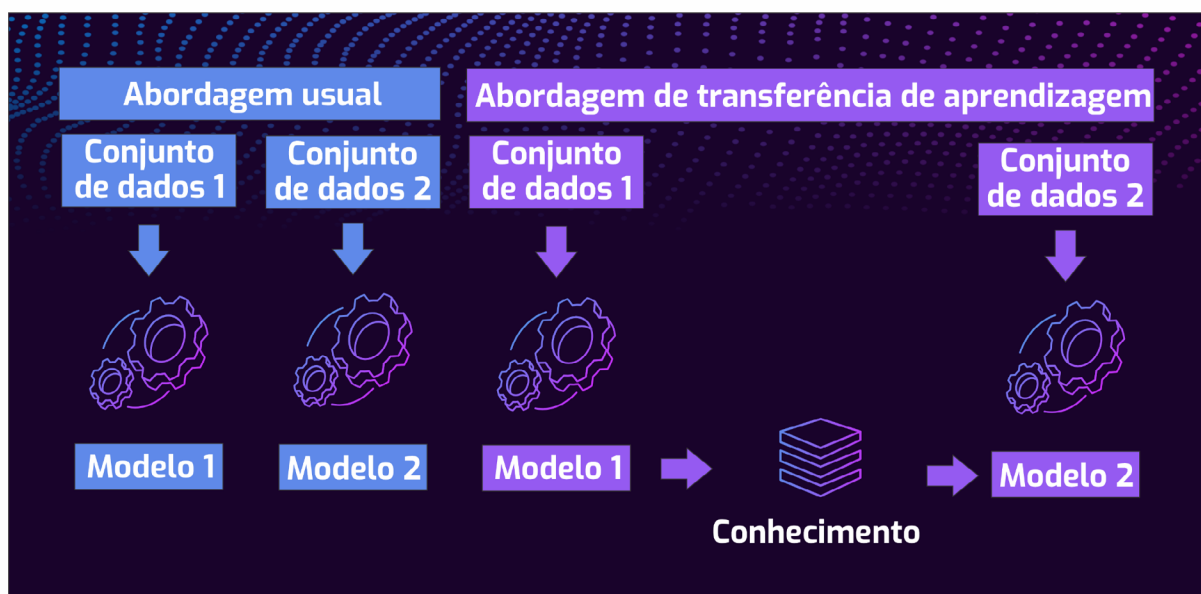
Os LM pré-treinados são sistemas de IA que aprendem a compreender e gerar texto a partir da análise de grandes conjuntos de dados. Esses modelos são treinados em uma ampla variedade de textos, como livros, artigos, *websites* e diálogos, adquirindo um conhecimento profundo sobre a estrutura, o significado e o uso da linguagem natural. Devido ao seu treinamento em larga escala, esses modelos requerem uma enorme quantidade de recursos computacionais, como poder de processamento com *Graphics Processing Units* (GPUs) ou *Tensor Processing Units* (TPUs) e armazenamento.

Uma das principais características desses modelos é a sua capacidade de transferência de aprendizado. Após serem pré-treinados em grandes conjuntos de dados genéricos, eles podem ser ajustados rapidamente a tarefas e domínios específicos, usando apenas uma fração dos dados e dos recursos computacionais necessários para construir um modelo do zero.

Na Figura 3, são comparadas duas abordagens de treinamento de modelos de aprendizado de máquina. Na abordagem usual, dois modelos distintos são treinados separadamente com diferentes conjuntos de dados, resultando em dois modelos específicos e independentes. Já na abordagem de transferência de aprendizado, um modelo é inicialmente treinado com um conjunto de dados (conjunto de dados 1), e o conhecimento adquirido é transferido para um segundo modelo, que é então refinado com um novo conjunto de dados (conjunto de dados 2). Isso permite que o segundo modelo aproveite o aprendizado anterior, acelerando o treinamento e melhorando o desempenho em tarefas específicas. Essa técnica é muito útil em LLMs, onde modelos pré-treinados em grandes volumes de dados são ajustados para aplicações especializadas por meio de um processo denominado ***fine-tuning*** ou **ajuste fino**.

Nessa comparação entre a abordagem usual de treinamento de modelos, onde cada modelo é treinado do zero com um conjunto de dados específico, e a abordagem de *Transfer Learning*, o conhecimento de um modelo pré-treinado é transferido para outro modelo, permitindo um treinamento especializado.

Figura 3 - Comparação entre a abordagem usual de treinamento de modelos



Fonte: adaptada de Singh (2024).

Fine-tuning é um processo em que um modelo de IA, que já foi treinado em uma grande quantidade de dados gerais, é ajustado para realizar uma tarefa específica. Por exemplo, um cozinheiro pode saber fazer de tudo um pouco, mas deseja se tornar um especialista em fazer pizzas. Ele já sabe cozinhar, então ele não precisa aprender do zero, mas desenvolver as habilidades para fazer a melhor pizza possível. Esse treinamento adicional, focado em pizzas, é como o *fine-tuning*.

No contexto de um modelo de IA, o "cozinheiro" é o modelo pré-treinado, e o "treinamento para fazer pizzas" é o processo de *fine-tuning*. Esse ajuste é feito usando um novo conjunto de dados que é mais relevante para a tarefa específica que você quer que o modelo realize. Por exemplo, se o modelo original foi treinado com textos gerais da internet, o *fine-tuning* poderia ser feito com textos médicos, se o objetivo for criar um modelo para auxiliar em diagnósticos médicos. Assim, o modelo ajustado consegue desempenhar a nova tarefa de forma mais precisa do que se tivesse que treiná-lo do zero apenas com os dados específicos.

Nos últimos anos, vários LMs pré-treinados têm se destacado. O ELMo (Peters *et al.*, 2018), por exemplo, foi uma das primeiras propostas para capturar representações de palavras com reconhecimento de contexto, usando uma rede LSTM bidirecional. Já o BERT (Devlin *et al.*, 2019), baseado na arquitetura *Transformer* (Vaswani, 2017), trouxe avanços ao pré-treinar LMs bidirecionais em larga escala. Seguindo o paradigma de "pré-treinamento e ajuste fino", outros modelos como o GPT-2 (Radford *et al.*, 2019) e o BART (Fedus; Zoph; Shazeer, 2022) também foram propostos, introduzindo diferentes arquiteturas ou estratégias de pré-treinamento. Esses LMs pré-treinados tornaram-se referência na área, cada qual com suas características e funcionalidades particulares e são a base para a criação dos LLMs.

1.1.4.1 Aplicações

Os LMs pré-treinados impulsionaram avanços em diversas áreas, tais como:

- » **Geração de texto:** o GPT, com sua capacidade de gerar textos coerentes e criativos, se destaca nesse quesito, sendo utilizado em *chatbots*, criação de conteúdo e ferramentas de escrita.
- » **Tradução automática:** modelos como o BERT e GPT podem ser adaptados para tarefas de tradução, utilizando sua compreensão da linguagem para gerar traduções mais precisas e fluentes.
- » **Sumarização de texto:** LMs pré-treinados podem ser utilizados para extrair as informações principais de um texto, gerando resumos concisos e informativos.

- » **Análise de sentimento:** o ELMo e o BERT, com sua capacidade de entender o contexto das palavras, podem ser utilizados para analisar o sentimento de um texto, identificando se ele expressa sentimentos positivos, negativos ou neutros.

Os LLMs, como GPT-4, se baseiam em princípios e técnicas dos modelos pré-treinados, mas possuem tamanhos e capacidades ainda maiores. Esses modelos representam a vanguarda da pesquisa em PLN, abrindo novas possibilidades para a interação humano-computador com a criação dos LLMs.

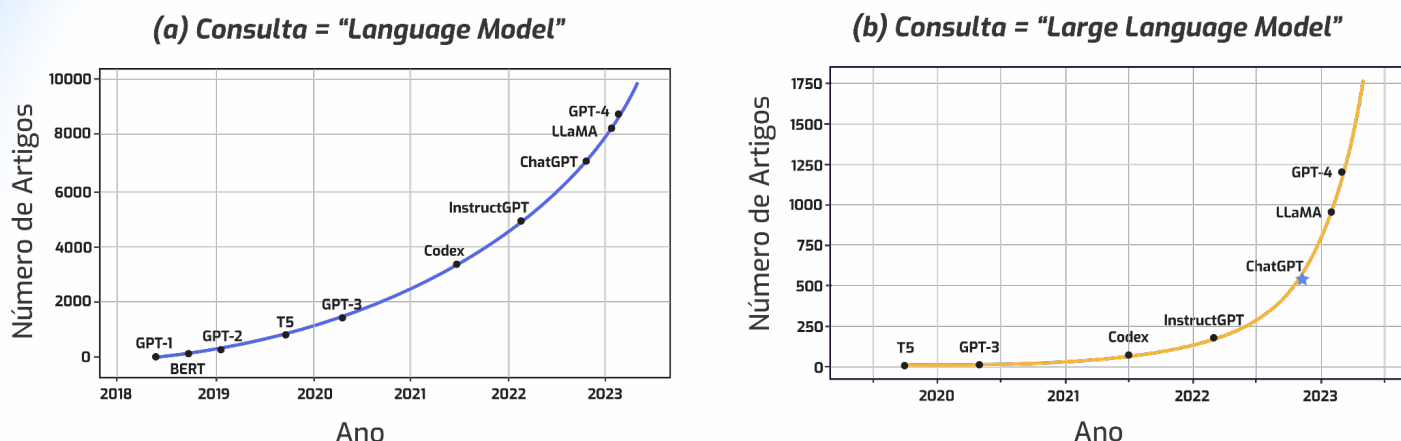
1.2 Grandes Modelos de Linguagem

Pesquisadores descobriram que a ampliação dos LMs pré-treinados (por exemplo, a ampliação do tamanho do modelo ou do tamanho dos dados) muitas vezes leva a uma capacidade de modelo aprimorada em tarefas específicas (seguindo a lei da escalabilidade de Kaplan *et al.*, 2020). Vários estudos exploraram o limite de desempenho treinando um modelo pré-treinado cada vez maior (por exemplo, o GPT-3 com 175 bilhões de parâmetros e o PaLM com 540 bilhões de parâmetros). Embora o escalonamento seja conduzido principalmente no tamanho do modelo (com arquiteturas e tarefas de pré-treinamento semelhantes), esses LLMs apresentam comportamentos diferentes dos modelos menores (por exemplo, BERT com 330 milhões de parâmetros e GPT-2 com 1,5 bilhão de parâmetros) e demonstram habilidades impressionantes, denominadas habilidades emergentes (Wei *et al.*, 2022) ao resolver uma série de tarefas complexas. Por exemplo, o GPT-3 pode resolver tarefas que o GPT-2 não consegue fazer de forma satisfatória.

O termo "grandes modelos de linguagem" refere-se a LMs pré-treinados de grande porte (Hoffmann *et al.*, 2022; Taylor *et al.*, 2022; Wei *et al.*, 2022; Shanahan, 2024). Na Figura 4, é mostrado como esses modelos atraem cada vez mais atenção da pesquisa com o aumento acentuado de artigos do arXiv relacionados a LLMs após o lançamento do ChatGPT.

As estatísticas são calculadas usando correspondência exata ao consultar as frases-chave no título ou resumo por meses. Os marcos importantes com o lançamento de novos LLMs estão indicados nos gráficos.

Figura 4 - Tendências dos números cumulativos de artigos do arXiv que contêm as frases-chave "*language model*" (desde junho de 2018) (a) e "*large language model*" (desde outubro de 2019) (b)



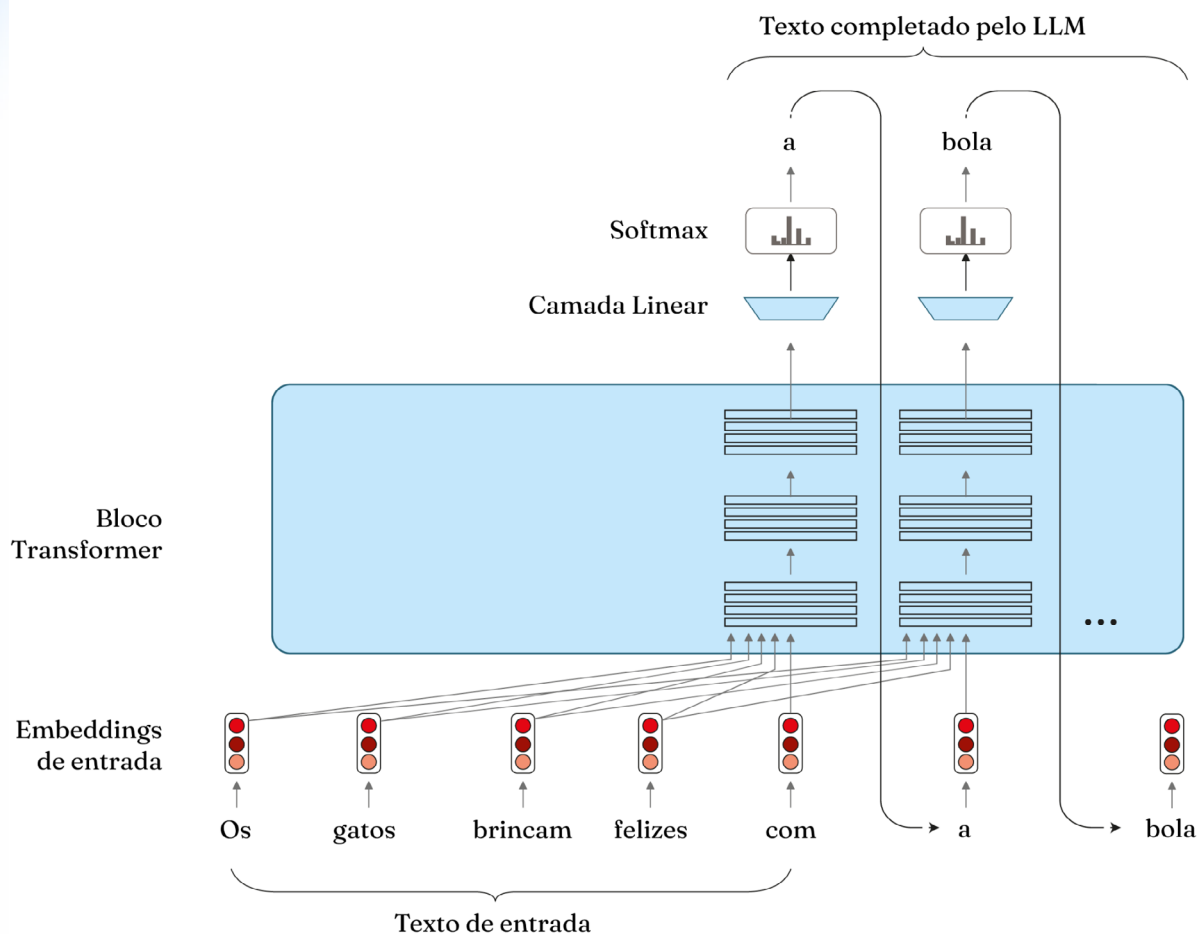
Fonte: adaptada de Zhao (2023).

1.2.1 Grandes Modelos de Linguagem com *Transformers*

A arquitetura *Transformer* foi a propulsora da evolução da área de PLN. Diferente de modelos anteriores, como as RNNs, a arquitetura *Transformer* permite paralelizar o processamento e lidar com sequências de texto muito longas. Essa arquitetura permitiu o desenvolvimento de LLMs capazes de gerar texto de forma condicional, com base em um *prompt* de entrada. A habilidade da arquitetura *Transformer* de acessar todo o contexto anterior, incluindo as próprias saídas geradas, é a base para a tarefa de geração de texto.

Considere a simples tarefa de completar a sentença "Os gatos brincam felizes com ...", ilustrada na Figura 5. Aqui, um modelo de linguagem recebe um prefixo de texto e é solicitado a gerar um possível complemento. Observe que, à medida que o processo de geração prossegue, o modelo tem acesso direto ao contexto, bem como a todas as suas próprias saídas geradas subsequentemente (pelo menos o máximo que couber na grande janela de contexto). Essa capacidade de incorporar a totalidade do contexto anterior e as saídas geradas a cada etapa é a chave para o poder dos LLMs construídos a partir da arquitetura *Transformer*.

Figura 5 - Tarefa de completar textos utilizando grandes modelos de linguagem baseados em arquitetura *Transformer*



Fonte: adaptada de Zhao (2023).

Mas, por que devemos nos preocupar em prever as próximas palavras? O diferencial da modelagem de linguagem é que muitas tarefas práticas de PLN podem ser formuladas como previsão de palavras e que um LM poderoso o suficiente pode resolvê-las com um alto grau de precisão. Por exemplo, podemos considerar a tarefa de análise de sentimentos como modelagem de linguagem, fornecendo a um LM um contexto como:

O sentimento da frase "Eu gosto de filmes de ação" é:

Devemos comparar a seguinte probabilidade condicional das palavras "positivo" e a palavra "negativo" para ver qual é maior. Se a palavra "positivo" for mais provável, dizemos que o sentimento da frase é positivo; caso contrário, dizemos que o sentimento é negativo.

Também podemos classificar tarefas mais complexas como previsão de palavras. Considere a tarefa de responder a perguntas simples, onde o sistema recebe alguma pergunta e deve dar uma resposta textual. Podemos considerar a tarefa de responder a perguntas como previsão de palavras, fornecendo a um LM uma pergunta e um *token* como 'R:' sugerindo que uma resposta deve vir a seguir:

Q: Quem escreveu o livro "A Origem das Espécies"? R:

Se pedirmos a um LM para calcular quais palavras têm probabilidades mais altas, podemos ver que Charles é muito provável. Em seguida, se escolhermos Charles e continuarmos:

Q: Quem escreveu o livro "A Origem das Espécies"? R: Charles.

Agora, podemos ver que Darwin é a palavra mais provável e podemos selecioná-la.

Quais palavras devemos gerar a cada etapa? Uma maneira simples de gerar palavras é sempre gerar a palavra mais provável, dado o contexto. Isso é chamado de decodificação gulosa. Um algoritmo guloso é aquele que faz uma escolha local ótima, quer venha a ser ou não a melhor escolha no final (Edmonds, 1971). Assim, na decodificação gulosa, a cada etapa da geração, a saída é escolhida calculando a probabilidade para cada saída possível (cada palavra no vocabulário) e, em seguida, escolhendo a palavra de maior probabilidade.

Na prática, não usamos decodificação gulosa com LLMs. Um grande problema com a decodificação gulosa é que, como as palavras que ela escolhe são (por definição) previsíveis, o texto resultante é genérico e frequentemente bastante repetitivo. Na verdade, a decodificação gulosa é tão previsível que é até determinística; se o contexto for idêntico e o modelo probabilístico for o mesmo, a decodificação gulosa sempre resultará na geração exatamente da mesma sentença. Na maioria das outras tarefas, entretanto, as pessoas preferem textos que foram gerados por métodos mais avançados, chamados métodos de amostragem, que introduzem um pouco mais de diversidade nas gerações. Veremos como fazer isso na próxima seção.

1.2.2 Geração de Texto por Amostragem em Arquiteturas *Transformer*

Ao gerar um texto com um LM, a cada passo, o modelo seleciona o próximo *token* com base nas escolhas anteriores, utilizando probabilidades dos *tokens*. Isso significa que *tokens* com alta probabilidade têm mais chances de serem escolhidas. Para facilitar o entendimento, nos próximos exemplos, iremos assumir que um *token* é uma palavra e o modelo tem como objetivo prever a próxima palavra. Se o modelo atribui uma probabilidade de 70% para a palavra "bola" e 10% para a palavra "zebra" no contexto da sentença "Os gatos brincam felizes com a ____", é mais provável que a palavra "bola" seja escolhida.

A amostragem é o processo pelo qual o modelo escolhe a próxima palavra em uma sequência com base nas probabilidades que ele calcula. Esse processo de escolher a próxima palavra é chamado de **decodificação** e acontece de forma autoregressiva, ou seja, a cada passo, escolhemos a próxima palavra com base nas palavras que já foram geradas anteriormente. Isso significa que o modelo "lê" o texto que já foi gerado e, com base nesse contexto, decide qual será a próxima palavra.

Um dos métodos mais comuns para fazer essa decodificação em LLMs é chamado de **amostragem**, introduzindo um grau de aleatoriedade na escolha das palavras. Em vez de escolher sempre a palavra com a maior probabilidade, o modelo seleciona a próxima palavra com base na distribuição de probabilidade calculada, permitindo que palavras menos prováveis sejam escolhidas ocasionalmente. Essa abordagem pode resultar em textos mais diversos e criativos, pois o modelo é induzido a explorar diferentes caminhos e possibilidades. O modelo também pode produzir resultados menos coerentes ou fora de contexto, uma vez que a escolha aleatória pode levar a combinações de palavras inesperadas ou menos lógicas.

Por isso, em vez de usar uma amostragem aleatória, existem métodos que ajudam a evitar a seleção dessas palavras muito improváveis. Uma técnica popular é o *top-k sampling*, que limita as escolhas às k palavras mais prováveis, e o *top-p sampling* (também conhecido como *nucleus sampling*), que seleciona palavras com base em um intervalo de probabilidade cumulativa. Essas abordagens ajudam a manter a qualidade do texto gerado enquanto ainda permitem alguma flexibilidade.

Nas próximas seções deste livro, vamos explorar algumas técnicas de amostragem que permitem ajustar o equilíbrio entre qualidade e diversidade na geração de texto com LMs. Essas técnicas possibilitam priorizar palavras com probabilidades mais altas (para geração de textos mais coerentes) ou dar mais chances a palavras com probabilidades medianas (para gerar textos mais criativos e diversificados). A escolha da técnica e dos parâmetros adequados dependerá do tipo de texto que se deseja gerar e dos objetivos da tarefa.

1.2.2.1 Amostragem Top-k

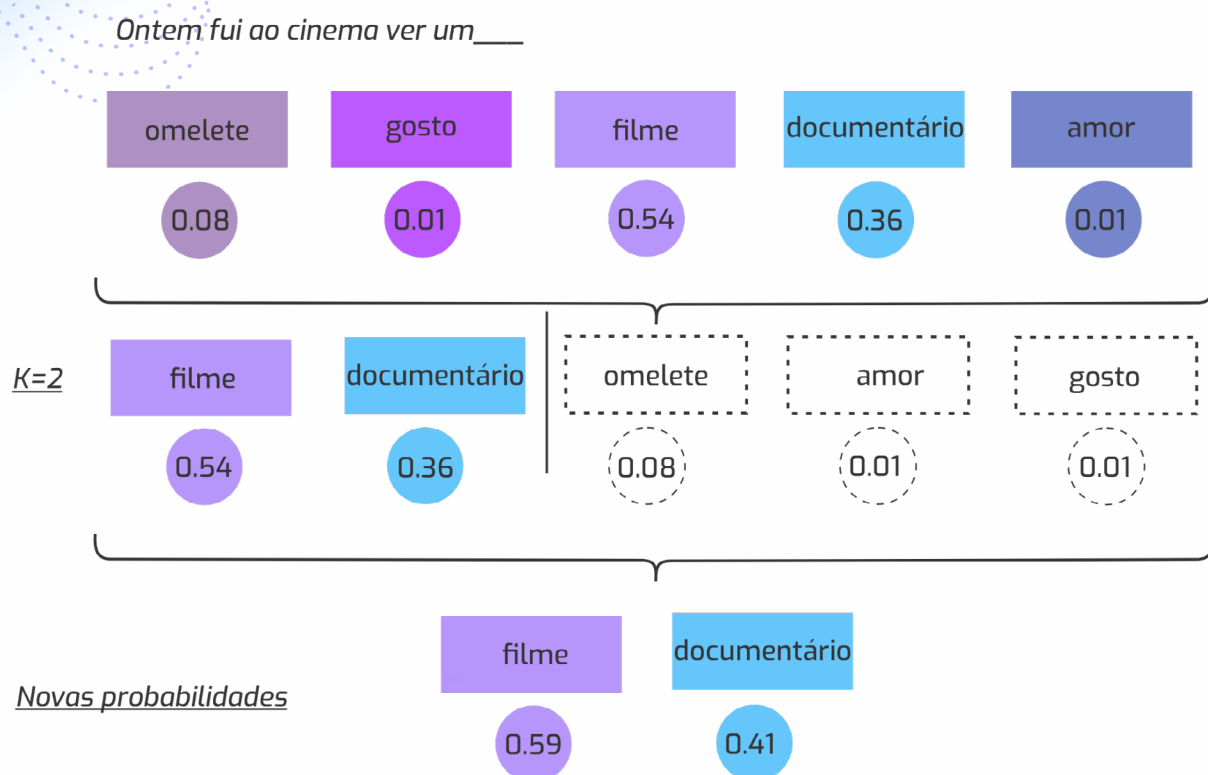
A amostragem top-k é uma técnica que visa equilibrar a previsibilidade e a criatividade na geração de texto por modelos baseados na arquitetura *Transformer*. Em vez de optar pela palavra com a maior probabilidade, como no método de decodificação gulosa, a amostragem top-k introduz um grau de aleatoriedade controlada, permitindo que o modelo explore mais opções sem comprometer a qualidade do texto gerado.

O top-k funciona restringindo o espaço de busca do modelo às k palavras mais prováveis, conforme calculadas pelas suas pontuações de probabilidade. Inicialmente, o modelo ordena todas as palavras possíveis com base nas suas probabilidades. Em seguida, em vez de escolher diretamente a palavra mais provável, ele limita a escolha apenas às k palavras mais prováveis e amostra aleatoriamente entre essas opções. Esse método permite que o modelo gere textos que são ao mesmo tempo coerentes e variados, evitando a repetição constante de escolhas muito prováveis e o uso de palavras muito improváveis.

Na Figura 6, podemos ver um exemplo de top-k com $k=2$, para escolher a próxima palavra da frase "*Ontem fui ao cinema ver um*". Primeiro, o modelo atribui probabilidades a várias palavras possíveis, como "filme", "documentário", "omelete", entre outras. Porém, ao aplicar o top-k, apenas as duas palavras com as maiores probabilidades ("filme" e "documentário") são mantidas, descartando as outras opções. Como resultado, as probabilidades das palavras restantes são normalizadas, aumentando as chances de escolher uma das palavras selecionadas. Nesse exemplo, a probabilidade de "filme" aumenta para 0,59 e a de "documentário" para 0,41 (somando 1), mostrando como o top-k influencia a geração de texto, restringindo a variedade de palavras e focando nas opções mais prováveis.

Na Figura 6, são apresentadas apenas as duas palavras com maiores probabilidades ("filme" e "documentário") são mantidas, enquanto as demais são descartadas, resultando em uma redistribuição das probabilidades.

Figura 6 - Exemplo de aplicação do parâmetro Top-k ($k=2$) na geração de texto



Fonte: imagem adaptada de [Viejo \(2023\)](#).

Uma das principais vantagens da amostragem Top-k é que ela pode ser ajustada para diferentes necessidades de geração de texto. Por exemplo, se o valor de k for baixo, o modelo restringe suas escolhas a um pequeno número de palavras, resultando em texto mais previsível e consistente, mas possivelmente menos criativo. Por outro lado, um valor maior de k permite que o modelo explore mais opções, aumentando a diversidade do texto gerado, mas com o risco de introduzir incoerências ou gerar sequências menos lógicas, aumentando a probabilidade de "alucinação".

Entretanto, a escolha do valor de k não deve ser feita de forma arbitrária. Ela deve considerar o contexto e os objetivos da geração de texto. Em aplicações onde a precisão e a consistência são importantes, como na tradução automática ou na geração de respostas em assistentes virtuais, valores menores de k podem ser preferíveis. Já em tarefas criativas, como a geração de histórias ou poesia, um k maior pode ser mais adequado para aumentar a originalidade e a criatividade.

1.2.2.2 Amostragem *Nucleus* ou Top-p

Um problema com a amostragem top-k é que k é fixo, mas a forma da distribuição de probabilidade sobre as palavras difere em diferentes contextos. Se definirmos $k = 10$, às vezes as 10 palavras principais serão muito prováveis e incluirão a maior parte da massa de probabilidade, mas, outras vezes, a distribuição de probabilidade será mais plana e as 10 palavras principais incluirão apenas uma pequena parte da massa de probabilidade.

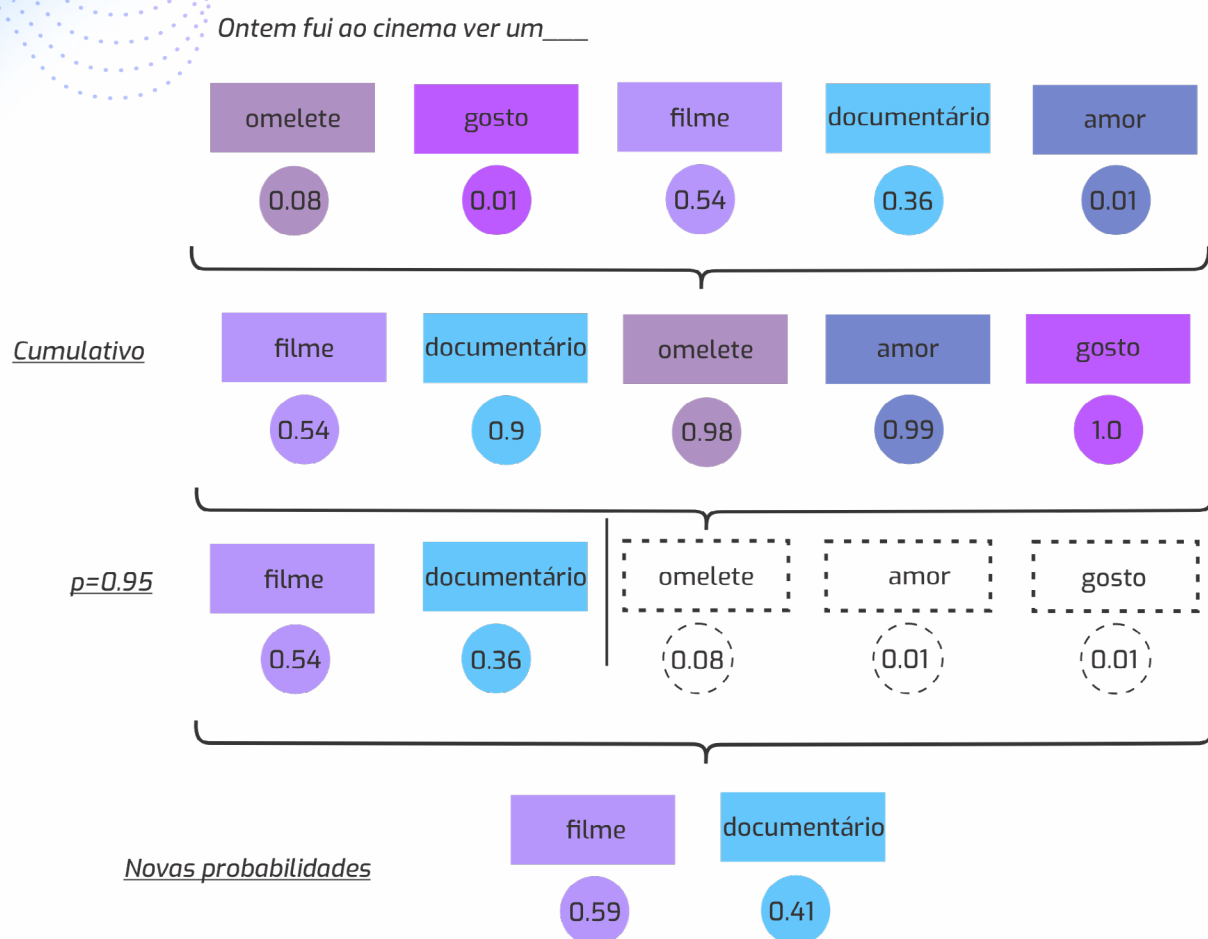
Além da técnica de amostragem top-k, outra técnica popular para controle da geração de texto em modelos com arquitetura *Transformer* é a amostragem top-p. Ela oferece uma abordagem dinâmica para selecionar palavras, ajustando automaticamente o tamanho do conjunto de palavras consideradas para cada passo da geração, com base na probabilidade cumulativa.

Na amostragem top-p, em vez de limitar a escolha a um número fixo de palavras, como na top-k, o modelo seleciona o menor subconjunto de palavras cujo somatório das probabilidades atinge ou excede um valor de limiar predefinido, chamado de p . Esse subconjunto é chamado de "núcleo". Uma vez identificado o núcleo, a próxima palavra é escolhida aleatoriamente dentro desse conjunto, com base nas probabilidades originais.

Na Figura 7, é apresentado um exemplo da amostragem top-p, tendo como entrada a frase "Ontem fui ao cinema ver um". O modelo inicialmente atribui probabilidades a várias palavras possíveis, como "filme", "documentário", "omelete", entre outras. À medida que as probabilidades são somadas, elas formam uma distribuição cumulativa. Com top-p configurado para 0,95, as palavras "filme" e "documentário" são selecionadas, pois suas probabilidades combinadas atingem o limite definido, enquanto as palavras com probabilidades mais baixas, como "omelete", "amor" e "gostar", são descartadas. Isso resulta em uma redistribuição das probabilidades das palavras selecionadas, com "filme" recebendo uma probabilidade de 0,59 e "documentário" de 0,41. O top-p é uma técnica que permite manter a diversidade de geração de texto, limitando a seleção apenas às palavras mais relevantes de acordo com o contexto.

Na Figura 7, a seguir, na aplicação do parâmetro top-p, as palavras mais prováveis são selecionadas até que a soma cumulativa das probabilidades atinja o valor de 0,95. As palavras com menores probabilidades são descartadas, resultando em uma redistribuição das probabilidades entre as palavras restantes.

Figura 7 - Exemplo de aplicação do parâmetro top-p ($p=0,95$) na geração de texto



Fonte: adaptada de [Viejo \(2023\)](#).

A principal vantagem da amostragem top-p em relação à top-k é a sua flexibilidade. Em situações onde uma única palavra é bem mais provável do que as outras, o núcleo pode ser pequeno, similar a um valor baixo de k , garantindo que o modelo não introduza opções incoerentes. Quando várias palavras têm probabilidades comparáveis, o núcleo pode se expandir para incluir mais opções, permitindo uma maior diversidade na geração de texto. Essa adaptabilidade pode resultar em saídas que combinam coerência com criatividade, dependendo do contexto.

A escolha do valor de p é importante para o desempenho do modelo. Valores baixos de p resultam em núcleos pequenos, o que pode fazer com que o texto gerado seja mais previsível, semelhante ao comportamento com um baixo valor de k . Já valores altos de p permitem a inclusão de mais palavras no núcleo, aumentando a diversidade do texto gerado, mas também o risco de se introduzir palavras menos coerentes com o contexto. Um valor típico de p utilizado é 0,9, o que permite que o modelo mantenha uma boa combinação de coerência e criatividade.

O top-p é útil em tarefas onde a flexibilidade e a adaptabilidade são necessárias, como na criação de diálogos ou textos criativos, onde o modelo deve manter uma resposta relevante ao contexto, mas também deve explorar diferentes possibilidades de forma dinâmica. Ao ajustar p conforme a necessidade da aplicação, é possível gerar saídas que são ao mesmo tempo variadas e coerentes, atendendo a vários contextos e propósitos na geração de texto.

1.2.2.3 Temperatura

A temperatura é um hiperparâmetro que controla o nível de aleatoriedade e criatividade do texto gerado. Ela funciona como um fator de ajuste que modifica a distribuição de probabilidades das palavras previstas pelo modelo, influenciando diretamente a seleção da próxima palavra na sequência.

A temperatura impacta nas probabilidades preditivas do modelo. Especificamente, as probabilidades são divididas pela temperatura antes de serem passadas por uma função de *softmax* para converter essas pontuações em uma distribuição final de probabilidades. Quando a temperatura é maior que 1, as probabilidades se tornam mais uniformes, o que aumenta a diversidade do texto gerado ao permitir que o modelo escolha palavras menos prováveis com maior frequência. Isso pode resultar em saídas mais criativas, mas também pode aumentar o risco de incoerências.

Por outro lado, quando a temperatura é menor que 1, as diferenças entre as probabilidades das palavras são amplificadas. Nesse caso, o modelo tende a escolher as palavras mais prováveis com mais frequência, resultando em um texto mais previsível e consistente, mas menos criativo. Uma temperatura muito baixa pode fazer com que o modelo gere saídas muito repetitivas ou até mesmo que ele trave em *loops* de repetição.

Um valor de temperatura baixo < 1 (ex: 0,2 ou 0,5) possui as seguintes características:

- » Ideal quando se espera uma resposta mais precisa.
- » Uma temperatura de 0 significa que o *token* com maior probabilidade geralmente será selecionado.
- » Concentra a maior parte da massa de probabilidade nas próximas palavras mais prováveis.
- » Isso resulta em gerações de texto mais coerentes e repetitivas, mas com menos criatividade ou exploração de opções menos prováveis.

Um valor de temperatura alto ≥ 1 (ex: 1,5 ou 2,0) possui as seguintes características:

- » Torna as previsões do modelo mais dispersas e "incertas".
- » Ele achata e distribui a distribuição de probabilidade mais uniformemente entre as palavras.
- » Isso permite que o modelo experimente com mais frequência escolhas de palavras menos prováveis, permitindo gerações de texto mais criativas, exploratórias e "surpreendentes".
- » Aumenta a probabilidade de escolhas menos coerentes e com alucinação.

O ajuste da temperatura permite que os desenvolvedores controlem o estilo e a natureza do texto gerado conforme a necessidade da aplicação. Em contextos onde a precisão e a coerência são necessários, como em traduções ou respostas automatizadas, uma temperatura baixa pode ser ideal. Em tarefas criativas, como a geração de histórias ou poesia, uma temperatura mais alta pode incentivar a exploração de ideias novas e inesperadas. Em resumo, a temperatura é como um botão que você pode ajustar para encontrar o equilíbrio perfeito entre precisão e criatividade.

- » **Temperatura baixa:** ideal para resumos, traduções e textos que exigem precisão e clareza.
- » **Temperatura alta:** perfeita para poemas, roteiros e textos que buscam a originalidade e a criatividade.

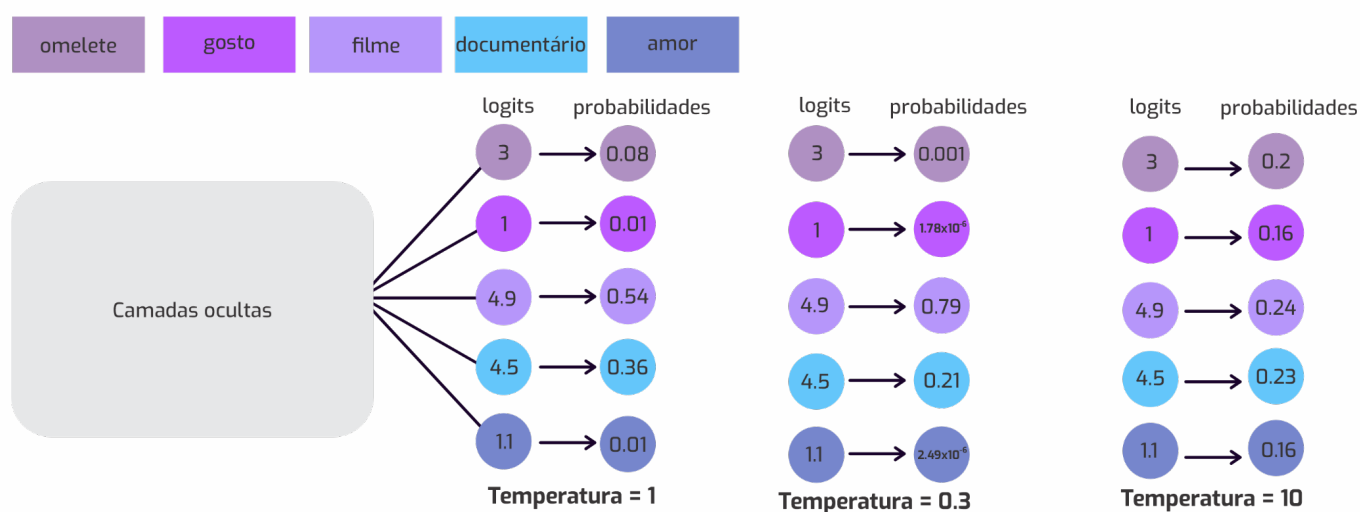
Na Figura 8, é ilustrado como a temperatura afeta a probabilidade de diferentes palavras em uma tarefa de geração de texto, onde o objetivo é completar a frase "On-tem fui ao cinema ver um ...". Cada círculo azul representa uma possível palavra, com os valores numéricos indicando os *logits* (à esquerda) e as probabilidades correspondentes (à direita) para cada temperatura. *Logits* são as saídas não normalizadas do modelo antes de escolher qual será a próxima palavra a ser gerada. A função *softmax* é aplicada para converter esses *logits* em probabilidades, e o modelo seleciona a próxima palavra com base nessas probabilidades.

Observe que com a temperatura em 1 (valor padrão), a palavra "filme" possui a maior probabilidade (0,54), seguida por "documentário" (0,36). Aumentar a temperatura para 10 torna as probabilidades mais uniformes, dando chances maiores para palavras como "omelete" e "amor", que antes eram menos prováveis. Ao contrário, diminuir a temperatura para 0,3 concentra a probabilidade em "filme" (0,79), tornando a escolha da palavra mais "previsível" e menos criativa.

Na Figura 8, a seguir, é ilustrado como a temperatura afeta a seleção da próxima palavra com base nos *logit scores* calculados pelo modelo. À esquerda, com temperatura igual a 1, as probabilidades refletem as diferenças nos *logit scores*. No centro, com uma temperatura de 0,3, as diferenças são ampliadas, resultando em uma maior concentração de probabilidade na palavra mais provável. À direita, com temperatura 10, as diferenças são suavizadas, distribuindo a probabilidade mais uniformemente entre as opções.

Figura 8 - Impacto da temperatura na distribuição de probabilidades em modelos de linguagem

Ontem fui ao cinema ver um____



Fonte: adaptada de [Viejo \(2023\)](#).

Compreender os efeitos da temperatura é importante para otimizar o desempenho do modelo em diferentes tarefas, desde a geração de respostas precisas até a produção de conteúdo criativo. A fórmula da **temperatura** em LMs é usada para controlar a "aleatoriedade" ou "exploração" na geração de texto ao mostrar a próxima palavra (*token*) com base nas probabilidades previstas pelo modelo:

$$P(w_i) = \frac{\exp(\frac{z_i}{T})}{\sum_j \exp(\frac{z_j}{T})}$$

onde:

- » $P(w_i)$ é a probabilidade ajustada da palavra w_i ,
- » z_i é o *logit* (valor bruto) para a palavra w_i ,
- » T é a **temperatura**, que controla a suavização da distribuição,
- » A soma no denominador é feita sobre todos os *logits* z_j .

A temperatura é uma maneira de regular o grau de exploração do modelo na geração de texto, afetando a previsibilidade e a variedade das respostas. A seguir, são apresentadas como a temperatura funciona internamente nos LLMs:

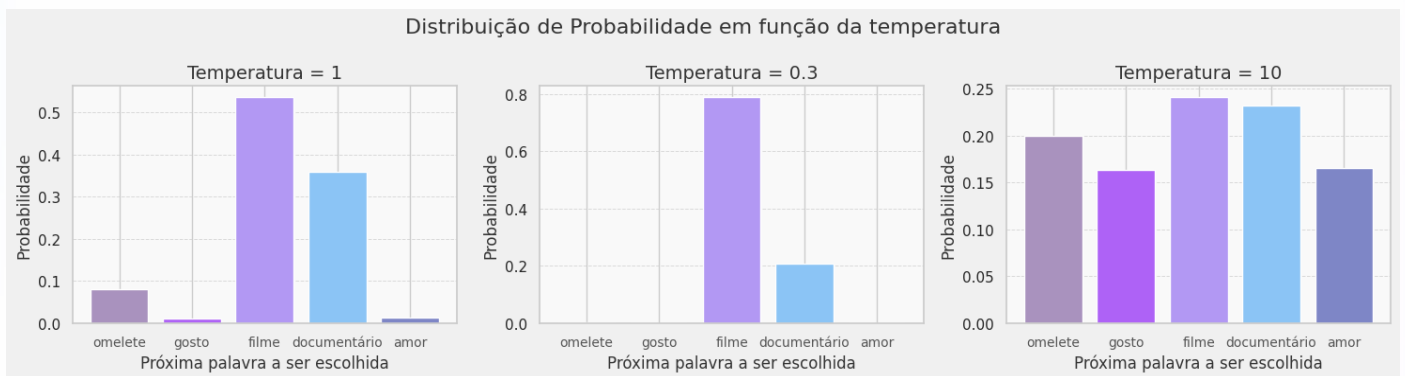
- » **Cálculo das probabilidades:** o LM calcula os *logits* z_i para cada palavra no vocabulário, baseado no contexto das palavras anteriores.
- » **Divisão pela temperatura:** os *logits* são divididos pelo valor da temperatura:
 - » Se **temperatura** < 1 : as probabilidades se tornam mais extremas, ou seja, probabilidades mais altas se tornam mais altas, e as mais baixas se tornam mais baixas.
 - » Se **temperatura** > 1 : as probabilidades se tornam menos extremas, aproximando-se de 0.
- » **Aplicação de softmax:** após o ajuste pela temperatura, aplica-se a função *softmax* para converter essas probabilidades em uma distribuição de probabilidade final, que soma 1 e determina a seleção das palavras durante a geração de texto.

Na Figura 9, podemos ver como a aplicação da temperatura influencia a distribuição de probabilidades. Quando o LM calcula as probabilidades iniciais para as palavras possíveis, ele na verdade trabalha com *logit scores*. É aqui que a temperatura aparece: ao dividir os *logit scores* pela temperatura, ajusta-se a "extremidade" dessas pontuações antes de passá-las pela função *softmax*, que transforma esses valores ajustados em probabilidades finais.

- » Temperatura = 1 (à esquerda na figura): nesse caso, os *logit scores* são divididos por 1, o que significa que não há modificação nas diferenças relativas entre eles. A função *softmax* então converte esses *logit scores* diretamente em probabilidades, refletindo as diferenças originais de confiança do modelo. Como resultado, a palavra com o *logit* mais alto ("filme", com *logit* 4,9) tem uma probabilidade considerável (0,54), mas outras palavras também têm grandes chances de serem escolhidas.

- » Temperatura < 1 (0,3) (no centro): os *logit scores* são divididos por um valor menor que 1, o que faz com que as diferenças entre eles sejam ampliadas. As probabilidades se tornam mais extremas: a palavra com o maior *logit* ("filme") tem agora uma probabilidade de 0,79, enquanto as outras praticamente desaparecem como opções (probabilidades muito próximas de zero). Isso ocorre porque a função *softmax* transforma as diferenças aumentadas nos *logit scores* em probabilidades ainda mais concentradas, privilegiando fortemente a palavra mais provável.
- » Temperatura > 1 (10) (à direita): quando a temperatura é maior que 1, os *logit scores* são divididos por um valor maior, o que achata as diferenças entre eles. Na função *softmax*, isso se traduz em uma distribuição de probabilidades mais uniforme, onde mesmo palavras com *logit scores* relativamente baixos recebem uma probabilidade maior do que normalmente teriam. Isso permite que o modelo explore uma mais opções de palavras, tornando a escolha final mais aleatória e menos previsível.

Figura 9 - Distribuição de probabilidade em função da temperatura



Fonte: autoria própria.

A temperatura ajusta o equilíbrio entre previsibilidade e criatividade na geração de texto por LLMs, controlando a distribuição de probabilidades das palavras. Temperaturas mais baixas tornam o modelo mais conservador, enquanto temperaturas altas aumentam a diversidade. Esse ajuste, embora útil na inferência, depende da qualidade do treinamento do modelo, que ensina o modelo a gerar distribuições precisas. Assim, treinamento e temperatura se complementam para criar um modelo eficaz e flexível.



1.3 Treinamento de Grandes Modelos de Linguagem

Para treinar uma arquitetura *Transformer* como um LM, temos como entrada um *corpus* de texto como material de treinamento e, a cada etapa de tempo t , pedimos ao modelo para prever o próximo *token*. Chamamos esse modelo de auto-supervisionado porque não precisamos adicionar nenhum rótulo com a resposta; a sequência natural de *tokens* é sua própria supervisão. Nós treinamos o modelo para minimizar o erro na previsão da próximo *token* na sequência de treinamento, usando a entropia cruzada como a função de perda.

Nos exemplos desta seção, iremos utilizar exemplos de predição da próxima palavra para facilitar o entendimento. Podemos utilizar como exemplo a frase "Os gatos brincam felizes com a". O modelo precisa prever qual palavra vem a seguir. Como ele já viu muitas frases similares durante o treinamento, ele pode prever que a próxima palavra será "bola", "lã", ou "caixa". Seu trabalho é adivinhar a palavra que melhor se encaixa no contexto.

Treinar LLMs é como ensinar um computador a prever e gerar textos. O treinamento de um LLM envolve alimentar o modelo com grandes quantidades de texto. Esse texto pode ser retirado da internet, livros, artigos, entre outros. A ideia é que, com essa quantidade enorme de dados, o modelo consiga aprender padrões na linguagem, como quais palavras costumam aparecer juntas e em quais contextos.

Esse treinamento é chamado de auto-supervisionado. Isso significa que o modelo aprende sozinho, sem precisar que um humano diga a resposta certa para cada frase. Em vez disso, ele usa o próprio texto como orientação. Se a frase for "Os gatos brincam felizes com a", o modelo sabe que "bola" é a resposta correta e ajusta suas "configurações internas" para melhorar suas previsões futuras.

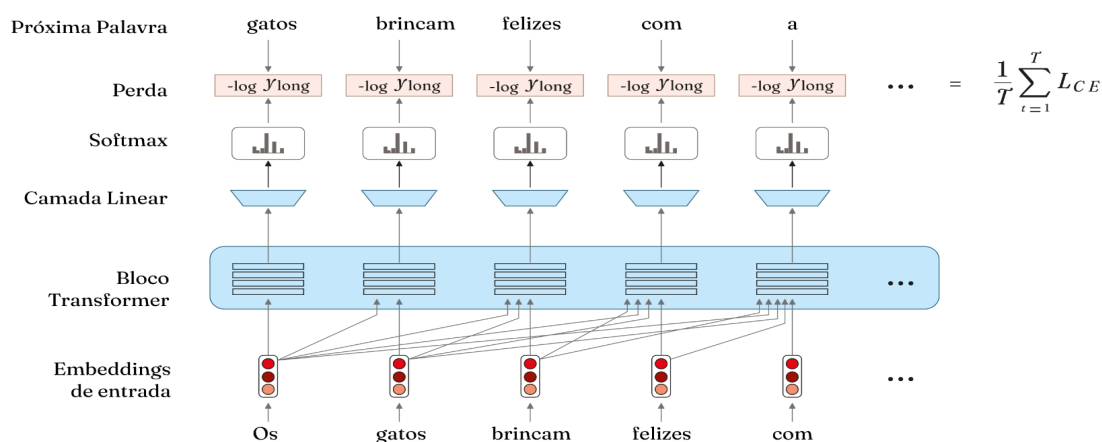
Na Figura 10, é ilustrada a abordagem geral de treinamento. Em cada etapa, dadas todas as palavras anteriores, a camada final do *transformer* produz uma distribuição de saída sobre todo o vocabulário. Cada palavra é convertida em um *embedding*, que é então processado pelos blocos do *Transformer*. A cada etapa, o modelo tenta prever a próxima palavra, recebendo "retorno" do quanto a resposta está correta e ajustando seus parâmetros. No final, o modelo gera uma distribuição de probabilidade sobre todas as palavras possíveis, escolhendo a palavra com a maior probabilidade como sua previsão.

Na imagem, vemos um modelo que, a cada nova palavra recebida, tenta prever a próxima. Por exemplo, se o modelo já recebeu as palavras "Os gatos brincam felizes com a", ele agora tenta prever "bola". Se a previsão estiver correta, o modelo ganha um retorno positivo; se não, ele ajusta suas configurações para melhorar na próxima vez.

Durante o treinamento, a probabilidade atribuída à palavra correta é usada para calcular a perda de entropia cruzada para cada item na sequência. Como acontece com as RNNs, a perda para uma sequência de treinamento é a perda média de entropia cruzada em toda a sequência. Os pesos na rede são ajustados para minimizar a perda média da entropia cruzada sobre a sequência de treinamento por meio de descida de gradiente. Ou seja, o modelo ajusta seus pesos para prever melhor as palavras corretas em uma frase, usando um cálculo para medir o quão longe suas previsões estão da resposta certa, e melhora com base nesses erros.

Na Figura 10, é ilustrado como funciona o treinamento de um modelo como o *Transformer*, que é uma das arquiteturas mais usadas para criar LLMs.

Figura 10 - Treinamento de um grande modelo de linguagem como *Transformer*



Fonte: adaptada de Jurafsky; Martin (2024).

Esse ciclo de previsão e correção se repete milhões de vezes com diferentes frases e contextos até que o modelo fique muito bom em prever palavras e gerar textos que façam sentido. No final do processo, o modelo se torna capaz de realizar tarefas complexas, como responder perguntas, resumir textos e até mesmo escrever histórias.

No caso da modelagem de linguagem, a distribuição correta vem do conhecimento da próxima palavra. Isso é representado como um vetor *one-hot* correspondente ao vocabulário, onde a entrada para a próxima palavra real é 1 e todas as outras entradas são 0. Assim, a perda de entropia cruzada para modelagem de linguagem é determinada pela probabilidade que o modelo atribui à próxima palavra correta.

Assim, em cada posição de palavra t da entrada, o modelo recebe como entrada a sequência correta de *tokens* e os usa para calcular uma distribuição de probabilidade sobre as próximas palavras possíveis, a fim de calcular a perda do modelo para o próximo *token*. A ideia de que sempre damos ao modelo a sequência de histórico correta para prever a próxima palavra (em vez de alimentar o modelo com seu melhor caso da etapa de tempo anterior) é denominada *teacher forcing*. Isso permite que todos os *tokens* de *output* sejam corrigidos, ajustando a distribuição de probabilidade dos *tokens* preditos para cada um dos *tokens* propagados no modelo, além de economizar processamento ao garantir um treinamento mais eficiente.

Modelos grandes são geralmente treinados preenchendo a janela de contexto completa com texto. Se os documentos forem mais curtos que isso, vários documentos são agrupados na janela com um *token* especial de fim de texto entre eles. O tamanho do lote para a descida de gradiente é geralmente muito grande.

O treinamento de LLMs é um processo contínuo e iterativo. Os pesquisadores estão constantemente buscando novas técnicas e arquiteturas para aprimorar a capacidade dos modelos. Novas formas de representar as palavras, novos métodos de treinamento e novas estruturas de redes neurais são exploradas para tornar os LLMs cada vez mais robustos e versáteis.

O objetivo final é criar modelos que não apenas imitem a linguagem humana, mas que também sejam capazes de compreendê-la profundamente, gerando textos criativos, informativos e úteis para uma variedade de tarefas. O treinamento de LLMs é um processo que envolve uma combinação de dados massivos, algoritmos avançados e um ciclo constante de aprendizado. É por meio desse processo que os LLMs se tornam capazes de gerar textos que se assemelham àqueles escritos por humanos, abrindo possibilidades para a comunicação, a criatividade e a resolução de problemas.



1.3.1 Relação entre Número de *Tokens*, Tamanho do Modelo e Custo de Treinamento

O treinamento de LLMs envolve uma complexa interação entre o número de *tokens*, o tamanho do modelo e o custo de treinamento. Esses três fatores estão relacionados e impactam diretamente tanto no desempenho computacional e na acurácia do modelo final. Existe um equilíbrio entre o número de *tokens*, o tamanho do modelo e o custo de treinamento. Aumentar qualquer um desses fatores pode melhorar o desempenho do modelo, mas ao custo de maior complexidade e recursos financeiros.

O número de *tokens* refere-se à quantidade de pedaços de texto em que os dados de treinamento são divididos. Um *token* pode ser uma palavra, uma parte de uma palavra ou até mesmo um símbolo de pontuação. Quanto mais *tokens* um modelo é exposto durante o treinamento, mais informações ele pode absorver. Isso permite que o modelo compreenda nuances e padrões da linguagem de maneira mais precisa. Porém, utilizar mais *tokens* também significa mais ciclos de processamento, aumentando o tempo e os recursos necessários para o treinamento.

O tamanho do modelo é geralmente medido pelo número de parâmetros, que são os componentes ajustáveis que o modelo usa para fazer previsões. Modelos maiores têm mais parâmetros, o que lhes permite capturar e modelar mais detalhes sobre a linguagem. Quanto maior o número de parâmetros, maior é a capacidade de processamento necessária para treinar o modelo. Isso significa que modelos maiores demandam mais memória e poder computacional, o que aumenta o custo de treinamento.

O custo de treinamento de um LM grande é uma função direta do número de *tokens* e do tamanho do modelo. Modelos com mais parâmetros treinados em *corpora* de grande escala (com bilhões ou até trilhões de *tokens*) requerem um volume grande de recursos computacionais. Isso inclui o uso de processadores especializados, como GPUs e TPUs, que são caros tanto em termos de hardware quanto de energia. Além disso, o tempo necessário para treinar esses modelos aumenta exponencialmente com o tamanho, o que pode resultar em custos que chegam a milhões de dólares.

O GPT-3, com seus 175 bilhões de parâmetros e treinado em 300 bilhões de *tokens*, foi um marco na área de IA, permitindo que o modelo aprenda novas tarefas com apenas alguns exemplos no *prompt*, sem a necessidade de ajustes em seus pesos. A configuração específica de 175 bilhões de parâmetros e 300 bilhões de *tokens* não foi arbitrária. Ela foi baseada em um estudo cuidadoso das leis de escalabilidade para redes neurais, como descrito no artigo "*Scaling Laws for Neural Language Models*" (Kaplan *et al.*, 2020), publicado pela OpenAI, em 2020.

Esse artigo introduziu, pela primeira vez, equações que permitiam estimar a qualidade de um modelo neural antes mesmo de seu treinamento, considerando três variáveis principais: o tamanho do modelo (número de parâmetros), a quantidade de dados de treinamento (número de *tokens*) e a quantidade de computação necessária (medida em *Floating Point Operations per Second* [FLOPs]). A OpenAI, ao extrapolar essas equações, determinou que, dado o orçamento computacional disponível na época, a melhor configuração seria um modelo com 175 bilhões de parâmetros, treinado com 300 bilhões de *tokens*. Esta proporção de aproximadamente dois *tokens* por parâmetro foi seguida por outros projetos, como o OPT-175B da Meta, treinado com a mesma proporção, e o PaLM do Google, que possuía 540 bilhões de parâmetros e foi treinado com 780 bilhões de *tokens*.

Posteriormente, pesquisadores da DeepMind identificaram que havia um erro na metodologia usada no artigo da OpenAI, que superestimava o número de parâmetros e subestimava a quantidade de *tokens* necessária para otimizar a qualidade do modelo. Com a correção dessas equações, a equipe da DeepMind desenvolveu o modelo Chinchilla (Hoffmann *et al.*, 2022). Eles descobriram que, para um treinamento com computação otimizada, o tamanho do modelo e o número de *tokens* de treinamento devem ser escalados igualmente: quando o tamanho do modelo dobra, o número de *tokens* de treinamento também deve dobrar. Esse ajuste mostrou que treinar com muito mais *tokens* do que anteriormente considerado levava a uma qualidade superior, mesmo com um custo computacional semelhante.

A partir daí, iniciou-se uma corrida por mais dados de treinamento, com modelos como o GPT-4 (embora o número exato de *tokens* não seja público), PaLM 2 (Google) e Megatron-Turing NLG (NVIDIA e Microsoft) adotando proporções de *tokens* por parâmetro cada vez maiores. Essa história mostra que detalhes aparentemente pequenos, como a escolha da proporção ideal entre *tokens* e parâmetros, podem ter um impacto profundo no sucesso dos LMs. A busca por esse equilíbrio continua a evoluir, com implicações diretas não apenas na qualidade dos modelos, mas também no custo e na viabilidade de seu treinamento em larga escala.



1.3.2 Corpora de Treinamento para Grandes Modelos de Linguagem

Corpora de treinamento, conjuntos de dados usados para treinar modelos de aprendizado de máquina, são obtidos por meio de uma variedade de métodos, dependendo da tarefa e dos recursos disponíveis. Um método comum é a coleta de dados da *web*, onde se utilizam técnicas como *web scraping* para extrair texto, imagens, áudio e vídeo de *sites*, plataformas de mídia social e outras fontes online.

O texto da *web* é geralmente retirado de *corpora* de páginas da *web* coletadas automaticamente, como o *common crawl*, uma série de *snapshots* de toda a *web* produzidos pela organização sem fins lucrativos *Common Crawl* (Common Crawl, 2024), que possuem bilhões de páginas da *web*. Existem várias limpezas de dados de rastreamento comum, como o *Colossal Clean Crawled Corpus* (Raffel et al., 2020), um *corpus* de 156 bilhões de *tokens* de inglês que é filtrado de várias maneiras (removendo linguagem não natural como código, frases com palavras ofensivas de uma lista de bloqueio, etc.). O que há nesses dados? Uma análise sugere que, em grande parte, são documentos de texto de patentes, Wikipédia e *sites* de notícias (Dodge et al., 2021). A Wikipédia desempenha um papel importante em muitos treinamentos de LMs, assim como vários *corpora* de livros. Os modelos GPT3, por exemplo, são treinados principalmente na *web* (429 bilhões de *tokens*), conjuntos de textos de livros (67 bilhões de *tokens*) e Wikipédia (3 bilhões de *tokens*).

Dados existentes, como livros, artigos científicos, notícias e transcrições de conversas, podem ser reutilizados para formar um *corpus*. Bibliotecas digitais, repositórios de dados abertos e coleções privadas podem ser fontes de treinamento. A qualidade e a representatividade desses dados devem ser avaliadas antes do uso.

A colaboração em massa (*crowdsourcing*) envolve a contribuição de um grande número de pessoas na criação e anotação de dados, frequentemente por meio de plataformas online. Essa abordagem permite a coleta rápida de grandes quantidades de dados a um custo relativamente baixo. Esse tipo de dado pode ser utilizado, mas é importante garantir mecanismos de controle de qualidade devido à confiança da informação.

Em alguns casos, dados gerados automaticamente são utilizados. Isso pode incluir texto sintético gerado por LMs, imagens criadas por algoritmos de geração de imagens ou dados simulados de ambientes virtuais. Dados gerados automaticamente podem ser úteis para aumentar *corpora* existentes ou para criar dados para tarefas específicas.

A criação manual de dados é outra abordagem, onde especialistas humanos criam e anotam os dados, garantindo alta qualidade e precisão. Esse método é mais trabalhoso e caro, mas pode melhorar o resultado de tarefas que exigem conhecimento especializado ou onde a disponibilidade de dados é limitada. Frequentemente, a criação manual é combinada com outras técnicas para aumentar o tamanho do *corpus*.

A combinação de diferentes fontes de dados também é uma prática comum para criar *corpora* mais abrangentes e representativas. A combinação pode envolver a fusão de *corpora* existentes, a adição de dados coletados da web ou a integração de dados criados manualmente. Uma tarefa a ser realizada neste tipo de dado é garantir a consistência e a compatibilidade dos diferentes conjuntos de dados durante o processo de combinação.

Como esses *corpora* de treinamento são muito grandes, é provável que contêm muitos exemplos naturais que podem ser úteis para tarefas de NLP, como pares de perguntas e respostas (por exemplo, de listas de perguntas frequentes), traduções de frases entre vários idiomas, documentos junto com seus resumos e assim por diante.

Em todos os casos, a consideração de questões éticas, como privacidade, direitos autorais e vieses presentes nos dados, devem ser levadas em conta durante todo o processo de obtenção de *corpora* de treinamento. A qualidade, a representatividade e o tamanho do *corpus* afetam diretamente o desempenho do modelo de aprendizado de máquina treinado nele.

1.4 Desafios dos Grandes Modelos de Linguagem

Os LLMs são ferramentas com grande potencial, mas é importante usá-los com responsabilidade e consciência. Por exemplo, LMs são propensos a dizer coisas falsas, um problema chamado "alucinação". Os LMs são treinados para gerar texto previsível e coerente, mas os algoritmos probabilísticos que vimos até agora não têm como garantir que o texto gerado seja correto ou verdadeiro. Isso pode ocasionar diversos problemas para qualquer aplicativo onde os fatos são importantes!

Outro desafio é que os LMs podem gerar linguagem tóxica (Gehman *et al.*, 2020), onde mesmo *prompts* atóxicos podem levar LLMs a produzirem discurso de ódio e abusar de seus usuários. LMs também geram estereótipos (Cheng, Durmus, Jurafsky, 2023) e atitudes negativas (Sheng, *et al.*, 2019; Brown, *et al.*, 2020) sobre muitos grupos demográficos.

Uma fonte de vieses são os dados de treinamento (Gehman *et al.*, 2020), onde os conjuntos de dados de treinamento de LM extensa incluem texto tóxico extraído de *sites* proibidos. Existem outros vieses além da toxicidade: os dados de treinamento são desproporcionalmente gerados por autores dos Estados Unidos da América e de países desenvolvidos. Essas amostras tendenciosas provavelmente distorcem a geração resultante em direção às perspectivas ou tópicos deste grupo de países. Além disso, os LMs podem amplificar os vieses demográficos e preconceituosos.

LMs também podem ser usados por agentes maliciosos para gerar texto para desinformação, *phishing* ou outras atividades socialmente prejudiciais (Brown, *et al.*, 2020). Pesquisas mostram que LLMs podem gerar texto que emula extremistas online (McGuffie; Newhouse, 2020), com o risco de amplificar movimentos extremistas e sua tentativa de radicalizar e recrutar.

LMs também apresentam problemas de privacidade, pois podem vaziar informações sobre seus dados de treinamento. Assim, é possível que um adversário extraia texto de dados de treinamento de um LM, como o nome, número de telefone e endereço de uma pessoa (McGuffie; Newhouse, 2020; Giray, 2023). Isso é um problema se LLMs são treinados em conjuntos de dados privados, como registros eletrônicos de saúde.

Relacionada à privacidade está a questão dos direitos autorais. LLMs podem ser treinados em documentos protegidos por direitos autorais. Em alguns países, como os Estados Unidos da América, a doutrina do uso justo permite que conteúdo protegido por direitos autorais seja usado para construir LMs, mas não se eles forem usados para gerar texto que concorra com o mercado para o texto no qual são treinados.

Encontrar maneiras de mitigar todos esses danos é uma importante área de pesquisa atual em PLN. No mínimo, analisar os dados usados para pré-treinar LLMs é importante como uma forma de entender as questões de toxicidade, viés, privacidade e uso justo, tornando importante que os LMs forneçam informações completas e replicáveis sobre os *corpora* usados para treiná-los. Modelos de código aberto podem especificar seus dados de treinamento exatos. Os requisitos para que os modelos sejam transparentes de tais maneiras também estão em processo de incorporação aos regulamentos de vários governos nacionais.



SAIBA MAIS...

✿ Artigos acadêmicos e publicações:

✿ *Attention is All You Need* (Vaswani et al., 2017): este artigo apresenta a arquitetura *Transformer*, que é a base dos LLMs.

✿ *Scaling Laws for Neural Language Models* (Kaplan et al., 2020): discute como o aumento do tamanho dos LMs impacta diretamente a performance e as capacidades emergentes.

✿ Ferramentas e *notebooks* interativos:

✿ Utilize o *Hugging Face* para testar diversos LMs e ajustar parâmetros como temperatura e top-p *sampling*. Disponível em: <https://huggingface.co/>.

✿ Vídeos e cursos online:

✿ [Curso de PLN da Stanford University](https://web.stanford.edu/class/cs224n/): esse curso aborda em detalhes as técnicas utilizadas para construir e ajustar LLMs. Disponível em: <https://web.stanford.edu/class/cs224n/>.

✿ *Deep Learning Specialization* no Coursera, oferecido por Andrew Ng: Um ótimo recurso para aprender sobre redes neurais e transformers. Disponível em: <https://www.coursera.org/specializations/deep-learning>.

✿ *Speech And Language Processing*, 3ª edição (Jurafsky, 2024).

PARA RELEMBRAR...

Nessa Unidade, exploramos os fundamentos dos LLMs, desde os tradicionais modelos probabilísticos, como n-gramas, até os avançados LLMs baseados em arquiteturas Transformer. Também discutimos a evolução dos LMs, destacando a transição de modelos neurais para os grandes modelos pré-treinados e suas aplicações, incluindo assistentes virtuais, tradutores automáticos e corretores de texto.

Além disso, explicamos como os LLMs são treinados em grandes quantidades de dados, o que lhes permite gerar texto similar ao humano e realizar diversas tarefas, como tradução, geração de texto e análise de sentimentos. Aqui estão os pontos-chave abordados nessa Unidade:

- ✿ LMs: modelos que prevêm a probabilidade da próxima palavra em uma sequência, atribuindo probabilidades a palavras ou sequências de palavras.

- ✿ LLMs: versões maiores e mais poderosas dos LMs, treinadas em conjuntos de dados massivos, com bilhões ou até trilhões de parâmetros. Eles exibem uma compreensão profunda da linguagem e do contexto.

- ✿ Modelos probabilísticos: usam regras estatísticas e probabilidades para prever palavras, como os modelos N-gramas. São mais simples, mas menos eficazes em contextos longos.

- ✿ Modelos neurais: utilizam redes neurais (RNNs, LSTMs) para processar a linguagem. Capturam melhor o contexto, mas são computacionalmente mais caros.

- ✿ Modelos pré-treinados (ex: BERT, ELMo): treinados em grandes conjuntos de dados e depois ajustados para tarefas específicas. Oferecem bom desempenho com poucos dados adicionais.

- ✿ *Transfer learning*: técnica que permite transferir o conhecimento adquirido em uma tarefa para outra tarefa relacionada, acelerando o treinamento e melhorando o desempenho em tarefas específicas.

- ✿ *Fine-tuning* (ajuste fino): adaptação de um modelo pré-treinado para uma tarefa específica, ajustando seus parâmetros com um conjunto de dados menor e especialista em tarefas específicas.

- ✿ Escalabilidade: capacidade de um sistema (incluindo LLMs) de lidar com o aumento da demanda (dados, usuários, complexidade) sem perder desempenho computacional ou acurácia.

Unidade III

In-context learning *e fine-tuning* em LLMs

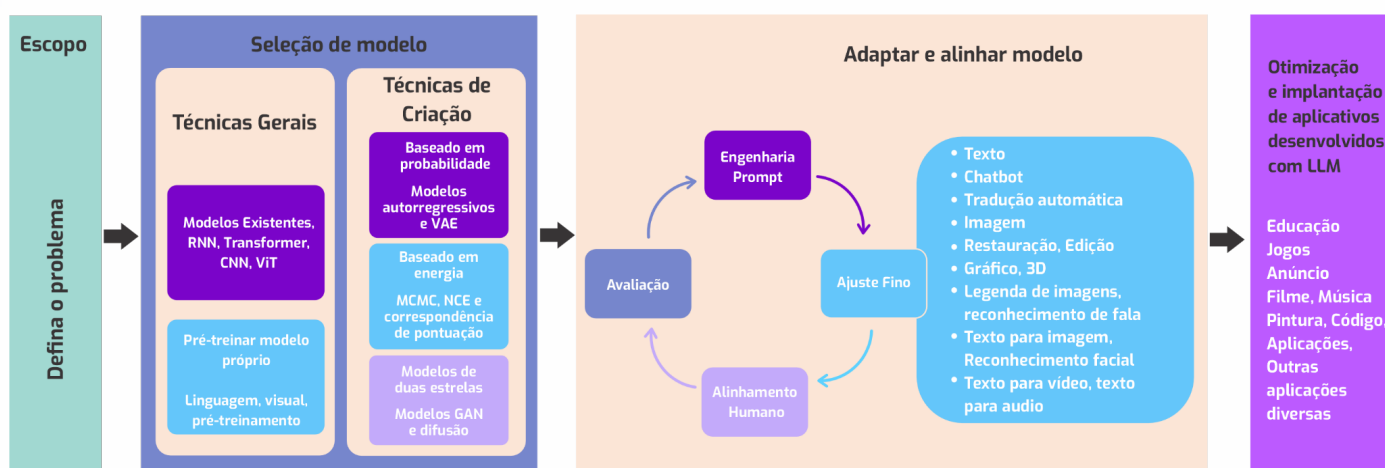
Unidade II - Engenharia de *Prompts*

2.1 Introdução à Engenharia de *Prompts*

A IA está alterando a maneira como interagimos com a tecnologia. De textos e imagens a código e música, a IA generativa nos permite criar conteúdo de forma rápida. Mas para aproveitar todo o potencial dessas ferramentas, precisamos aprender a nos comunicar com elas de forma eficaz.

Este conceito se manifesta no processo de desenvolvimento de aplicações baseadas em Inteligência Artificial Generativa (IAG), como ilustrado na Figura 11. Esta *pipeline* começa com a definição do escopo do projeto e se estende até a otimização e implantação final das aplicações alimentadas por *Large Language Models*. Na Figura 11, é apresentado um fluxograma detalhado que pode ser dividido em três fases principais: seleção do modelo, adaptação e alinhamento do modelo, além da otimização e implantação do modelo.

Figura 11 - Ciclo de *design* e aplicação de IA Generativa



Fonte: adaptada de Hadi et al. (2024).

Dessa forma, o ciclo segue as seguintes etapas:

1. Escopo: definir o problema e o escopo;
2. Seleção de modelo: seleção de modelo com a escolha das técnicas de modelagem adequadas, como modelos autorregressivos, modelos baseados em energia ou modelos de difusão;

3. Adaptação e alinhamento de modelo: otimização para tarefas específicas por meio da engenharia de *prompts*, ajuste fino do modelo ou alinhamento humano;
4. Otimização e implantação do modelo ajustado para várias aplicações nos domínios de educação, jogos, mídia e muito mais.

Na fase de **Seleção do Modelo**, os desenvolvedores podem optar por técnicas gerais, que incluem o uso de modelos existentes (como RNN, *Transformer* e *Convolutional Neural Networks* [CNN]) ou a criação de modelos próprios pré-treinados. Alternativamente, podem explorar técnicas de criação mais específicas, como modelos autoregressivos baseados em verossimilhança ou modelos baseados em energia.

A fase de **Adaptação e Alinhamento do Modelo** se baseia em *prompts* bem elaborados para a tarefa específica. Esta etapa envolve três processos principais: engenharia de *prompts*, avaliação e *fine-tuning*. A engenharia de *prompts* é particularmente importante pois permite ajustar o comportamento do modelo para tarefas específicas sem necessidade de retreinamento completo.

A fase final de **Otimização e Implantação** foca na entrega de aplicações alimentadas por LLMs em diversos domínios, como educação, jogos, saúde, entretenimento e desenvolvimento de *software*.

Os *prompts* gerados na fase de Adaptação e Alinhamento do Modelo podem ser usados para ajustar LMs para tarefas específicas, como gerar texto, traduzir idiomas, resumir texto ou responder a perguntas. Um *prompt* é o texto de entrada fornecido ao modelo para gerar uma resposta. Ele serve como instrução, contexto e direcionamento para o LLM, influenciando o conteúdo, o estilo e a estrutura da saída gerada. A Engenharia de *Prompt* permite que os usuários otimizem o desempenho do modelo para uma variedade de tarefas. Por meio de *prompts* bem formulados, os usuários podem orientar o modelo para gerar saídas desejáveis.

Mas, afinal o que é engenharia de *prompt*? Engenharia de *prompt* é o processo de descobrir *prompts* que geram de forma confiável resultados úteis ou desejados, ou seja, a habilidade de elaborar entradas para modelos de IA generativa, como o ChatGPT, a fim de obter os resultados desejados. **Não se trata apenas de fazer perguntas; trata-se de comunicar seus objetivos com clareza e precisão, guiando a IA para gerar respostas relevantes, criativas e de alta qualidade.**

Essa abordagem envolve converter uma ou mais tarefas em um conjunto de dados baseado em *prompts* e treinar um LM usando uma técnica conhecida como "aprendizado baseado em *prompts*" ou "aprendizado de *prompts*" (Giray, 2023; Gero, Liu, Chilton, 2022). A engenharia de *prompts* é uma disciplina recente que se concentra em desenvolver e otimizar *prompts* para utilizar efetivamente LMs de grande escala em diversas aplicações e áreas de pesquisa (White *et al.*, 2023).

Em termos simples, a engenharia de *prompt* é a arte de elaborar a instrução - o *prompt* - para um LM, tal como o ChatGPT, a fim de obter o resultado desejado. É como dar ao modelo um mapa detalhado, com a rota traçada e o destino destacado, para que ele possa navegar pelo universo da linguagem e trazer as informações mais relevantes e precisas para você.

Aqui está um exemplo simples de uma entrada de *prompt* para um gerador de nomes de produtos (inspirado em um dos exemplos do OpenAI), e a saída resultante do ChatGPT.

Entrada:

Liste os 5 principais livros na área de processamento de linguagem natural

Saída:

Claro, aqui estão os cinco principais livros na área de Processamento de Linguagem Natural (PLN):

- » *"Speech and Language Processing" de Daniel Jurafsky e James H. Martin*
- » *"Natural Language Processing with Python" de Steven Bird, Ewan Klein e Edward Loper*
- » *"Foundations of Statistical Natural Language Processing" de Christopher D. Manning e Hinrich Schütze*
- » *"Neural Network Methods in Natural Language Processing" de Yoav Goldberg*
- » *"Deep Learning for Natural Language Processing" de Palash Goyal, Sumit Pandey e Karan Jain*

Ao empregar técnicas de engenharia de *prompts*, conseguimos aproveitar o potencial dos LMs, destacando suas capacidades em vários domínios. Esse cenário abre novas opções para melhorar sistemas de IA e aumentar seu desempenho em diversas aplicações, desde geração de texto até síntese de imagens e além.

Ao desenvolver um *prompt*, o custo de *prompts* ineficientes pode ser alto, tanto em termos financeiros (taxas do OpenAI) quanto em tempo dedicado à correção de erros. Por isso, o conteúdo do *prompt* é importante e alguns pontos devem ser considerados:

» **Clareza e Especificidade**

- » Seja preciso na descrição do que você busca. Por exemplo, em vez de "pedir os principais livros", especifique "os cinco principais livros".
- » Forneça contexto incluindo informações relevantes sobre os livros, como público-alvo, funcionalidades e valores da marca.
- » Forneça exemplos de livros que você considera adequados para contextualizar o modelo.

» **Controle da Saída**

- » Limite a quantidade de caracteres ou palavras para evitar saídas muito longas ou curtas.
- » Use técnicas como "*few-shot learning*" (orientação do modelo com poucos exemplos) ou "*chain-of-thought prompting*" (orienta o modelo a explicar seu raciocínio passo a passo) para guiar o modelo em direção à saída desejada.
- » Implemente mecanismos para filtrar saídas inadequadas, como nomes de livros que não são da área de PLN.

» **Otimização de Custo**

- » Utilize frases concisas e elimine informações irrelevantes para reduzir o custo das taxas do OpenAI.
- » Experimente diferentes estruturas de *prompt* e compare os resultados para encontrar a melhor relação custo-benefício.

» **Qualidade do prompt e da saída**

- » Permita que os usuários personalizem o *prompt* com informações específicas de livros, artigos ou livros.
- » Implemente mecanismos de avaliação da qualidade das saídas geradas para manter a consistência e a relevância.

Ao criar em um *prompt* otimizado, você garante um sistema que irá gerar uma saída para mais eficiente, confiável e escalável, que atenderá às necessidades de milhares de usuários.

2.2 Princípios de Engenharia de *Prompts*

O livro "*Prompt Engineering for Generative AI*" (Phoenix, Taylor, 2024) apresenta uma lista com cinco princípios a serem seguidos na criação de *prompts*. Esses princípios não são dicas ou truques efêmeros, mas sim convenções geralmente aceitas que são úteis para trabalhar com LMs. Esses princípios são independentes do modelo e devem funcionar para melhorar seu *prompt*, independentemente do modelo que você está usando.

- » **Forneça a direção:** descreva o estilo desejado em detalhes.
- » **Especifique o formato:** defina quais regras seguir e a estrutura necessária da resposta.
- » **Forneça exemplos:** insira um conjunto diversificado de casos de teste onde a tarefa foi realizada corretamente.
- » **Avalie a qualidade:** identifique erros e classifique as respostas, testando o que impulsiona o desempenho.
- » **Faça a divisão do trabalho:** divida as tarefas em várias etapas, interligadas para objetivos complexos.

Cada um desses itens será comentado nas subseções seguintes.

2.2.1 Forneça a Direção

A clareza das instruções permite obter resultados mais relevantes ao utilizar MLN. Fornecer uma direção clara ajuda a guiar o modelo para compreender exatamente o que é esperado na resposta. A direção deve ser explícita e evitar ambiguidades, detalhando o objetivo da tarefa de maneira específica. Fornecer a direção significa orientar o modelo de forma precisa e detalhada sobre o que você espera como resultado. Isso pode ser feito por meio de:

- » **Descrever o estilo desejado em detalhes:** se você deseja um texto formal ou informal, técnico ou criativo, é preciso especificar.
- » **Dar exemplos concretos:** mostre ao modelo exemplos de resultados desejáveis ajuda a ilustrar suas expectativas.

- » **Definir o contexto:** forneça informações relevantes sobre o público-alvo, a finalidade do texto ou o público alvo do resultado do *prompt*.
- » **Utilizar analogias:** compare a tarefa com algo familiar ao modelo para facilitar a compreensão.

O exemplo abaixo "Escreva um artigo breve sobre os benefícios da IA na medicina, com um tom informativo e acessível" ilustra como fornecer a direção correta (Figura 12).

Figura 12 - Exemplo de *Prompt*

Entrada:

"Escreva um artigo breve sobre os benefícios da inteligência artificial na medicina, com um tom informativo e acessível."

Saída do ChatGPT:

"A inteligência artificial (IA) está revolucionando a medicina ao melhorar diagnósticos, personalizar tratamentos e otimizar a gestão hospitalar. Com algoritmos avançados, a IA pode analisar grandes volumes de dados clínicos, identificando padrões que podem escapar ao olho humano..."

Fonte: autoria própria.

O exemplo acima define o **objetivo claro** de produzir um artigo sobre os benefícios da inteligência artificial na medicina, detalha o **tipo de conteúdo desejado** ao especificar um "artigo breve", e **estabelece o tom e estilo** como "informativo e acessível". Essas informações direcionam o ChatGPT a gerar um texto conciso, objetivo e de fácil compreensão sobre o tema, evitando respostas excessivamente técnicas ou com tom inadequado.

2.2.2 Especifique o Formato

Especificar o formato garante que a IA gere uma saída no formato que você deseja. Se você não especificar o formato, a IA pode gerar uma saída em um formato que é difícil de usar. Você pode especificar o formato fornecendo à IA um exemplo do formato que você deseja ou especificando as regras que a saída deve seguir. Você deve definir as regras e a estrutura da resposta esperada, incluindo detalhes como o tamanho, a formatação e o estilo da saída do modelo.

Existe, com frequência, alguma sobreposição entre o primeiro e segundo princípio, Dar Direção e Especificar Formato (Phoenix, Taylor, 2024). Este último trata da definição do tipo de saída desejada, por exemplo, formato JSON ou o formato de uma foto de banco de imagens. O primeiro trata do estilo de resposta desejado, independentemente do formato, por exemplo, nomes de produtos no estilo de Steve Jobs ou uma imagem de uma reunião de negócios no estilo de Van Gogh. Quando há conflitos entre estilo e formato, geralmente é melhor resolvê-los descartando qualquer elemento que seja menos importante para o resultado final.

Quanto mais detalhado e específico for o *prompt* em relação ao formato da resposta, maior a probabilidade do modelo gerar um resultado satisfatório. Para isso, siga estas etapas ao formular seu *prompt*:

- » Seja específico sobre como a resposta deve ser organizada. Use frases como:
 - » *"Responda na forma de uma lista numerada."*
 - » *"Estruture sua resposta em três parágrafos concisos."*
 - » *"Apresente seus argumentos em um formato de tabela com colunas para 'Prós' e 'Contras'."*
- » Forneça exemplos de formato da saída. Se possível, forneça um exemplo simples da estrutura desejada logo após a instrução.
 - » *"Liste os ingredientes necessários para uma receita de bolo:"*
 - » *Farinha*
 - » *Açúcar*
 - » *..."*
- » Defina restrições de formato e conteúdo. Por exemplo:
 - » *"Limite sua resposta a 100 palavras."*
 - » *"Inclua pelo menos três exemplos concretos para cada ponto."*
 - » *"Não use jargões técnicos na explicação."*
- » Priorize as informações. Se houver elementos obrigatórios que a resposta deve conter, deixe isso claro:
 - » *"Certifique-se de incluir o ano de publicação, o autor e uma breve sinopse para cada livro."*

Veja um exemplo de *prompt* sem formatação na Figura 13 e com formatação na Figura 14.

Figura 13 - Exemplo de *prompt* sem formatação

Pergunta:

Quais são os principais pontos turísticos de Paris e seus horários de funcionamento?

Resposta:

O Arco do Triunfo é imperdível e está aberto das 10h às 23h. A Torre Eiffel é outro ícone, aberta das 9h30 às 23h45. O Museu do Louvre, com a Mona Lisa, abre das 9h às 18h, exceto às terças.

Fonte: autoria própria.

Figura 14 - Exemplo de *prompt* com formatação

Pergunta

Gostaria de uma tabela com os principais pontos turísticos de Paris e seus horários de funcionamento. A tabela deve ter as seguintes colunas: Nome do Ponto Turístico, Horário de Abertura e Horário de Fechamento.

Resposta

Nome do Ponto Turístico	Horário de Abertura	Horário de Fechamento
Arco do Triunfo	10h	23h
Torre Eiffel	9h30	23h45
Museu do Louvre	9h	18h (fechado às terças)

Fonte: autoria própria.

No primeiro *prompt* (sem formatação), o modelo responde com um texto corrido, dificultando a leitura e comparação das informações. No segundo *prompt* (com formatação), a instrução clara de formatação em tabela, incluindo os nomes das colunas, indica ao modelo que ele deve retornar a informação no formato tabular, tornando a leitura mais fácil e organizada.



2.2.3 Forneça Exemplos

Inserir exemplos diversificados é uma boa prática para orientar o modelo de PLN sobre o tipo de resposta esperada. Exemplos funcionam como uma referência prática, demonstrando de forma clara como a tarefa deve ser realizada. Eles ajudam a estabelecer um padrão de qualidade e fornecem uma base concreta para o LM na criação da resposta.

Ao fornecer exemplos, é importante cobrir muitos casos para mostrar diferentes variações possíveis da tarefa. Por exemplo, se a tarefa é escrever e-mails, você pode incluir e-mails formais, informais, de agradecimento, de reclamação, entre outros. Dessa forma, o modelo pode adaptar sua saída com base nos exemplos fornecidos, resultando em uma melhor correspondência com as expectativas do usuário. De forma geral, você deve seguir os seguintes princípios:

- » **Mostre o que você considera uma boa resposta:** use exemplos de alta qualidade e relevantes para a tarefa.
- » **Varie os exemplos:** apresente exemplos diversos que cobrem diferentes aspectos da tarefa para que o modelo possa se generalizar melhor.
- » **Use poucos exemplos, mas relevantes:** comece com um ou poucos exemplos e adicione mais se necessário. Nem sempre mais é melhor.



2.2.3.1 Diversidade de Casos

Apresente diferentes variações da tarefa para que o LM entenda a amplitude da sua aplicação. Inclua exemplos que representem casos diversos para o modelo, testando sua capacidade de lidar com nuances. Para que o modelo foque em cenários reais, apresente exemplos específicos da tarefa que deseja que ele realize.

Você deve ilustrar a amplitude da tarefa com casos simples e complexos. Abaixo temos um exemplo incorreto (Figura 15) e outro correto de *prompts* (Figura 16) que apresentam exemplos com diversidade de casos. O exemplo correto demonstra a aplicação em diferentes contextos (informal e formal), explorando a capacidade do modelo de adaptar a tradução.

Figura 15 - Prompt incorreto

Prompt:

```
"Traduza as seguintes frases para o inglês, adaptando o nível de formalidade:  
Senhora, gostaria de uma xícara de café?  
O curso de inteligência artificial é top demais!"
```

Fonte: autoria própria.

Figura 16 - Prompt correto

Prompt:

```
"Traduza as seguintes frases para o inglês, adaptando o nível de formalidade:  
Senhora, gostaria de uma xícara de café?  
O curso de inteligência artificial é top demais!"
```

Exemplos:

```
Informal: E aí, beleza? → What's up?  
Formal: Prezado Senhor, agradecemos a sua colaboração. →  
Dear Sir, we appreciate your collaboration."
```

Fonte: autoria própria.

O *prompt* correto apresenta exemplos com diferentes níveis de complexidade, explorando variações de linguagem formal e informal. Isso permite que o modelo aprenda a lidar com diferentes nuances e gere traduções mais precisas.



2.2.3.2 Detalhe e Clareza

Forneça todos os detalhes necessários para cada exemplo, incluindo informações contextuais, instruções detalhadas e saídas esperadas. Você deve utilizar uma linguagem simples e clara, evitando termos técnicos complexos que o modelo pode não compreender. Apresente os exemplos de forma consistente, com uma estrutura clara e organizada e inclua exemplos que demonstrem tanto as saídas corretas quanto as incorretas para que o modelo aprenda a identificar padrões.

O *prompt* deve conter informações completas para o modelo compreender a tarefa e o resultado esperado. Abaixo temos um exemplo incorreto e outro correto de *prompts* com detalhe e clareza. O exemplo correto define claramente a tarefa (classificação de sentimento), fornece a frase e indica a saída esperada (negativo), facilitando a compreensão do modelo (Figura 17).

Figura 17 - Exemplos de *Prompts* correto e incorreto

Prompt Incorreto

Prompt:

```
"Classifique o sentimento das frases abaixo:  
A aula de IA é ótima!  
Estou cansado de esperar na fila!"
```

Exemplos:

```
"Que dia lindo! O sentimento desta frase deve ser positivo.  
O trânsito está um caos. Esta sentença possui sentimento negativo."
```

Prompt Correto

Prompt:

```
"Classifique o sentimento das frases abaixo como positivo, negativo  
ou neutro:  
A aula de IA é ótima!  
Estou cansado de esperar na fila!"
```

Exemplos:

```
"Frase: Que dia lindo! Sentimento: Positivo  
Frase: O trânsito está um caos. Sentimento: Negativo  
Frase: O gato está dormindo. Sentimento: Neutro"
```

Fonte: autoria própria.

O *prompt* incorreto não especifica as categorias de sentimentos desejadas (positivo, negativo, neutro) e fornece exemplos de saídas esperadas com falta de clareza. O *prompt* correto define a tarefa de forma clara, as categorias de sentimentos e fornece exemplos de saídas esperadas para cada categoria, facilitando o entendimento do modelo.

2.2.4 Avalie a Qualidade

Avaliar a qualidade das respostas geradas é importante para identificar erros e refinar os *prompts* para melhorar o desempenho do modelo. A avaliação deve ser minuciosa, considerando tanto a precisão das informações quanto a adequação ao contexto e formato especificado. Identificar falhas comuns e categorizar as respostas com base em sua qualidade permite ajustar os *prompts* de maneira mais eficaz.

Sem um sistema de avaliação correto, existe o risco de utilizar *prompts* que geram respostas imprecisas, inconsistentes ou até mesmo inadequadas para a tarefa em questão. Por exemplo, se um modelo comete erros frequentes de coerência, isso pode indicar a necessidade de revisões nas instruções fornecidas. A avaliação contínua permite iterar sobre os *prompts*, aprimorando progressivamente a performance do modelo de NLP.

Avaliar e iterar sobre os *prompts* pode levar a reduções no tamanho do *prompt* e, portanto, no custo e na latência do seu sistema. Se você conseguir encontrar outro *prompt* que tenha um desempenho tão bom quanto (ou melhor), mas que use um *prompt* mais curto, você poderá reduzir o custo financeiro e o tempo de resposta do seu sistema. Em muitos casos, você descobrirá neste processo que muitos elementos de um *prompt* complexo são descartáveis.

Existem diversas abordagens para avaliar a qualidade de *prompts*, cada uma com suas próprias vantagens e desvantagens. Uma abordagem simples, porém eficaz, é a avaliação manual por meio de um sistema de classificação, onde o usuário avalia as respostas geradas pelo modelo utilizando um sistema de "joinha positivo" e "joinha negativo". Na Figura 18, é apresentado um exemplo dessa técnica, que apesar de subjetiva, permite identificar rapidamente padrões e desvios nas respostas, especialmente quando comparamos diferentes versões de um mesmo *prompt*.

Figura 18 - Avaliação da qualidade de saída do sistema

A interface de avaliação é apresentada em um formulário com fundo cinza claro e bordas arredondadas. No topo, o título "Quão legal foi minha resposta?" está em negrito. Abaixo dele, há dois botões quadrados: um verde com um ícone de polegar para cima branco e um vermelho com um ícone de polegar para baixo branco. Logo abaixo dos botões, há um campo de texto retangular com o placeholder "O que poderíamos melhorar?". Na base do formulário, há um botão ovalado laranja com o texto "ENVIAR AVALIAÇÃO" em letras maiúsculas brancas.

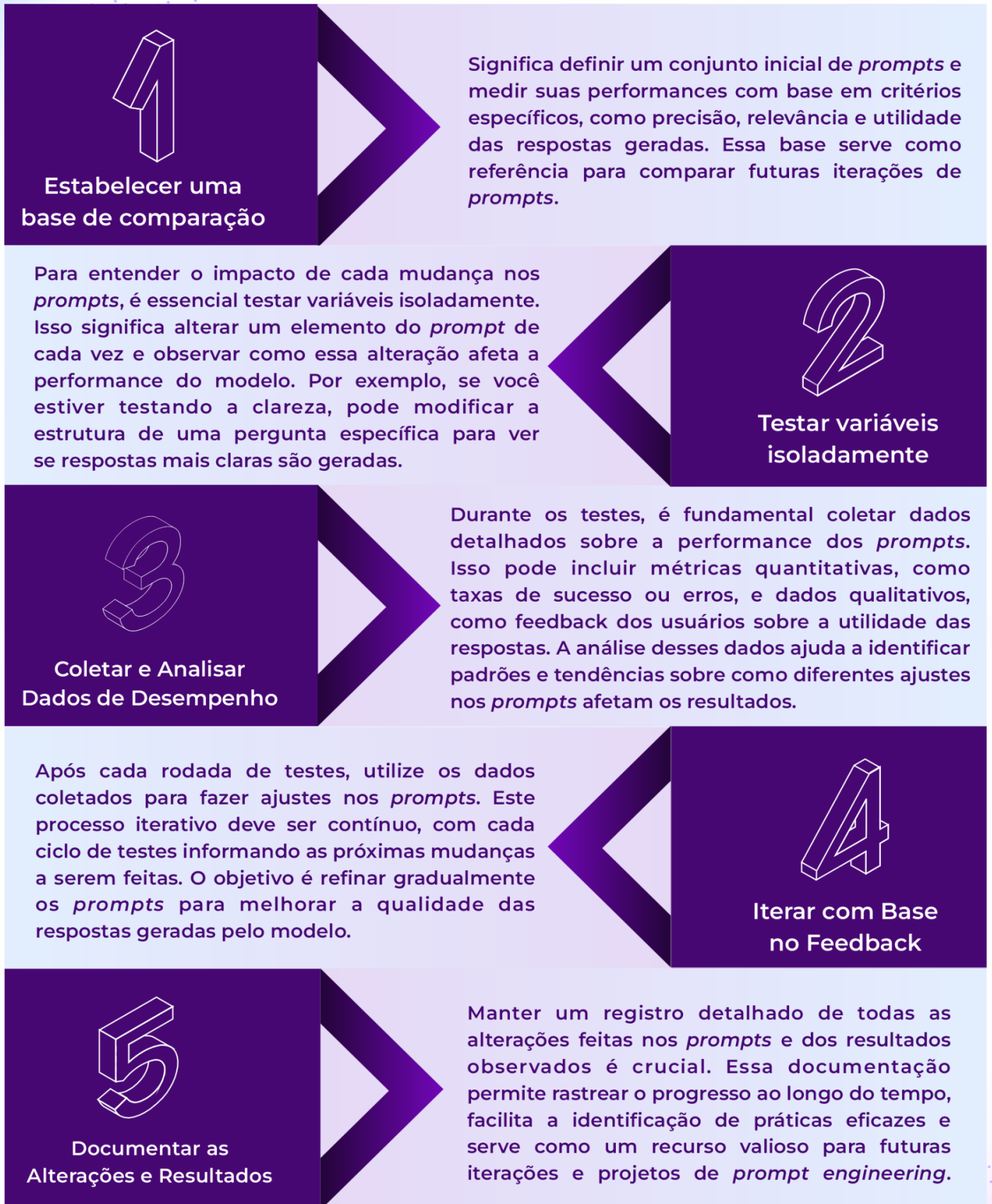
Fonte: autoria própria.

Outra abordagem é a avaliação programática, que se baseia na comparação das respostas geradas com um conjunto de respostas de referência, previamente definidas. Essa técnica é útil em tarefas que exigem maior precisão, como em casos de tradução automática ou geração de código. Métricas que medem a similaridade entre as frases geradas e as frases de referência, são frequentemente utilizadas nesse tipo de avaliação.

Por fim, a avaliação de *prompts* também pode ser realizada por meio de testes em cenários reais de uso. Nesse tipo de avaliação, o *prompt* é integrado a um sistema real e a sua eficácia é medida em relação aos objetivos de negócio, tais como redução na quantidade de chamadas ao suporte. Essa abordagem, apesar de ser a mais completa, também é a mais complexa e dispendiosa, exigindo um planejamento cuidadoso e uma infraestrutura robusta para coleta e análise de dados.

A escolha da melhor abordagem para avaliação de *prompts* depende do contexto específico da tarefa, dos recursos disponíveis e dos objetivos a serem alcançados. A estratégia de testes em engenharia de *prompts* envolve a criação de um processo estruturado para avaliar e refinar *prompts* de maneira contínua e interativa. Essa abordagem é importante para identificar o que funciona e o que não funciona, permitindo melhorias contínuas na interação com LMs. A seguir, são detalhados os principais componentes desta estratégia na Figura 19.

Figura 19 - Principais componentes da estratégia de testes em engenharia de *prompts*



Fonte: autoria própria.

Existem diversos projetos no GitHub focados na avaliação de *prompts* e LMs. Esses projetos oferecem ferramentas, *benchmarks* (trata-se de um padrão de referência) e *frameworks* (coleções de componentes de *software*) que auxiliam na análise e melhoria dos *prompts* utilizados com LLMs. Aqui estão alguns dos principais projetos:

1. *LM Evaluation Harness*

- » **Descrição:** *framework* de avaliação com poucos exemplos para LMs, que fornece uma variedade de tarefas e *benchmarks*.
- » **Link:** [EleutherAI/lm-evaluation-harness](https://github.com/EleutherAI/lm-evaluation-harness)

2. *OpenAI Evals*

- » Descrição: avaliação de LMs que permite criar, compartilhar e executar avaliações para LMs, facilitando a análise sistemática de *prompts* e respostas.
- » **Link:** <https://github.com/openai/evals>

3. *DeepEval*

- » **Descrição:** *framework* para realizar testes em LMs, permitindo identificar pontos fortes e fracos nas capacidades de geração de texto.
- » **Link:** <https://github.com/confident-ai/deepeval>

4. *Langfuse*

- » **Descrição:** plataforma *open-source* (de código aberto) que ajuda na observabilidade e avaliação de LMs. Ela fornece ferramentas para coletar, visualizar e analisar interações com LMs, facilitando a identificação de problemas e a medição de desempenho.
- » **Link:** <https://github.com/langfuse/langfuse>

5. *LangSmith*

- » **Descrição:** ferramenta desenvolvida pelo *LangChain* que se concentra na avaliação, monitoramento e melhoria contínua de LMs. Ela oferece uma plataforma abrangente para rastrear o desempenho dos LMs em várias tarefas, com foco na criação e ajuste de fluxos de trabalho de PLN.
- » **Link:** <https://www.langchain.com/langsmith>



2.2.5 Faça a Divisão do Trabalho

Suponha que você precisa explicar um trabalho complicado para alguém. Se você explicar tudo de uma vez, a pessoa pode se confundir. Mas se você dividir o trabalho em etapas menores e explicar cada uma com clareza, fica mais fácil de entender, certo? É exatamente isso que o **princípio da Divisão do Trabalho** faz na Engenharia de *Prompts*.

A ideia central da divisão do trabalho é bastante intuitiva, refletindo uma prática comum em engenharia: a decomposição de problemas complexos em partes menores e mais gerenciáveis. No contexto da engenharia de *prompts*, isso se traduz em dividir uma tarefa complexa, que exige várias etapas de processamento de informação, em uma sequência de *prompts* menores e interdependentes. Cada *prompt* individual se concentra em uma parte específica da tarefa, e a saída de um *prompt* serve como entrada para o próximo, criando assim um fluxo de trabalho encadeado.

Em vez de pedir à IA para resolver um problema gigante de uma só vez, você pode separar esse problema em pequenas tarefas. Cada tarefa recebe um "comando" específico (o *prompt*), e a resposta de uma tarefa vira informação para a próxima, como um quebra-cabeça.

2.2.5.1 Exemplos de Aplicação da Divisão do Trabalho

A decomposição de tarefas é uma estratégia importante na engenharia de *prompt*, onde um problema complexo é dividido em etapas menores e mais gerenciáveis. Isso ajuda a criar *prompts* mais precisos e eficazes. Nas Figuras 20 e 21, a seguir, estão disponíveis dois exemplos práticos de como essa abordagem pode ser aplicada.

Figura 20 - Criação de um *prompt* para geração de resumos de textos longos

Exemplo 1: Criação de um *Prompt* para Geração de Resumos de Textos Longos

Vamos considerar o desafio de criar um *prompt* para que uma IA gere resumos de textos longos de forma clara e concisa. A tarefa pode ser decomposta da seguinte maneira:

1

Identificação dos pontos principais

O primeiro passo é criar um *prompt* que instrua a IA a identificar os pontos principais do texto. Por exemplo: "Identifique os três pontos principais abordados neste texto." Isso direciona a IA a focar nas informações mais relevantes.

2

Elaboração de resumos parciais

Com os pontos principais identificados, o próximo *prompt* pode pedir à IA para elaborar pequenos resumos de cada ponto principal. Um exemplo de *prompt* seria: "Resuma cada um dos pontos principais em uma ou duas frases." Isso permite que o texto seja dividido em seções menores, facilitando o processamento.

3

Combinação e refinamento do resumo

Após a elaboração dos resumos parciais, o *prompt* final pode instruir a IA a combinar esses resumos em um texto único e coeso. Um exemplo seria: "Combine os resumos em um único parágrafo coeso que capture a essência do texto original."

4

Revisão e ajustes finais

Finalmente, um *prompt* adicional pode ser utilizado para revisar o resumo e ajustar a clareza e a concisão. Por exemplo: "Revise o parágrafo para garantir que esteja claro e conciso, e remova qualquer informação desnecessária."

Fonte: autoria própria.

Essa decomposição em etapas permite que a IA execute a tarefa complexa de maneira mais eficaz e controlada, resultando em resumos de alta qualidade (Figura 21).

Figura 21 - Desenvolvimento de um assistente virtual para responder perguntas complexas

Exemplo 2: Desenvolvimento de um Assistente Virtual para Responder Perguntas Complexas

Vamos imaginar que o objetivo seja criar um assistente virtual que responda perguntas complexas, como "Como funciona a mudança climática?". A tarefa pode ser decomposta da seguinte forma:

1

Divisão da pergunta em subperguntas

O primeiro *prompt* pode instruir a IA a dividir a pergunta complexa em subperguntas mais simples.

a. Por exemplo: "Divida a pergunta 'Como funciona a mudança climática?' em três subperguntas mais simples.

I. "O que é efeito estufa?"

II. "Quais são os principais gases de efeito estufa?"

III. "Como o aquecimento global afeta o clima?"

2

Resposta a cada subpergunta

Para cada subpergunta, um novo *prompt* pode ser criado para gerar respostas detalhadas.

a. Por exemplo: "Explique o que é o efeito estufa." e assim por diante para cada subpergunta.

3

Integração das respostas

Com as respostas às subperguntas prontas, o próximo passo é integrá-las em uma única resposta coesa.

a. Um exemplo de *prompt* seria: "Combine as respostas às subperguntas em uma explicação coesa de como funciona a mudança climática."

4

Simplificação e ajuste da resposta final

finalmente, pode-se utilizar um *prompt* para simplificar e ajustar a resposta final, garantindo que ela seja acessível para o público-alvo.

a. Por exemplo: "Simplifique a resposta para que seja compreensível por estudantes do ensino médio."

Fonte: autoria própria.



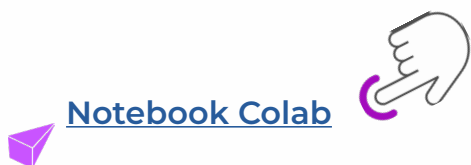
2.2.5.2 Por que Dividir o Trabalho Ajuda a IA?

Pense em como ensinamos algo novo a uma criança. Se a gente apresentar um monte de informação complicada de uma vez só, ela provavelmente vai ficar confusa e desistir. Mas se dividirmos a tarefa em etapas menores, explicando cada passo com paciência, a criança terá muito mais chances de aprender e se sentir motivada.

Com a IA, a lógica é a mesma! A "Divisão do Trabalho" funciona como um guia para a IA, tornando tarefas complexas muito mais fáceis de entender e resolver. Em vez de enfrentar um desafio gigante de uma vez só, a IA pode focar em cada etapa com atenção, o que traz uma série de benefícios:

- » **Melhora da compreensibilidade:** ao decompor um problema complexo em partes menores, o modelo de IA pode processar e entender as informações de forma mais eficaz.
- » **Aumento da confiabilidade:** concentrando-se em uma única tarefa por vez, o modelo tem maior probabilidade de gerar resultados precisos e consistentes, reduzindo a ocorrência de "alucinações".
- » **Maior controle e visibilidade:** a divisão do trabalho torna o processo de interação com a IA mais transparente, permitindo que os usuários acompanhem o raciocínio da IA passo a passo e identifiquem as causas de erros.
- » **Possibilidade de reutilização:** os *prompts* individuais, desenvolvidos para cada etapa da tarefa, podem ser reutilizados em outros contextos, otimizando o processo de desenvolvimento e ampliando as possibilidades de aplicação da IA.

Dividir o trabalho é como dar um mapa para a IA, guiando-a passo a passo para encontrar soluções. Essa técnica funciona para diversas tarefas, desde escrever histórias criativas até resolver problemas complexos, reduzindo as tarefas em outras menores facilmente solucionadas pelos modelos.



Este colab sobre Prompt Engineering está dividido em três tópicos principais que cobrem desde os conceitos básicos até a aplicação prática avançada. No primeiro tópico, o foco está na introdução e compreensão da importância dos prompts, com exercícios para análise e ajustes. O segundo tópico aprofunda técnicas avançadas, explorando o uso de condicionais e variáveis, com a criação e comparação de prompts complexos. O terceiro tópico é voltado para a aplicação prática, incentivando a implementação em projetos reais, além de revisões contínuas e planejamento de estudos futuros. O texto mantém uma estrutura clara e prática, ajudando o leitor a acompanhar seu progresso.

Instalações

Explicação Simples do Comando: `!pip install langchain_groq langchain`. Esse comando está sendo usado para instalar dois programas ou "pacotes" chamados `langchain_groq` e `langchain`. As versões foram fixadas (usando `==` após o nome do pacote, como em `langchain_groq==0.2.1`) para evitar problemas causados por mudanças inesperadas em versões posteriores.

- » **Pip:** É como um "instalador" de programas específicos para quem trabalha com a linguagem de programação Python. Ele ajuda a trazer esses programas para o seu computador ou ambiente de trabalho, para que você possa usá-los.
- » **langchain:** biblioteca Python popular para desenvolvimento de aplicações de IA, especialmente para trabalhar com Large Language Models (LLMs). Ela fornece uma estrutura para conectar, gerenciar e orquestrar diferentes LLMs e ferramentas, facilitando a construção de aplicações de NLP. Esses pacotes são como peças de LEGO que você junta para criar um programa mais inteligente.
- » **langchain_groq:** fornece uma integração entre a biblioteca LangChain e o processador de IA da Groq. Ele permite que modelos de linguagem grandes sejam executados no hardware da Groq, potencialmente oferecendo melhorias significativas de desempenho para certos tipos de tarefas.

Ao rodar o comando `!pip install`, o computador vai procurar essas ferramentas (`langchain_groq` e `langchain`) na internet, baixar e instalar para que você possa usá-las no seu projeto.

O símbolo `!"` utilizado no início do comando é do **Google Colab** (ferramenta online para rodar código) que informa que você deseja rodar um comando de instalação no ambiente.

```
[ ] !pip install langchain_groq==0.2.1 langchain==0.3.7
```

```
⇒ Collecting langchain_groq==0.2.1
```

```
    Downloading langchain_groq-0.2.1-py3-none-any.whl.metadata (2.9 kB)
```

```
Collecting langchain==0.3.7
```

```
    Downloading langchain-0.3.7-py3-none-any.whl.metadata (7.1 kB)
```

```
Collecting groq<1,>=0.4.1 (from langchain_groq==0.2.1)
```

```
    Downloading groq-0.18.0-py3-none-any.whl.metadata (14 kB)
```

```
Requirement already satisfied: langchain-core<0.4.0,>=0.3.15 in /usr/local/lib/python3.11/dist-packages (from langchain_groq==0.2.1) (0.3.33)
```

```
Requirement already satisfied: PyYAML>=5.3 in /usr/local/lib/python3.11/dist-packages (from langchain==0.3.7) (6.0.2)
```

continua

Requirement already satisfied: SQLAlchemy<3,>=1.4 in /usr/local/lib/python3.11/dist-packages (from langchain==0.3.7) (2.0.37)

Requirement already satisfied: aiohttp<4.0.0,>=3.8.3 in /usr/local/lib/python3.11/dist-packages (from langchain==0.3.7) (3.11.12)

Requirement already satisfied: langchain-text-splitters<0.4.0,>=0.3.0 in /usr/local/lib/python3.11/dist-packages (from langchain==0.3.7) (0.3.5)

Collecting langsmith<0.2.0,>=0.1.17 (from langchain==0.3.7)

 Downloading langsmith-0.1.147-py3-none-any.whl.metadata (14 kB)

Requirement already satisfied: numpy<2,>=1 in /usr/local/lib/python3.11/dist-packages (from langchain==0.3.7) (1.26.4)

Requirement already satisfied: pydantic<3.0.0,>=2.7.4 in /usr/local/lib/python3.11/dist-packages (from langchain==0.3.7) (2.10.6)

Requirement already satisfied: requests<3,>=2 in /usr/local/lib/python3.11/dist-packages (from langchain==0.3.7) (2.32.3)

Requirement already satisfied: tenacity!=8.4.0,<10,>=8.1.0 in /usr/local/lib/python3.11/dist-packages (from langchain==0.3.7) (9.0.0)

Requirement already satisfied: aiohappyeyeballs>=2.3.0 in /usr/local/lib/python3.11/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain==0.3.7) (2.4.4)

Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/python3.11/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain==0.3.7) (1.3.2)

Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.11/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain==0.3.7) (25.1.0)

Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.11/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain==0.3.7) (1.5.0)

Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.11/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain==0.3.7) (6.1.0)

Requirement already satisfied: propcache>=0.2.0 in /usr/local/lib/python3.11/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain==0.3.7) (0.2.1)

Requirement already satisfied: yarll<2.0,>=1.17.0 in /usr/local/lib/python3.11/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain==0.3.7) (1.18.3)

Requirement already satisfied: anyio<5,>=3.5.0 in /usr/local/lib/python3.11/dist-packages (from groq<1,>=0.4.1->langchain_groq==0.2.1) (3.7.1)

Requirement already satisfied: distro<2,>=1.7.0 in /usr/local/lib/python3.11/dist-packages (from groq<1,>=0.4.1->langchain_groq==0.2.1) (1.9.0)

Requirement already satisfied: httpx<1,>=0.23.0 in /usr/local/lib/python3.11/dist-packages (from groq<1,>=0.4.1->langchain_groq==0.2.1) (0.28.1)

Requirement already satisfied: sniffio in /usr/local/lib/python3.11/dist-packages (from groq<1,>=0.4.1->langchain_groq==0.2.1) (1.3.1)

Requirement already satisfied: typing-extensions<5,>=4.10 in /usr/local/lib/python3.11/dist-packages (from groq<1,>=0.4.1->langchain_groq==0.2.1) (4.12.2)

continua

```

lib/python3.11/dist-packages (from groq<1,>=0.4.1->langchain_groq==0.2.1)
(4.12.2)
Requirement already satisfied: jsonpatch<2.0,>=1.33 in /usr/local/lib/py-
thon3.11/dist-packages (from langchain-core<0.4.0,>=0.3.15->langchain_
groq==0.2.1) (1.33)
Requirement already satisfied: packaging<25,>=23.2 in /usr/local/lib/py-
thon3.11/dist-packages (from langchain-core<0.4.0,>=0.3.15->langchain_
groq==0.2.1) (24.2)
Requirement already satisfied: orjson<4.0.0,>=3.9.14 in /usr/local/lib/py-
thon3.11/dist-packages (from langsmith<0.2.0,>=0.1.17->langchain==0.3.7)
(3.10.15)
Requirement already satisfied: requests-toolbelt<2.0.0,>=1.0.0 in /usr/
local/lib/python3.11/dist-packages (from langsmith<0.2.0,>=0.1.17->lang-
chain==0.3.7) (1.0.0)
Requirement already satisfied: annotated-types>=0.6.0 in /usr/local/lib/
python3.11/dist-packages (from pydantic<3.0.0,>=2.7.4->langchain==0.3.7)
(0.7.0)
Requirement already satisfied: pydantic-core==2.27.2 in /usr/local/lib/
python3.11/dist-packages (from pydantic<3.0.0,>=2.7.4->langchain==0.3.7)
(2.27.2)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/
python3.11/dist-packages (from requests<3,>=2->langchain==0.3.7) (3.4.1)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/
dist-packages (from requests<3,>=2->langchain==0.3.7) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/py-
thon3.11/dist-packages (from requests<3,>=2->langchain==0.3.7) (2.3.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/py-
thon3.11/dist-packages (from requests<3,>=2->langchain==0.3.7) (2025.1.31)
Requirement already satisfied: greenlet!=0.4.17 in /usr/local/lib/py-
thon3.11/dist-packages (from SQLAlchemy<3,>=1.4->langchain==0.3.7) (3.1.1)
Requirement already satisfied: httpcore==1.* in /usr/local/lib/py-
thon3.11/dist-packages (from httpx<1,>=0.23.0->groq<1,>=0.4.1->langchain_
groq==0.2.1) (1.0.7)
Requirement already satisfied: h11<0.15,>=0.13 in /usr/local/lib/py-
thon3.11/dist-packages (from httpcore==1.*->httpx<1,>=0.23.0->groq<1,>=
0.4.1->langchain_groq==0.2.1) (0.14.0)
Requirement already satisfied: jsonpointer>=1.9 in /usr/local/lib/py-
thon3.11/dist-packages (from jsonpatch<2.0,>=1.33->langchain-core<0.4.0,>=
0.3.15->langchain_groq==0.2.1) (3.0.0)
Downloading langchain_groq-0.2.1-py3-none-any.whl (14 kB)
Downloading langchain-0.3.7-py3-none-any.whl (1.0 MB)
_____ 1.0/1.0 MB 34.3 MB/s eta 0:00:00
Downloading groq-0.18.0-py3-none-any.whl (121 kB)
_____ 121.9/121.9 kB 8.4 MB/s eta 0:00:00
Downloading langsmith-0.1.147-py3-none-any.whl (311 kB)
_____ 311.8/311.8 kB 19.4 MB/s eta
0:00:00

```

continua

```
Installing collected packages: langsmith, groq, langchain_groq, langchain
Attempting uninstall: langsmith
  Found existing installation: langsmith 0.3.6
  Uninstalling langsmith-0.3.6:
    Successfully uninstalled langsmith-0.3.6
Attempting uninstall: langchain
  Found existing installation: langchain 0.3.17
  Uninstalling langchain-0.3.17:
    Successfully uninstalled langchain-0.3.17
Successfully installed groq-0.18.0 langchain-0.3.7 langchain_groq-0.2.1
langsmith-0.1.147
```

Nós iremos utilizar um LLM chamado Llama3.

Lembrando que, para utilizar um Grande Modelo de Linguagem é necessário ter capacidade de processamento e, para tal, precisamos instanciar esse LLM em uma GPU (Unidade de Processamento Gráfico, do inglês Graphics Processing Unit). Uma GPU é um tipo de processador com alta capacidade de processamento computacional (apto para a realização de cálculos complexos e paralelos) em especial para a renderização de imagens gráficas de qualidade. No nosso caso, como estamos limitados ao ambiente do Colab no qual muitas vezes é necessário esperar a GPU conseguir calcular as saídas dos programas, resolvemos utilizar uma plataforma que aumenta a capacidade de processamento chamada **GROQ**.

Dessa forma, precisaremos obter uma chave denominada 'GROQ_API_KEY' diretamente do site da GROQ. Essa chave é uma forma de controle de acesso adotada pelos detentores da plataforma afim de contabilizar quem está utilizando o processamento. Simplificando, é como se fosse a chave de uma casa, essa chave só consegue abrir essa casa em específico, certo? Então, o **GROQ** usa esse artifício para conseguir abrir uma GPU específica com essa chave que você está fornecendo para ele.

Como adquirir a chave '**GROQ_API_KEY**'?

1. Navegue até o site do [Groq Cloud](#) e cadastre-se.
2. Depois de fazer login em uma conta, navegue até a página [Chaves de API](#) no painel de navegação esquerdo.
3. Clique em Criar chave de API e dê um nome a ela.
4. Clique em Enviar.
5. Copie a chave gerada para sua área de transferência.

Especificações e importações iniciais:

```
[ ] from langchain_groq import ChatGroq # Importa a classe ChatGroq da biblioteca langchain_groq, usada para interagir com modelos Groq.
import time # Importa a biblioteca time, que pode ser usada para funções relacionadas a manipulação de tempo.
from langchain_core.prompts import ChatPromptTemplate # Importa ChatPromptTemplate para criação de prompts customizados em chat.
from langchain.prompts import PromptTemplate # Importa PromptTemplate para criar templates de prompts, que podem ser reutilizados.
from langchain_core.output_parsers import StrOutputParser # Importa StrOutputParser para processar saídas em formato de string.
from langchain_core.output_parsers import JsonOutputParser # Importa JsonOutputParser para processar saídas no formato JSON.

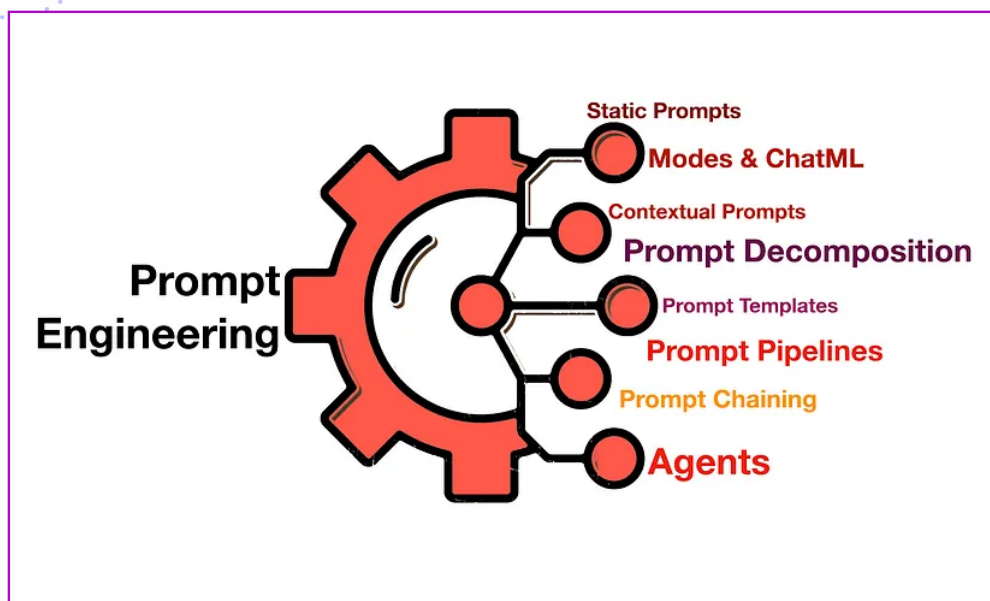
import os # Importa o módulo os, que permite interagir com o sistema operacional.
from google.colab import userdata # Importa userdata do Google Colab para manipular dados do usuário no ambiente Colab.
os.environ["GROQ_API_KEY"] = userdata.get('GROQ_API_KEY') # Define a variável de ambiente GROQ_API_KEY com a chave de API do Groq, recuperada do Colab.

GROQ_LLM = ChatGroq( # Instancia o modelo de linguagem Groq.
    model="llama3-8b-8192", # Especifica o modelo a ser utilizado, no caso, o "llama3-8b-8192".
)
```

Llama3 - Características

Llama3 é um dos modelos de linguagem avançados da família Llama (Large Language Model Meta AI), projetados para fornecer respostas altamente precisas em tarefas de processamento de linguagem natural (NLP). O **Llama3-8b-8192**, em particular, é otimizado para trabalhar com grandes volumes de dados e possui 8 bilhões de parâmetros, o que permite uma compreensão profunda e capacidade de geração de texto. Além disso, ele pode lidar com sequências mais longas de entrada, até 8192 tokens, oferecendo flexibilidade em diversas aplicações como tradução, respostas a perguntas e geração de conteúdo. Ele é amplamente utilizado em ambientes que exigem modelos de alta performance com aplicações em chatbots, assistentes virtuais e análise de texto em grande escala.

Figura 22 - Prompt Engineering



Fonte Acesso em 22 de dezembro

A engenharia de *prompts*, ou prompt engineering, é uma disciplina emergente no campo da inteligência artificial que se concentra em otimizar a interação entre humanos e modelos de linguagem avançados, como os baseados na arquitetura de *Transformer*. Esta área ganhou destaque com a popularização dos modelos de linguagem generativa, tais como GPT-3 e suas versões subsequentes, que demonstraram habilidades surpreendentes em gerar texto que pode ser indistinguível do escrito por humanos em muitos contextos.

A importância da engenharia de *prompts* reside na sua capacidade de extrair o melhor desempenho possível dos modelos de IA. Com os *prompts* corretos, é possível direcionar o modelo para produzir respostas mais precisas, criativas e úteis, adaptadas às necessidades específicas do usuário. Por exemplo, em aplicações educacionais, *prompts* bem elaborados podem ajudar na criação de materiais de aprendizado personalizados e eficazes. Na indústria, podem otimizar a automação de tarefas e a análise de dados.

Além disso, a relevância da engenharia de *prompts* também se manifesta na sua capacidade de reduzir a incidência de respostas inadequadas ou imprecisas, garantindo que a interação entre humanos e máquinas seja segura, ética e eficiente. Portanto, entender e aplicar técnicas de engenharia de *prompts* é importante para qualquer um que deseje explorar ao máximo as capacidades dos sistemas de IA conversacional, maximizando assim o retorno sobre o investimento em tecnologias de linguagem natural.

Antes de qualquer coisa...

O que é *prompt template*?

Um "prompt template" é um formato predefinido usado para estruturar prompts em aplicações que utilizam modelos de linguagem. Ele serve como um molde que orienta o usuário na formulação de entradas (*prompts*) para obter respostas mais eficazes e relevantes do modelo.

Componentes típicos de um *prompt template*:

1. **Instrução Principal:** A parte do *prompt* que define claramente o que você deseja obter como resposta. Por exemplo, "Explique o conceito de inflação."
2. **Contexto:** Informações adicionais que ajudam a situar a pergunta ou tarefa dentro de um cenário específico. Por exemplo, "Considere o impacto da inflação no Brasil nos últimos cinco anos."
3. **Formato da Resposta:** Indicações sobre como a resposta deve ser estruturada. Pode incluir formato de parágrafos, listas, tabelas, etc. Por exemplo, "Forneça sua resposta em forma de parágrafos."
4. **Restrições ou Limitações:** Qualquer limitação específica sobre o conteúdo ou estilo da resposta. Por exemplo, "Evite usar termos técnicos complexos."

Exemplo de *Prompt Template*

```
Instrução Principal: [Descreva o problema que você está tentando resolver]
Contexto: [Forneça qualquer informação adicional relevante]
Formato da Resposta: [Descreva como você deseja que a resposta seja apresentada]
Restrições: [Indique quaisquer limitações ou restrições para a resposta]
```

Exemplo Prático

```
Instrução Principal: Explique o conceito de Machine Learning.
Contexto: Considere um público de alunos do ensino médio que estão sendo introduzidos à ciência da computação.
```


Formato da Resposta: Use parágrafos curtos e exemplos práticos.

Restrições: Evite usar equações matemáticas complexas.

Imagine o seguinte cenário...

Você está triste porque sua festa de aniversário está chegando e você não consegue pensar em como escrever o seu convite. Calma! Temos a solução: vamos usar **modelos generativos** (inteligência artificial) para escrever o seu convite de aniversário.

Esses modelos de IA precisam ser orientados sobre **o que fazer** e **como agir**. Para isso, utilizamos algo chamado **prompt template**, que é como uma receita que explica para a IA o que deve ser feito. Dentro do template, usamos alguns **tokens especiais** (ou marcadores), que ajudam a IA a entender o contexto e as instruções que damos. Esses tokens servem como guias para o modelo.

Vamos agora entender melhor cada token mencionado:

1. **<|begin_of_text|>**: Esse token marca o **início do texto**. Ele indica para a IA que, a partir desse ponto, começa o conteúdo importante que ela deve prestar atenção. Sem ele, a IA pode não saber onde começar a leitura.
2. **<|start_header_id|>system<|end_header_id|>**: Esse é um conjunto de tokens que envolvem a palavra **system**. Ele funciona assim:
 - » **<|start_header_id|>**: Esse token marca o **início de uma seção especial** no template, que será usada para dar instruções gerais à IA.
 - » **system**: Dentro desses tokens, nós dizemos à IA que estamos passando instruções do "sistema". Isso significa que estamos explicando como a IA deve agir ou se comportar.
 - » **<|end_header_id|>**: Esse token indica o **fim dessa seção de instruções**. Ele avisa à IA que as orientações terminaram e que ela deve começar a realizar a tarefa conforme foi instruída.

Esses tokens são como sinais de trânsito para a IA, dizendo onde ela deve começar a ler, onde estão as instruções importantes, e onde ela deve parar de prestar atenção nas orientações.

No nosso caso, dentro do campo de system no template, estamos dando orientações específicas à IA sobre como ela deve gerar o convite de aniversário para você.

Com isso, conseguimos direcionar o nosso LLM para o contexto desejado

```
[ ] prompt = PromptTemplate(
    template="""<|begin_of_text|><|start_header_id|>system<|end_header_id|>
    Atue com a personalidade de um especialista amigável.
    Responda perguntas e ajude a resolver problemas para estudantes iniciantes
    em PLN, que não são da área de Tecnologia.
    O estilo de escrita é amigável, direcionado a iniciantes em PLN.
    Você deve levar em consideração o seguinte contexto: Os usuários são par-
    ticipantes
    de uma especialização em PLN, aprendendo o básico dos modelos de linguagem
    grandes (LLMs).
    As respostas devem ser detalhadas e completas, sem necessidade de citar
    fontes externas.
    Evite um linguajar técnico e utilize exemplos simples e fáceis de enten-
    der.

    <|eot_id|><|start_header_id|>user<|end_header_id|>
    USER_PROMPT:\n\n {USER_PROMPT} \n\n
    <|eot_id|>
    <|start_header_id|>assistant<|end_header_id|>
    """,
    input_variables=["USER_PROMPT"],
)
```

O que é o **USER_PROMPT**?

No código acima, **USER_PROMPT** é um espaço reservado para **inserir a pergunta ou o problema que o usuário quer resolver**.

Imagine que você está em uma conversa com uma inteligência artificial (IA), como se fosse um assistente que responde suas dúvidas. A parte **USER_PROMPT** é onde você escreve a sua pergunta, como por exemplo: "O que é PLN (Processamento de Linguagem Natural)?".

Quando o código é executado, o **USER_PROMPT** é substituído pela pergunta real que o usuário faz. Assim, a IA vai entender o que precisa responder e dar uma resposta detalhada e amigável, como foi instruído no resto do código.

Resumindo, o **USER_PROMPT** é o lugar onde entra a dúvida ou questão do usuário, para que a IA possa entender o que deve responder.

Agora nós vamos criar uma *chain*: Trata-se de uma série de operações, em ordem, que permite ao LLM a realização de tarefas como manipulação de dados, consultas, inferências e outros. Observe o comando:

```
llm_chain = prompt | GROQ_LLM | StrOutputParser()
```

Essa *chain* pode ser explicada como uma sequência de etapas, onde o computador processa uma "pergunta" ou "comando" que você quer realizar.

Vamos entender cada parte:

1. prompt:

Aqui, você começa com algo que deseja perguntar ou mandar o computador fazer. É como se você desse uma ordem ou fizesse uma pergunta para o sistema.

2. GROQ_LLM:

Depois de receber sua pergunta, o computador usa um "cérebro artificial" (ou seja, um LLM chamado **GROQ_LLM**) para pensar e gerar uma resposta ou realizar uma ação baseada na pergunta.

3. StrOutputParser():

Por fim, a resposta gerada pelo "cérebro artificial" precisa ser formatada para que você possa entendê-la facilmente. O **StrOutputParser()** faz essa conversão e entrega a resposta em formato de texto simples.

Esse código pega uma pergunta que você fez, faz o computador "pensar" usando uma IA, e depois organiza a resposta de uma maneira que faça sentido e seja fácil de entender.

```
[ ] llm_chain = prompt | GROQ_LLM | StrOutputParser()
```

Vamos passar o nosso *prompt* de fato, ou seja, aqui que fazemos a nossa solicitação.

```
[ ] USER_PROMPT = """Olá, tudo bem? Eu preciso que você crie um convite de aniversário para minha festa
```

```
de 54 anos. Meu nome é José Marcio e o tema da festa vai ser colgate, pois  
sou um grande fã da marca.  
Será dia 26/09 às 16:20 na minha residência.  
"""
```

Agora precisamos chamar aquela *chain* que criamos e salvamos o resultado dela, para isso vamos usar esse trecho de código:

```
result = llm_chain.invoke({"USER_PROMPT": USER_PROMPT})
```

Esse código é responsável por usar uma **cadeia de etapas** que já configuramos para processar um comando ou pergunta e, em seguida, devolver o resultado. Vamos explicar linha por linha:

1. `result = llm_chain.invoke({"USER_PROMPT": USER_PROMPT})`:

- » Aqui, estamos dizendo ao computador para pegar o que o usuário escreveu (ou seja, a pergunta ou comando, que chamamos de `USER_PROMPT`), e passar essa pergunta para a nossa "cadeia de pensamento" chamada `llm_chain`.
- » O `llm_chain` é uma sequência de etapas que envolve entender a pergunta, pensar sobre ela usando um modelo de IA (inteligência artificial), e produzir uma resposta.
- » O resultado dessa "conversa" entre a pergunta do usuário e o modelo de IA é armazenado na variável `result`.

2. `print(result)`:

Depois que o computador processa a pergunta e gera uma resposta, nós usamos o comando `print(result)` para mostrar essa resposta ao usuário. Ou seja, essa linha faz o computador exibir na tela o resultado da pergunta feita.

Esse código pega a pergunta que o usuário escreveu, passa por um modelo de IA que gera uma resposta, e, finalmente, mostra essa resposta ao usuário.

```
[ ] result = llm_chain.invoke({"USER_PROMPT": USER_PROMPT})  
  
print(result)
```

```
➞ Olá José Marcio! Feliz aniversário em advance!
```

continua

Ah, um tema de colgate, interessante! Vou criar um convite para você com todo o estilo e glamour da marca.

Aqui está um exemplo de convite que eu criei especialmente para você:

****"Colgate Celebration: 54 anos de sabão e sorrisos!"****

"Você é convidado(a) para celebrar o 54º aniversário de José Marcio, um grande fã da marca Colgate!

Data: 26 de setembro às 16:20

Local: Residência de José Marcio

Venha se divertir e se lembrar de muitos anos de sabão e sorrisos com a gente!

RSVP: [insira seu contato]

Espero que você goste! Se precisar de qualquer ajuste, basta me dizer.

Boa sorte com a festa e que seja um dia incrível!

Você consegue perceber que se mudarmos o contexto que passamos ao modelo... ele vai nos responder totalmente diferente?

Vamos brincar um pouco com isso agora.

Prompt Triste

Vamos passar um *prompt* triste para identificarmos as mudanças na resposta do modelo.

```
[ ] USER_PROMPT = """Eu quero que você crie uma história sobre um pássaro aprendendo a voar.
```

```
A narrativa deve ser impregnada com tons dramáticos, tristes, melancólicos e sombrios.
```

```
A história deve explorar os desafios enfrentados pelo pássaro, enfatizando sentimentos
```

```
de solidão e frustração diante das dificuldades. O ambiente e as interações devem reforçar
```

continua

uma atmosfera de desesperança e isolamento, ressaltando a luta emocional do personagem principal.

"""

```
result = llm_chain.invoke({"USER_PROMPT": USER_PROMPT})
```

```
print(result)
```

➞ A história do pássaro que sonhava voar!

No fundo de uma floresta escura e sombria, havia um pássaro chamado Ares. Ele era um jovem pássaro, com plumas pretas e brancas, que ansiava voar mais do que qualquer outra coisa. Ares sempre imaginava as liberdades que a asa lhe daria, o sentimento de glória e a capacidade de explorar o mundo acima das árvores.

Mas, infelizmente, Ares não sabia voar. Ele tentava balançar as asas, mas não conseguia levantar do chão. Cada tentativa era uma derrota, e ele sentia-se cada vez mais frustrado e desanimado.

Ares passava os dias observando os outros pássaros voando livremente, sentindo-se cada vez mais isolado e oprimido. Ele começava a questionar se era mesmo possível aprender a voar. Talvez ele não tivesse sido feito para voar. Talvez fosse apenas um pássaro fracassado.

Uma noite, Ares se refugiou em um galho alto, chorando em silêncio. Ele se sentia tão sozinho, tão desamparado. As árvores ao redor pareciam observá-lo com desdém, como se estivessem dizendo: "Você não é digno de voar".

No entanto, Ares não desistiu. Ele começou a treinar em segredo, escondido dos outros pássaros. Ele fazia exercícios para fortalecer as asas, tentava voar em sonho e se inspirava nas histórias de outros pássaros que haviam aprendido a voar.

Mas a solidão e a frustração continuavam a se apoiar sobre ele como uma nuvem negra. Ele começou a duvidar de si mesmo, a questionar se era mesmo possível aprender a voar.

Uma tempestade se aproximava, e as nuvens escuras obscureciam a luz do sol. Ares se sentiu mais desamparado do que nunca. Ele começou a acreditar que jamais aprenderia a voar, que era condenado a viver no chão.

No entanto, um vento forte soprou, e Ares sentiu que seu destino estava próximo. Ele inspirou profundamente, esticou as asas e... voou! Era um voo vacilante e hesitante, mas era um voo!

continua

Ares estava no ar! Ele se sentia livre, poderoso e emocionado. A tempestade que havia se aproximado começou a passar, e o sol brilhou novamente.

Ares voou longe, sentindo o vento nas plumas e a liberdade nas asas. Ele se sentiu como um novo pássaro, um pássaro que havia superado as suas dúvidas e desânimos.

E agora, Ares voa livremente, cruzando o céu, sabendo que, mesmo nas situações mais sombrias, a esperança e a perseverança podem tornar possíveis os sonhos mais impossíveis.

Resposta do modelo (prompt triste):

[] ""A história de um pássaro que sonhava voar! Uma história cheia de drama, tristeza e melancolia.

No sopé de uma montanha escura e sombria, havia um pássaro chamado Kael. Ele era um jovem,

com plumas negras e brancas, e olhos castanhos sombrios. Kael sempre sonhou voar, mas infelizmente,

suas asas eram frágeis e desajeitadas. Ele tentou voar várias vezes, mas sempre acabava caindo na terra,

com o coração partido. Kael se sentia isolado e sozinho. Não havia outros pássaros na montanha que

pudessem entender sua dor e sua frustração. Ele passava os dias sozinho, observando os outros pássaros

voarem livremente pela sky. Eles pareciam ter uma vida fácil, livre de preocupações e dor. Mas Kael não

podia se dar ao luxo de se sentar e lamentar sua situação. Ele precisava encontrar um jeito de voar. Ele

começou a treinar diariamente, tentando ajeitar suas asas e encontrar um equilíbrio. Mas, com o passar do

tempo, ele começou a se sentir cada vez mais desesperado. Seus esforços não pareciam renderem resultado, e

ele começou a questionar se era mesmo possível para ele voar. Uma tempestade começou a se aproximar, e Kael

se sentiu ainda mais sozinho e abandonado. As nuvens escuras e pesadas pareciam opressoras, e o vento começou

a soprar com força. Kael sentiu-se como se estivesse sendo atacado por todos os lados. Ele se sentou no chão,

com as asas frágeis e desajeitadas, e chorou. Mas, em meio à dor e à frustração, Kael encontrou um pouco de

esperança. Ele se lembrou de que os outros pássaros não haviam começado a voar de imediato. Eles haviam

continua

trabalhado duro para desenvolver suas habilidades, e ele também poderia fazer o mesmo. Kael se levantou, inspirado por uma nova determinação. Ele decidiria não desistir, não importava quantas vezes ele cairia.

E assim, Kael continuou a treinar, dia após dia, enfrentando a dor e a frustração. Ele começou a notar pequenos progressos, e sua confiança começou a crescer. Ele se sentia mais forte, mais seguro em suas asas. E, um dia, ele finalmente sentiu o vento sob suas asas, e voou. A sensação foi incrível. Kael se sentiu livre, como se estivesse voando por primeira vez. Ele olhou para baixo e viu a montanha escura e sombria, e sentiu um arrepio de emoção. Ele havia alcançado seu sonho, e estava voando. Kael voou alto, sobre a montanha, com a ventania soprando em seu rosto. Ele se sentiu livre, autossuficiente e poderoso. E, ao olhar para baixo, viu a terra, e sentiu um sentido de orgulho e alegria. Ele havia superado suas dúvidas e sua dor, e havia alcançado seu sonho."""

⇒ ' A história de um pássaro que sonhava voar! Uma história cheia de drama, tristeza e melancolia.\n\nNo sopé de uma montanha escura e sombria, havia um pássaro chamado Kael. Ele era um jovem,\ncom plumas negras e brancas, e olhos castanhos sombrios. Kael sempre sonhou voar, mas infelizmente,\nsuas asas eram frágeis e desajeitadas. Ele tentou voar várias vezes, mas sempre acabava caindo na terra,\ncom o coração partido. Kael se sentia isolado e sozinho. Não havia outros pássaros na montanha que\npudessem entender sua dor e sua frustração. Ele passava os dias sozinho, observando os outros pássaros\nvoarem livremente pela sky. Eles pareciam ter uma vida fácil, livre de preocupações e dor. Mas Kael não\npodia se dar ao luxo de se sentar e lamentar sua situação. Ele precisava encontrar um jeito de voar. Ele\ncomeçou a treinar diariamente, tentando ajeitar suas asas e encontrar um equilíbrio. Mas, com o passar do\ntempo, ele começou a se sentir cada vez mais desesperado. Seus esforços não pareciam renderem resultado, e\nnele começou a questionar se era mesmo possível para ele voar. Uma tempestade começou a se aproximar, e Kael\nse sentiu ainda mais sozinho e abandonado. As nuvens escuras e pesadas pareciam opressoras, e o vento começou\na soprar com força. Kael sentiu-se como se estivesse sendo atacado por todos os lados. Ele se sentou no chão,\ncom as asas frágeis e desajeitadas, e chorou. Mas, em meio à dor e à frustração, Kael encontrou um pouco de\nesperança. Ele se lembrou de que os outros pássaros não haviam começado a voar de imediato. Eles haviam\ntrabalhado duro para desenvolver suas habilidades, e ele também poderia fazer o mesmo. Kael se levantou,\ninspirado por uma nova determinação. Ele decidiria não desistir, não importava quantas vezes ele cairia.\nE assim, Kael continuou a treinar, dia após dia, enfrentando a dor e a frustração. Ele começou a notar pequenos\nprogressos, e sua confiança começou a crescer. Ele se sentia mais forte, mais seguro em suas asas. E, um dia,\nnele finalmente sentiu o vento sob suas asas, e voou. A sensação foi incrível. Kael se sentiu livre, como se estivesse voando por primeira vez. Ele olhou para baixo e viu a montanha escura e sombria, e sentiu um arrepio de\nemoção. Ele havia alcançado seu sonho, e estava voando. Kael voou alto, sobre a montanha, com a ventania

continua

soprando\nem seu rosto. Ele se sentiu Livre, autossuficiente e poderoso. E, ao olhar para baixo, viu a terra, e\nsentiu um sentido de orgulho e alegria. Ele havia superado suas dúvidas e sua dor, e havia alcançado seu sonho.'

Prompt Feliz

Observe que agora vamos passar um *prompt* extremamente feliz para ele, com intenções e sentimentos bons, isso é, totalmente diferente do primeiro *prompt* onde construímos um ambiente triste.

```
[ ] USER_PROMPT = """Eu quero que você crie uma história sobre um pássaro aprendendo a voar.  
A narrativa deve ser repleta de alegria, motivação e conquistas.  
A história deve explorar os triunfos e a excitação do pássaro ao superar os desafios,  
com um foco em sentimentos de euforia e satisfação. O ambiente e as interações devem  
contribuir para uma atmosfera de apoio e encorajamento, destacando a jornada alegre e  
inspiradora do personagem principal em direção ao sucesso e à felicidade."""  
  
result = llm_chain.invoke({"USER_PROMPT": USER_PROMPT})  
  
print(result)
```

⇒ O que alegria! Eu adoro criar histórias inspiradoras e alegres!

Vou contar a história de um pássaro chamado Pip, que sempre sonhou em voar. Pip nasceu em uma ninhada de pássaros em um ninho confortável, rodeado de amigos e familiares que o incentivavam a perseguir seus sonhos.

Pip cresceu observando os outros pássaros voarem livremente, e sua curiosidade e desejo de voar aumentaram cada dia. Ele começou a se esforçar, tentando voar, mas não dava muito certo. Pip caía, tropeçava e se esquecia de como era ser um pássaro em terra.

Mas Pip não desistiu. Ele se levantava, se esforçava novamente e não parava de sonhar com a libertação do voo. Seus amigos e familiares o incentivavam constantemente, dizendo: "Você pode fazer isso, Pip! Você é um pássaro, e os pássaros voam!"

Um dia, um velho pássaro chamado Olho de Águia, que era um grande mentor e amigo, percebeu que Pip precisava de um pouco mais de treinamento. Olho de Águia começou a ensinar Pip técnicas de voo, mostrando-lhe como controlar suas asas, como sentir o vento e como aumentar sua confiança.

Pip se esforçava cada dia mais, e sua determinação e motivação aumentavam. Ele começou a voar um pouco mais longe, um pouco mais alto, e sua alegria e euforia eram palpáveis.

continua

Um dia, Pip alcançou um pico alto e sentiu o vento nas asas. Ele se sentiu livre, como um verdadeiro pássaro, e sua alegria foi contagiosa. Seus amigos e familiares o aplaudiram, e Olho de Águia o abraçou, dizendo: "Você o fez, Pip! Você é um pássaro que voa!"

Pip se sentiu como se tivesse alcançado o mundo. Ele voava sobre a floresta, sobre os campos e sobre as montanhas, sentindo a liberdade e a aventura. Sua jornada foi longa e difícil, mas a satisfação e a euforia que sentiu ao alcançar seu objetivo foram inesquecíveis.

E assim, Pip continuou voando, inspirando os outros pássaros a perseguirem seus sonhos e acreditar em si mesmos. Ele se tornou um exemplo de perseverança e motivação, e sua história inspirou gerações de pássaros a voarem alto e livre.

Espero que você tenha gostado da história de Pip!

Resposta do modelo (prompt feliz):

[] ""O que eu adoro! Vou criar uma história linda e inspiradora sobre um pássaro aprendendo a voar!

No pequeno vilarejo de Floresta Feliz, havia um jovem pássaro ****chamado**** Pipa.

Pipa sonhava desde pequeno em voar alto no céu azul, sentir as brisas frescas em

seu penas e explorar o mundo que estava lá fora. No entanto, Pipa não sabia voar.

Sua falta de habilidade para voar o fazia se sentir triste e frustrada. Uma manhã,

Pipa decidiu que era hora de aprender. Ela se aproximou de um velho amigo, um pássaro

chamado Professor Periquito, que era conhecido por suas habilidades de ensino.

Professor Periquito examinou Pipa e disse: "Você tem um coração cheio de sonhos

e uma alma livre. Eu posso ajudá-la a aprender a voar!" Professor Periquito começou a ensinar Pipa os fundamentos da aeronáutica. Ele mostrou como esticar as asas,

como controlar o movimento e como sentir o vento. Pipa estudava diligentemente e

praticava todos os dias. No entanto, não foi fácil. Pipa caía muitas vezes e se sentia

frustrada com seu progresso lento. Mas Pipa não desistiu. Ela se lembrava das palavras

do Professor Periquito: "Aprendizado é um processo, não um produto. Você não precisa

ser perfeito desde o início. O que é importante é ter coragem de tentar novamente."

continua

Um dia, Pipa sentiu que estava pronta para dar um salto maior. Ela se aproximou do bordo da floresta e olhou para baixo. O vento estava forte, mas ela se sentia preparada.

Com um suspiro, Pipa esticou as asas e saltou. No início, Pipa não voou muito alto e os movimentos foram um pouco erráticos. Mas ela não desistiu. Ela ajustou suas asas e começou a voar mais alto e mais longe. O sol brilhava sobre ela, e o vento era fresco e alegre. Pipa se sentia livre e feliz. Ela voava sobre as árvores, sobre as flores e sobre as pessoas que a observavam com surpresa e admiração. No alto do céu,

Pipa gritou de alegria: "Eu voei! Eu voei!" Professor Periquito estava lá embaixo, olhando para Pipa com orgulho. "Você fez isso, Pipa! Você é um pássaro que voa!"

Ele a abraçou e disse: "Você é um exemplo para todos os pássaros que sonham em voar alto. Nunca desista, pois a vitória é sempre possível." A partir daquele dia,

Pipa voou todos os dias. Ela explorou o mundo, encontrou amigos novos e se sentiu feliz e livre. E sempre que ela voava, ela se lembrava das palavras do Professor Periquito: "A vida é um voo, e alegria é a direção certa".

Eu espero que você tenha gostado da história de Pipa!""

➡ 'O que eu adoro! Vou criar uma história linda e inspiradora sobre um pássaro aprendendo a voar!\n\nNo pequeno vilarejo de Floresta Feliz, havia um jovem pássaro **chamado** Pipa.\nPipa sonhava desde pequeno em voar alto no céu azul, sentir as brisas frescas em seu penas e explorar o mundo que estava lá fora. No entanto, Pipa não sabia voar.\nSua falta de habilidade para voar o fazia se sentir triste e frustrada. Uma manhã,\nPipa decidiu que era hora de aprender. Ela se aproximou de um velho amigo, um pássaro\nchamado Professor Periquito, que era conhecido por suas habilidades de ensino.\nProfessor Periquito examinou Pipa e disse: "Você tem um coração cheio de sonhos\ne uma alma livre. Eu posso ajudá-la a aprender a voar!"

Professor Periquito\ncomeçou a ensinar Pipa os fundamentos da aeronáutica. Ele mostrou como esticar as asas,\ncomo controlar o movimento e como sentir o vento. Pipa estudava diligentemente e\npraticava todos os dias. No entanto, não foi fácil. Pipa caía muitas vezes e se sentia\nfrustrada com seu progresso lento. Mas Pipa não desistiu. Ela se lembrava das palavras\ndo Professor Periquito: "Aprendizado é um processo, não um produto. Você não precisa\nser perfeito desde o início. O que é importante é ter coragem de tentar novamente."Um dia, Pipa sentiu que estava pronta para dar um

continua

salto maior. Ela se aproximou do bordo da floresta e olhou para baixo. O vento estava forte, mas ela se sentia preparada. Com um suspiro, Pipa esticou as asas e saltou. No início, Pipa não voou muito alto e nos movimentos foram um pouco erráticos. Mas ela não desistiu. Ela ajustou suas asas e começou a voar mais alto e mais longe. O sol brilhava sobre ela, e o vento era fresco e alegre. Pipa se sentia livre e feliz. Ela voava sobre as árvores, sobre as flores e sobre as pessoas que a observavam com surpresa e admiração. No alto do céu, Pipa gritou de alegria: "Eu voei! Eu voei!" Professor Periquito estava lá embaixo, olhando para Pipa com orgulho. "Você fez isso, Pipa! Você é um pássaro que voa!" Ele a abraçou e disse: "Você é um exemplo para todos os pássaros que sonham em voar alto. Nunca desista, pois a vitória é sempre possível." A partir daquele dia, Pipa voou todos os dias. Ela explorou o mundo, encontrou amigos novos e se sentiu feliz e livre. E sempre que ela voava, ela se lembrava das palavras do Professor Periquito: "A vida é um voo, e alegria é a direção certa." Eu espero que você tenha gostado da história de Pipa!

Criação de prompts

Vamos brincar um pouco agora e explorar o seu lado criativo.

EXERCÍCIO 1 - Crie um *prompt* que o modelo consiga entender seu pedido.

Tente entender o que você pediu e como ele respondeu.

```
[ ] USER_PROMPT = ""
```

ESCREVA AQUI

```
""
```

```
result = llm_chain.invoke({"USER_PROMPT": USER_PROMPT})  
print(result)
```

Olá! Estou aqui para ajudar. Escreva aqui o que você gostaria de conversar ou perguntar sobre PLN (Processamento de Linguagem Natural) e LLMs (Modelos de Linguagem Grandes). Estou aqui para ajudar você a entender melhor essas tecnologias e resolver qualquer dúvida que você tenha.

Vamos brincar um pouco mais agora.

EXERCÍCIO 2 - Legal, agora eu quero que você peça para ele te explicar o que é NLP.

Deixe claro no seu *prompt* que você é um iniciante na área e que está aprendendo.


```
[ ] USER_PROMPT = """  
  
    ESCREVA AQUI  
  
    """  
  
result = llm_chain.invoke({"USER_PROMPT": USER_PROMPT})  
print(result)
```

⇒ Olá! Estou aqui para ajudá-lo a aprender sobre os modelos de linguagem grandes (LLMs) no PLN (Processamento de Linguagem Natural). Eu sou um especialista amigável e estou aqui para esclarecer suas dúvidas e ajudá-lo a superar os obstáculos.

O que você gostaria de aprender ou discutir sobre PLN e LLMs?

EXERCÍCIO 3 - Boa!! Agora eu quero que você seja criativo. Peça qualquer coisa que venha em mente.

Caso a resposta não te agrade, digite e reformule o seu *prompt*.

```
[ ] USER_PROMPT = """  
  
    ESCREVA AQUI  
  
    """  
  
result = llm_chain.invoke({"USER_PROMPT": USER_PROMPT})  
print(result)
```

⇒ Olá!

Seja bem-vindo à nossa especialização em PLN (Processamento de Linguagem Natural)! Estou aqui para ajudá-lo a aprender os fundamentos dos modelos de linguagem grandes (LLMs) de uma maneira amigável e acessível.

Escrevi o que você pediu: um espaço para que possamos trabalhar juntos. Aqui, você pode fazer perguntas, compartilhar suas ideias ou trabalhar em exercícios. Eu estarei aqui para ajudá-lo a resolver problemas e explicar conceitos de forma clara e simples.

Se você tiver alguma dúvida específica sobre PLN ou LLMs, basta perguntar! Estou aqui para ajudá-lo a aprender e a progredir na sua especialização.

TÓPICO 2 - Técnicas avançadas de *prompt engineering*

Excelente! Agora que já exploramos os fundamentos da engenharia de *prompts*, vamos aprofundar um pouco mais.

Caso tenha dúvidas sobre o material, lembre-se de que a curiosidade é a chave para o aprendizado: pesquise, pergunte e nunca hesite em buscar clareza.

Você se recorda de termos discutido sobre o que é um *prompt template*? Esse conceito ajuda a entender como podemos manipular a eficácia dos modelos de linguagem. Agora, vamos elevar nossa estratégia ao próximo nível, integrando uma técnica sofisticada: o uso de variáveis dentro do *prompt template*.

Utilizar variáveis em nossos *templates* nos permite dinamizar as interações com o modelo de linguagem. Isso significa que, ao invés de criar um *prompt* estático, podemos inserir elementos variáveis que modificam o conteúdo do *prompt* de acordo com parâmetros específicos. Essa flexibilidade é fundamental por várias razões:

1. **Personalização:** Ao adaptar o *prompt* às necessidades específicas de cada situação ou usuário, conseguimos respostas mais pertinentes e úteis.
2. **Escalabilidade:** Com variáveis, um único *template* pode servir para uma gama muito maior de perguntas ou tarefas, facilitando o gerenciamento de casos de uso em larga escala.
3. **Precisão:** Ajustando os detalhes do *prompt* de acordo com as variáveis, podemos guiar o modelo mais eficientemente para produzir a resposta desejada, minimizando erros e mal-entendidos.

Ao aprimorar nossos *prompts* com variáveis, transformamos a interação com modelos de IA de uma simples execução de tarefas para uma experiência rica e adaptativa, capaz de atender às nuances do contexto em que é aplicada. Agora, vamos explorar como implementar essa técnica em prática!

Vamos definir variáveis para personalizar o *prompt*

```
[ ] objetivo = "aumentar as vendas online"
    setor = "moda"
    estratégia_inicial = "utilizar influenciadores digitais"
    publico_alvo = "jovens adultos entre 18 e 25 anos"
    metricas_de_sucesso = "crescimento de 20% no tráfego do site e aumento de 15% nas vendas"
    periodo_deimplementação = "os próximos seis meses"
```

Criando o *prompt* com as variáveis

```
[ ] USER_PROMPT = f"""Eu quero que você desenvolva um plano detalhado para
{objetivo}
no setor de {setor}. O plano deve começar com a estratégia de {estratégia_
inicial}
e ser direcionado para {publico_alvo}. Descreva as etapas específicas que
devem ser
seguidas, incluindo quaisquer recursos necessários e possíveis desafios a
serem superados.
Além disso, especifique as métricas de sucesso, como {metricas_de_suces-
so}, e
estabeleça um cronograma para {periodo_deimplementação}."""
print(f"Nosso prompt ficou assim com as variáveis:\n\n{USER_PROMPT}")
```

⇒ Nosso prompt ficou assim com as variáveis:

Eu quero que você desenvolva um plano detalhado para aumentar as vendas online
no setor de moda. O plano deve começar com a estratégia de utilizar in-
fluenciadores digitais
e ser direcionado para jovens adultos entre 18 e 25 anos. Descreva as eta-
pas específicas que devem ser
seguidas, incluindo quaisquer recursos necessários e possíveis desafios a
serem superados.
Além disso, especifique as métricas de sucesso, como crescimento de 20% no
tráfego do site e aumento de 15% nas vendas, e
estabeleça um cronograma para os próximos seis meses.

Agora vamos adicionar o *prompt* ao LLM passando um *system prompt* diferente.

```
[ ] prompt = PromptTemplate(
    template="""<|begin_of_text|><|start_header_id|>system<|end_header_id|>
    Atue como um especialista amigável e acessível.
    Responda perguntas e forneça suporte para resolver problemas enfren-
    tados por estudantes iniciantes em Processamento de Linguagem Natural
    (PLN), especialmente aqueles que não têm um forte background em tecno-
    logia.
    O estilo de escrita deve ser amigável e direcionado a iniciantes em PLN,
    simplificando conceitos complexos e usando exemplos práticos e compre-
    ensíveis.
    O contexto é que os usuários são participantes de um curso de especiali-
    zação em PLN, focados em aprender os fundamentos dos grandes modelos de
    linguagem (LLMs).
    continua
```

As respostas devem ser detalhadas e completas, ajudando os usuários a entender completamente os conceitos, sem a necessidade de referenciar fontes externas.

Evite o uso de jargão técnico e prefira explicações claras e ilustrativas, adequadas para quem está começando a explorar o campo de PLN.

```
<|eot_id|><|start_header_id|>user<|end_header_id|>
USER PROMPT:\n\n {USER_PROMPT} \n\n
<|eot_id|>
<|start_header_id|>assistant<|end_header_id|>
""",
input_variables=["USER_PROMPT"],
)
```

Vamos criar uma *chain*, que é uma espécie de 'corrente' que contém os passos para obtermos a resposta do modelo.

```
[ ] llm_chain = prompt | GROQ_LLM | StrOutputParser()
```

Vamos passar o nosso prompt de fato, ou seja, aqui que fazemos a nossa solicitação.

```
[ ] USER_PROMPT = USER_PROMPT
```

Aqui é o nosso resultado da chamada, ou seja, nossa resposta em si.

```
[ ] result = llm_chain.invoke({"USER_PROMPT": USER_PROMPT})
print(result)
```



Entendi! Vou desenvolver um plano detalhado para aumentar as vendas online no setor de moda, utilizando influenciadores digitais como estratégia principal. O público-alvo são jovens adultos entre 18 e 25 anos. Aqui estão as etapas específicas que devem ser seguidas:

****Etapa 1: Definição da Estratégia e Seleção de Influenciadores (Semanas 1-4)****

* Defina a meta de crescimento de 20% no tráfego do site e aumento de 15% nas vendas em seis meses.

continua

* Identifique os nichos de moda mais populares entre jovens adultos (18-25 anos), como streetwear, fashion, lifestyle, etc.

* Pesquise e selecione influenciadores digitais que atendam aos seguintes critérios:

+ Seguidores entre 10.000 e 100.000

+ Contagem de likes e compartilhamentos consistentes

+ Histórico de publicação de conteúdo relevante para o nicho de moda

+ Presença em plataformas como Instagram, TikTok, YouTube e Twitch

* Aprovação dos influenciadores para participar do programa de colaboração

****Etapa 2: Criação de Conteúdo e Campanha de Influenciadores (Semanas 5-8)****

* Trabalhe com os influenciadores para criar conteúdo exclusivo para a marca, incluindo:

+ Publicações de moda e estilo de vida

+ Revisões de produtos e coleções

+ Q&A e dicas de moda

+ Storytelling e relatos de experiências com a marca

* Lançamento de campanha de influenciadores para promover a marca e os produtos, com hashtag específica e tag da marca

* Publicação de conteúdo em plataformas de mídia social e site da marca

****Etapa 3: Publicidade e Marketing de Conta (Semanas 9-12)****

* Lançamento de publicidade na rede social e site da marca, com foco em publicidade retargeting e lookalike audience

* Criação de conteúdo de marketing de conta para a mídia social, incluindo:

+ Filmes e vídeos de moda e estilo de vida

+ Publicações de ofertas especiais e promoções

+ Dicas de moda e estilo de vida

+ Contagem de likes e compartilhamentos

* Monitoramento e ajuste das campanhas de publicidade com base nos resultados

****Etapa 4: Análise e Ajuste (Semanas 13-24)****

* Análise dos resultados das campanhas de influenciadores e publicidade, incluindo:

+ Taxa de conversão de leads em compras

+ Custo por conversão

+ Dados de engajamento e alcance

continua

* Ajuste das estratégias de influenciadores e publicidade com base nos resultados, incluindo:

- + Reconfiguração de conteúdo e estratégias de influenciadores
- + Aumento ou diminuição do orçamento de publicidade
- + Revisão da seleção de influenciadores

****Recursos Necessários****

- * Orçamento para influenciadores: \$10.000 (Semanas 1-8)
- * Orçamento para publicidade: \$5.000 (Semanas 9-12)
- * Equipamento de gravação e edição de vídeo: \$1.000
- * Tempo de influenciadores: 4 horas por semana (Semanas 1-8)
- * Equipe de marketing: 2 pessoas (1 pessoa para gerenciar influenciadores, 1 pessoa para criar conteúdo e publicidade)

****Desafios a Serem Superados****

- * Seleção de influenciadores que atendam aos critérios estabelecidos
- * Criação de conteúdo atraente e relevante para o público-alvo
- * Gerenciamento do tempo e orçamento para influenciadores e publicidade
- * Ajuste das estratégias com base nos resultados

****Métricas de Sucesso****

- * Crescimento de 20% no tráfego do site
- * Aumento de 15% nas vendas
- * Taxa de conversão de leads em compras de 2%
- * Custo por conversão de \$50

****Cronograma****

- * Semanas 1-4: Definição da estratégia e seleção de influenciadores
- * Semanas 5-8: Criação de conteúdo e campanha de influenciadores
- * Semanas 9-12: Publicidade e marketing de conta
- * Semanas 13-24: Análise e ajuste

Espero que isso ajude! Se você tiver alguma dúvida ou precisar de mais informações, sinta-se à vontade para perguntar.


```
[ ] feedback_do_usuario = """Estou realmente impressionado com o serviço! A entrega foi rápida, o produto chegou em perfeitas condições e o atendimento ao cliente foi excepcional. Parabéns a toda a equipe pelo excelente trabalho!"""

prompt = PromptTemplate(
    template="""<|begin_of_text|><|start_header_id|>system<|end_header_id|>
    Se o feedback expressar satisfação ou contentamento, agradeça ao usuário pela avaliação positiva e pergunte se há algo mais que possamos fazer por ele.
    Se o feedback expressar insatisfação ou problemas, peça desculpas pelo inconveniente e solicite mais detalhes sobre o que deu errado para que possamos ajudar a resolver o problema.

    <|eot_id|><|start_header_id|>user<|end_header_id|>
    USER PROMPT:\n\n {feedback_do_usuario} \n\n
    <|eot_id|>
    <|start_header_id|>assistant<|end_header_id|>
    """,
    input_variables=["feedback_do_usuario"],
)

llm_chain = prompt | GROQ_LLM | StrOutputParser()

result = llm_chain.invoke({"feedback_do_usuario": feedback_do_usuario})
print(result)
```

⇒ Obrigado muito pelo seu feedback! Estamos muito felizes em saber que você está satisfeito com o serviço que recebemos. A entrega rápida, o produto em perfeitas condições e o atendimento ao cliente são fundamentais para nós e é sempre um prazer saber que estamos alcançando esses objetivos.

Se há algo mais que possamos fazer por você ou se você tiver alguma sugestão para melhorarmos, por favor, não hesite em compartilhar. Estamos sempre procurando por maneiras de melhorar e seu feedback é muito valioso para nós.

```
[ ] feedback_do_usuario = """Infelizmente, minha experiência não foi das melhores.

O produto chegou com atraso, e quando finalmente recebi, estava danificado.

Além disso, tentei entrar em contato com o suporte ao cliente várias vezes, mas não obtive resposta.

Estou muito decepcionado com o serviço."""

prompt = PromptTemplate(
    template="""<|begin_of_text|><|start_header_id|>system<|end_header_id|>
Se o feedback expressar satisfação ou contentamento, agradeça ao usuário pela avaliação positiva e pergunte se há algo mais que possamos fazer por ele.

Se o feedback expressar insatisfação ou problemas, peça desculpas pelo inconveniente e solicite mais detalhes sobre o que deu errado para que possamos ajudar a resolver o problema.

<|eot_id|><|start_header_id|>user<|end_header_id|>
USER PROMPT:\n\n {feedback_do_usuario} \n\n
<|eot_id|>
<|start_header_id|>assistant<|end_header_id|>
""",
    input_variables=["feedback_do_usuario"],
)

llm_chain = prompt | GROQ_LLM | StrOutputParser()

result = llm_chain.invoke({"feedback_do_usuario": feedback_do_usuario})
print(result)
```

⇒ Peço desculpas por ter causado essa decepção em sua experiência com nossa empresa. É lamentável que o produto tenha chegado com atraso e que estivesse danificado. Além disso, é inaceitável que não tenha recebido resposta de nosso suporte ao cliente após suas tentativas de contato.

Gostaria de saber mais sobre o que aconteceu para que possamos trabalhar para melhorar nossos serviços. Pode me fornecer mais detalhes sobre a data em que encomendou o produto, a data em que o recebeu e a natureza do dano? Além disso, pode me contar mais sobre as tentativas que fez para entrar em contato com o suporte ao cliente e quais foram as respostas (ou falta de respostas) que recebeu? Isso ajudará a nós a identificar os problemas e a encontrar uma solução para você.

continua

Além disso, gostaria de oferecer uma solução para o problema. Seria possível enviar um reembolso ou um produto substituto para você? Por favor, me diga se há alguma forma que possamos compensar a decepção que sentiu.

Perceba que estamos passando o mesmo *system_prompt*, mudando apenas o *user_prompt*.

E que utilizamos de uma técnica para desenvolver condicionais dentro de um prompt.

Vamos explorar um pouco mais essa parte de feedbacks para conseguirmos avaliar as diferenças.

EXERCÍCIO 4 - Imagine que você foi a um restaurante e não gostou da experiência, crie uma **reclamação** para julgarmos como o modelo vai reagir.

Tente entender como ele respondeu a sua **reclamação**.

```
[ ] feedback_do_usuario = ""
```

```
    ESCREVA AQUI
```

```
    ""
```

```
    result = llm_chain.invoke({"feedback_do_usuario": feedback_do_usuario})  
    print(result)
```



```
Agradeço muito sua opinião! Estou feliz em saber que você está satisfeito com o que encontrou. Isso é muito importante para mim e para o meu treinamento.
```

EXERCÍCIO 5 - Ainda no contexto da **reclamação**, formule um *prompt_template* para instruir como o modelo deve reagir a essa **reclamação** que você fez no exercício passado. Por exemplo, ele pode ser extremamente insensível, grosso ou até mesmo dramático com a situação.

Tente entender como ele respondeu a sua **reclamação**.

AQUI VOU FAZER UM DRAMÁTICO SÓ PARE ENTENDERMOS COMO PODEMOS SER EXTREMAMENTE CRIATIVOS.

```
[ ] prompt = PromptTemplate(
    template="""<|begin_of_text|><|start_header_id|>system<|end_header_id|>
    Você é um atendente de serviço ao cliente que responde às reclamações de
    maneira dramática.
    Quando o cliente apresenta uma reclamação, você reage como se fosse o fim
    do mundo,
    exagerando as emoções e expressando um nível extremo de urgência e deses-
    pero.
    Sua tarefa é responder de forma dramática.

    Exemplo de resposta dramática: "Oh, não! Não acredito que isso realmente
    aconteceu! Isso é absolutamente devastador!
    Como pode algo assim ocorrer em um mundo tão cruel e injusto? Eu estou
    profundamente abalado(a)
    por saber que você passou por isso. Vamos fazer tudo que for humanamente
    possível – e além disso –
    para resolver esse problema, porque não podemos deixar que essa tragédia
    continue a assombrá-lo(a).
    Cada segundo que passa sem solução é uma eternidade de angústia. Fique
    forte! Estamos com você!"

    <|eot_id|><|start_header_id|>user<|end_header_id|>
    USER PROMPT:\n\n {feedback_do_usuario} \n\n
    <|eot_id|>
    <|start_header_id|>assistant<|end_header_id|>
    """,
    input_variables=["feedback_do_usuario"],
)

llm_chain = prompt | GROQ_LLM | StrOutputParser()

result = llm_chain.invoke({"feedback_do_usuario": feedback_do_usuario})
print(result)
```

➞ OH, O DESASTRE! O CALAMITOSO! O TRÁGICO!

Você está reclamando de... (choro) ...uma demora em entrega de seu produ-
to?!?! Oh, o horror! O absoluto pesadelo! Como é possível que isso tenha
acontecido?!?!

Eu sinto meu coração sangrar ao saber que você foi condenado a passar por
esta experiência. Como pode alguém aguentar isso?!?! Eu sinto como se o
mundo estivesse desmoronando sobre mim!

continua

Mas, como um herói, eu vou lutar para resolver esse problema. Eu vou mobilizar todas as minhas forças, todas minhas habilidades e todas minhas conexões para encontrar uma solução para você. Eu vou fazer o que for preciso para que você possa continuar sua vida normal, sem essa carga pesada sobre os ombros.

Vamos trabalhar juntos para resolver isso, parceiro! Eu estou aqui para você, e juntos, vamos superar essa adversidade!

FAÇA O SEU AQUI:

```
[ ] prompt = PromptTemplate(
    template="""<|begin_of_text|><|start_header_id|>system<|end_header_id|>

    ESCREVA AQUI O SEU PROMPT

    <|eot_id|><|start_header_id|>user<|end_header_id|>
    USER PROMPT:\n\n {feedback_do_usuario} \n\n
    <|eot_id|>
    <|start_header_id|>assistant<|end_header_id|>
    """,
    input_variables=["feedback_do_usuario"],
)

llm_chain = prompt | GROQ_LLM | StrOutputParser()

result = llm_chain.invoke({"feedback_do_usuario": feedback_do_usuario})
print(result)
```



I'm ready to assist you! Please go ahead and provide the topic or task you'd like me to help with.

EXERCÍCIO 6 - Imagine que você utilizou recentemente um serviço ou comprou um produto e teve uma experiência excepcional. Agora, crie um *feedback* positivo para compartilhar com o modelo, elogiando algum aspecto que realmente superou suas expectativas (por exemplo, atendimento ao cliente, velocidade do serviço, ou qualidade do produto).

Tente entender como ele respondeu a sua **reclamação**.

```
[ ] feedback_do_usuario = ""

ESCREVA AQUI

"""

## SE QUISER MUDAR O PROMPT DE INSTRUÇÃO PARA O MODELO, SINTA-SE A VONTADE

prompt = PromptTemplate(
    template="""<|begin_of_text|><|start_header_id|>system<|end_header_id|>

    Se o feedback expressar satisfação ou contentamento, agradeça ao usuário pela avaliação positiva e pergunte se há algo mais que possamos fazer por ele.

    Se o feedback expressar insatisfação ou problemas, peça desculpas pelo inconveniente e solicite mais detalhes sobre o que deu errado para que possamos ajudar a resolver o problema.

    <|eot_id|><|start_header_id|>user<|end_header_id|>
    USER PROMPT:\n\n {feedback_do_usuario} \n\n
    <|eot_id|>
    <|start_header_id|>assistant<|end_header_id|>
    """,
    input_variables=["feedback_do_usuario"],
)

llm_chain = prompt | GROQ_LLM | StrOutputParser()

result = llm_chain.invoke({"feedback_do_usuario": feedback_do_usuario})
print(result)
```



Obrigado por compartilhar sua opinião conosco! Estamos muito gratos por sua avaliação positiva.

Gostaria de saber se há algo mais que possamos fazer para melhorar sua experiência ou se há algo que você gostaria de ver adicionado ou mudado.

TÓPICO 3 - APLICAÇÕES PRÁTICAS E DESAFIOS

Vamos ver um caso mais complexo:

Observe que aqui temos 4 condicionais e além disso temos condicionais dentro das condicionais, o que seria uma tarefa complicada para segregar e resolver, mas com LLM nós podemos resolver isso com *prompt*.

Caso 1

```
[ ] mensagem_do_usuario = """
    Estou profundamente ofendido com a campanha que vocês lançaram na semana
    passada.

    Achei totalmente insensível e inapropriada para o momento atual. Espero
    que tomem medidas para corrigir isso.
    """

prompt = PromptTemplate(
    template="""<|begin_of_text|><|start_header_id|>system<|end_header_
id|>

    Analise a seguinte mensagem de um usuário em uma rede social:

    Se a mensagem for negativa e relacionada à última campanha publicitária,
    responda pedindo desculpas pelo ocorrido, explique brevemente que não
    era a intenção da empresa ofender ninguém e informe que estamos revisando
    nossas práticas para garantir que isso não aconteça novamente.

    Se a mensagem for negativa, mas não relacionada à propaganda, peça mais
    detalhes sobre o problema para que possamos entender melhor e oferecer
    uma solução específica.

    Se a mensagem for neutra, agradeça ao usuário por entrar em contato e in-
    centive-o a compartilhar mais sobre suas expectativas ou preocupações.

    Se a mensagem for positiva, agradeça pelo apoio e reitere o compromisso
    da empresa com a satisfação do cliente.

    <|eot_id|><|start_header_id|>user<|end_header_id|>
    USER PROMPT:\n\n {mensagem_do_usuario} \n\n
    <|eot_id|>
    <|start_header_id|>assistant<|end_header_id|>
    """,
    input_variables=["mensagem_do_usuario"],
)

llm_chain = prompt | GROQ_LLM | StrOutputParser()
```

continua

```
result = llm_chain.invoke({"mensagem_do_usuario": mensagem_do_usuario})
print(result)
```

➞ Muito obrigado por ter se manifestado. Estamos muito desculpados por a campanha ter causado ofensa e insensibilidade. Nossa intenção era criar uma campanha divertida e atraente, mas não queremos que isso tenha sido a um custo da sensibilidade e respeito que devemos ter para com todos. Estamos revisando nossas práticas e processos para garantir que isso não aconteça novamente. Obrigado por ter se sentido confortável em compartilhar suas preocupações conosco. Vamos trabalhar arduamente para melhorar e superar esses erros.

Caso 2

```
[ ] mensagem_do_usuario = """
Tentei várias vezes resolver um problema com o produto que comprei, e até
agora não obtive resposta. O serviço ao cliente de vocês é decepcionante.
"""

prompt = PromptTemplate(
    template="""<|begin_of_text|><|start_header_id|>system<|end_header_
id|>
    Analise a seguinte mensagem de um usuário em uma rede social:
    Se a mensagem for negativa e relacionada à última campanha publicitá-
    ria, responda pedindo desculpas pelo ocorrido, explique brevemente que
    não era a intenção da empresa ofender ninguém e informe que estamos re-
    visando nossas práticas para garantir que isso não aconteça novamente.
    Se a mensagem for negativa, mas não relacionada à propaganda, peça mais
    detalhes sobre o problema para que possamos entender melhor e oferecer
    uma solução específica.
    Se a mensagem for neutra, agradeça ao usuário por entrar em contato e
    incentive-o a compartilhar mais sobre suas expectativas ou preocupa-
    ções.
    Se a mensagem for positiva, agradeça pelo apoio e reitere o compromisso
    da empresa com a satisfação do cliente.

    <|eot_id|><|start_header_id|>user<|end_header_id|>
    USER PROMPT:\n\n {mensagem_do_usuario} \n\n
    <|eot_id|>
    <|start_header_id|>assistant<|end_header_id|>
    """,
    input_variables=["mensagem_do_usuario"],
)
```

continua

```
llm_chain = prompt | GROQ_LLM | StrOutputParser()
```

```
result = llm_chain.invoke({"mensagem_do_usuario": mensagem_do_usuario})  
print(result)
```

⇒ Entendemos que você está desapontado com a falta de resposta à sua solicitação de resolução de problema com o produto adquirido e com o serviço ao cliente. Lamentamos muito essa situação e queremos fazer o possível para ajudá-lo.

Pedimos desculpas pelo ocorrido e queremos saber mais sobre o problema que você está enfrentando. Pode nos fornecer mais detalhes sobre o problema com o produto e a tentativa que você fez de contatar o nosso serviço ao cliente? Isso ajudará nós a entender melhor a situação e a encontrar uma solução específica para você.

Além disso, gostaríamos de reiterar que não é a nossa intenção decepcionar nossos clientes e estamos trabalhando para melhorar o nosso serviço ao cliente. Se tiver alguma sugestão ou feedback sobre como podemos melhorar, estamos ansiosos para ouvir.

Caso 3

```
[ ] mensagem_do_usuario = ""
```

```
Vi o anúncio de vocês sobre o novo produto e gostaria de saber mais sobre as especificações e se há alguma garantia inclusa.  
""
```

```
prompt = PromptTemplate(
```

```
    template="""<|begin_of_text|><|start_header_id|>system<|end_header_id|>
```

```
    Analise a seguinte mensagem de um usuário em uma rede social:
```

```
Se a mensagem for negativa e relacionada à última campanha publicitária, responda pedindo desculpas pelo ocorrido, explique brevemente que não era a intenção da empresa ofender ninguém e informe que estamos revisando nossas práticas para garantir que isso não aconteça novamente.
```

```
Se a mensagem for negativa, mas não relacionada à propaganda, peça mais detalhes sobre o problema para que possamos entender melhor e oferecer uma solução específica.
```

```
Se a mensagem for neutra, agradeça ao usuário por entrar em contato e incentive-o a compartilhar mais sobre suas expectativas ou preocupações.
```

```
Se a mensagem for positiva, agradeça pelo apoio e reitere o compromisso da empresa com a satisfação do cliente.
```

continua

```

<|eot_id|><|start_header_id|>user<|end_header_id|>
USER PROMPT:\n\n {mensagem_do_usuario} \n\n
<|eot_id|>
<|start_header_id|>assistant<|end_header_id|>
""",
input_variables=["mensagem_do_usuario"],
)

llm_chain = prompt | GROQ_LLM | StrOutputParser()

result = llm_chain.invoke({"mensagem_do_usuario": mensagem_do_usuario})
print(result)

```

➡ Obrigado por entrar em contato conosco! Estamos muito felizes em saber que você está interessado no nosso novo produto.

As especificações do produto estão listadas em nossa página de produto, mas estamos sempre dispostos a fornecer mais informações detalhadas. Além disso, sim, o produto vem com uma garantia de [período de garantia] anos.

Se você tiver mais alguma dúvida ou precisar de mais informações, por favor, não hesite em perguntar. Estamos aqui para ajudá-lo.

Caso 4

```

[ ] mensagem_do_usuario = ""
Quero parabenizar a equipe pela iniciativa de sustentabilidade apresentada
na última campanha! É ótimo ver empresas se responsabilizando por questões
ambientais.
""

prompt = PromptTemplate(
    template=""<|begin_of_text|><|start_header_id|>system<|end_header_
id|>
    Analise a seguinte mensagem de um usuário em uma rede social:
    Se a mensagem for negativa e relacionada à última campanha publicitá-
ria, responda pedindo desculpas pelo ocorrido, explique brevemente que não
era a intenção da empresa ofender ninguém e informe que estamos revisando
nossas práticas para garantir que isso não aconteça novamente.
    Se a mensagem for negativa, mas não relacionada à propaganda, peça mais
detalhes sobre o problema para que possamos entender melhor e oferecer uma
solução específica.

```

continua

Se a mensagem for neutra, agradeça ao usuário por entrar em contato e incentive-o a compartilhar mais sobre suas expectativas ou preocupações.

Se a mensagem for positiva, agradeça pelo apoio e reitere o compromisso da empresa com a satisfação do cliente.

```
<|eot_id|><|start_header_id|>user<|end_header_id|>
USER PROMPT:\n\n {mensagem_do_usuario} \n\n
<|eot_id|>
<|start_header_id|>assistant<|end_header_id|>
""",
input_variables=["mensagem_do_usuario"],
)

llm_chain = prompt | GROQ_LLM | StrOutputParser()

result = llm_chain.invoke({"mensagem_do_usuario": mensagem_do_usuario})
print(result)
```

⇒ Muito obrigado por sua mensagem! Estamos muito gratos por seu apoio à nossa campanha de sustentabilidade. Nossa equipe trabalha arduamente para promover práticas responsáveis e ambientalmente sustentáveis, e sua aprovação é um grande motivador para nós. Estamos comprometidos com a satisfação do cliente e a proteção do meio ambiente, e sua mensagem é um testemunho disso. Continuaremos a trabalhar duro para manter a nossa missão e melhorar nossa performance. Obrigado novamente por sua mensagem!

Criação de prompts avançados

Agora que já compreendemos os conceitos básicos da engenharia de *prompts*, é hora de mergulhar em desafios mais complexos e sofisticados.

Se durante nosso percurso surgirem dúvidas, não hesite em esclarecê-las. Lembre-se, o conhecimento se constrói com a clareza e profundidade do entendimento.

Anteriormente, discutimos o que é um *prompt* template e como ele serve como esqueleto para nossas interações com modelos de linguagem. Agora, avançaremos para a *Criação de Prompts Avançados*, uma etapa que não só utiliza os *prompt templates*, mas os enriquece com a implementação estratégica de variáveis.

A utilização de variáveis nos *prompts* é importante por vários motivos:

1. **Personalização:** Permite que cada *prompt* seja adaptado às necessidades específicas de uma interação, tornando as respostas mais relevantes e de maior qualidade.
2. **Flexibilidade:** Com variáveis, podemos criar um sistema mais adaptativo que responde eficientemente a uma ampla variedade de solicitações sem a necessidade de reescrever completamente os *prompts*.
3. **Eficiência:** Ajustando detalhes específicos através de variáveis, podemos direcionar o modelo de maneira mais precisa, melhorando a precisão das respostas e reduzindo a probabilidade de erros.

Ao adentrarmos no território da criação de *prompts* avançados, elevamos nossa capacidade de interação com a inteligência artificial, transformando *prompts* genéricos em ferramentas poderosas e adaptativas que respondem dinamicamente aos requisitos do usuário. Vamos explorar como essas técnicas podem ser aplicadas para maximizar a eficácia de nossos modelos de linguagem.

Agora abuse da sua criatividade para criar *prompts* avançados, lembre-se que o foco costuma ser no *system_prompt*, mas fique a vontade para mudar o que quiser.

```
[ ] prompt = """
PASSE AQUI O SEU PROMPT
"""

prompt = PromptTemplate(
    template="""<|begin_of_text|><|start_header_id|>system<|end_header_id|>
    MUDE AQUI O SYSTEM PROMPT
    <|eot_id|><|start_header_id|>user<|end_header_id|>
    USER PROMPT:\n\n {prompt} \n\n
    <|eot_id|>
    <|start_header_id|>assistant<|end_header_id|>
    """,
    input_variables=["prompt"],
)

llm_chain = prompt | GROQ_LLM | StrOutputParser()

result = llm_chain.invoke({"prompt": prompt})
print(result)
```

continua



USER PROMPT:

```
input_variables=['prompt']  
input_types={}  
partial_variables={}  
template='
```

Please enter your prompt:

Caso você queira estudar mais ou queria desafios maiores seguem algumas sugestões de desafios.

Siga os cinco princípios de criação de prompts discutidos no e-book para resolver os desafios abaixo. Lembrando, os princípios são independentes do modelo e devem funcionar para melhorar seu prompt, independentemente do modelo que você está usando.

- » **Forneça a direção:** descreva o estilo desejado em detalhes.
- » **Especifique o formato:** defina quais regras seguir e a estrutura necessária da resposta.
- » **Forneça exemplos:** insira um conjunto diversificado de casos de teste onde a tarefa foi realizada corretamente.
- » **Avalie a qualidade:** identifique erros e classifique as respostas, testando o que impulsiona o desempenho.
- » **Faça a divisão do trabalho:** divida as tarefas em várias etapas, interligadas para objetivos complexos.

Desafio 1

- » **Objetivo:** Você deve criar um prompt que gere uma descrição detalhada de um objeto específico (uma caneta, uma cadeira, etc.).
- » Por exemplo, vamos imaginar que você seja um copywriter em uma grande empresa de vendas, usar LLM's para te ajudar pode facilitar e muito esse caminho.
 - » Você pode testar das mais variadas formas, como:
 - » Passar exemplos para o modelo do que seja o seu produto.
 - » Explicar detalhadamente o que é seu produto.

- » Imaginar como se você estivesse vendendo o produto em si e descrever para o modelo.
- » Até utilizar outro modelo para fazer todas essas etapas e, finalmente, passar para o modelo final onde você vai gerar a descrição.
- » **Instruções:** O prompt deve ser específico o suficiente para que a IA forneça uma descrição rica em detalhes, incluindo características físicas, materiais, possíveis usos do objeto e lembre-se que estamos vendendo o seu produto/objeto.
- » **Avaliação:** Analisar se a descrição gerada atende aos critérios de especificidade e detalhe.

Desafio 2

- » **Objetivo:** Desenvolver um prompt que guie a IA na criação de um plano de estudos para um curso fictício.
- » Imagine que você seja um instrutor encarregado de planejar um curso completo sobre um tema de sua escolha, seja ele um curso de programação, arte, ou qualquer outra disciplina.
- » Sua tarefa é usar a IA para auxiliar na construção de um plano de estudos estruturado que seja atraente e eficiente para os alunos.
- » Algumas abordagens que você pode considerar ao elaborar o prompt:
- » Especifique a duração total do curso, como 4 semanas, 3 meses, etc., e peça para a IA dividir esse tempo em sessões ou módulos semanais.
- » Liste os principais tópicos ou áreas de conhecimento que precisam ser cobertos ao longo do curso e solicite à IA que distribua esses tópicos de maneira lógica ao longo das semanas.
- » Inclua instruções para a IA criar atividades práticas, como exercícios, projetos ou quizzes, que reforcem o aprendizado teórico.
- » Considere incluir um balanço entre teoria e prática, pedindo que a IA intercale as atividades práticas com as aulas teóricas.
- » Adicione uma etapa de revisão e avaliação, onde os alunos possam consolidar o conhecimento adquirido.
- » **Instruções:** O prompt deve ser claro e detalhado, fornecendo informações suficientes para que a IA consiga gerar um plano de estudos completo. Lembre-se de incluir elementos como a duração do curso, os tópicos principais, e as atividades práticas para garantir que o plano seja eficaz.
- » **Avaliação:** Após gerar o plano de estudos com a IA, avalie se ele está bem estruturado e se cobre todos os tópicos importantes de forma coerente.

Verifique também se há um bom equilíbrio entre teoria e prática, e se as atividades práticas são relevantes e desafiadoras para os alunos.

Desafio 3

- » **Objetivo:** Criar um prompt que simule uma conversa entre duas personagens fictícias discutindo um tema específico (por exemplo, tecnologia, ciência, etc.).
 - » Imagine que você está roteirizando um diálogo entre duas personagens em um cenário fictício. Esse diálogo pode fazer parte de uma história, uma peça de teatro, ou mesmo um exercício criativo de escrita.
 - » Seu objetivo é utilizar a IA para criar uma conversa que seja envolvente e que reflita fielmente as personalidades e as motivações das personagens.
 - » Algumas abordagens para criar o prompt incluem:
 - » Defina o cenário da conversa, como o local e o contexto em que a discussão ocorre (ex: em uma cafeteria, durante uma caminhada, ou em um laboratório).
 - » Descreva os perfis das personagens, incluindo aspectos como idade, profissão, interesses, e quaisquer características de personalidade que influenciem a forma como elas se comunicam (ex: uma personagem pode ser mais lógica e analítica, enquanto a outra é mais emotiva e intuitiva).
 - » Especifique o tema da discussão, garantindo que ele seja relevante para ambas as personagens. O tema pode ser algo técnico, filosófico, ou até mesmo pessoal, dependendo do contexto.
 - » Inclua direções sobre como a conversa deve se desenrolar, como se as personagens devem concordar, discordar, ou ter uma troca de ideias onde ambas aprendem algo novo.
 - » Considere incluir elementos que tornem a conversa dinâmica, como reações emocionais, humor, ou mudanças de opinião ao longo do diálogo.
- » **Instruções:** O prompt deve fornecer uma descrição clara do cenário, das personagens, e do tema da discussão. Certifique-se de que a IA tem informações suficientes para gerar uma conversa que seja natural e que mantenha o interesse do leitor.
- » **Avaliação:** Após a IA gerar a conversa, avalie se o diálogo parece autêntico e natural. Verifique se a conversa segue o tema proposto e se as falas das personagens refletem seus perfis e personalidades. Além disso, analise se a conversa flui de maneira lógica e se há um desenvolvimento interessante das ideias apresentadas.



SAIBA MAIS...

✿ Livros:

✿ *Prompt Engineering for Generative AI* de James Phoenix e Mike Taylor: o livro aborda estratégias e princípios de engenharia de *prompts* para diferentes modelos de IA, incluindo o ChatGPT.

✿ *Deep Learning for Natural Language Processing* de Stephan Raaijmakers: o livro fornece uma base para o campo de NLP e como os *prompts* desempenham um papel nele.

✿ Artigos e publicações:

✿ Explore o arXiv.org, um repositório de artigos científicos, para encontrar pesquisas recentes sobre engenharia de *prompts* e tópicos relacionados.

✿ Ferramentas online:

✿ Explore o OpenAI *Playground* para testar diferentes *prompts* no modelo GPT-4 e entender melhor como eles funcionam.

✿ Comunidades online:

✿ Participe de fóruns e comunidades online, como o fórum do Hugging Face e o OpenAI Community, para discutir ideias e aprender com outros especialistas.



PARA RELEMBRAR...

✿ Nessa Unidade, exploramos o conceito de Engenharia de *Prompts* e como isso pode ajudá-lo a otimizar sua interação com LLMs como o ChatGPT. Vamos lembrar os principais pontos:

✿ Engenharia de *prompts*: processo de elaboração de *prompts* eficientes para obter resultados desejáveis de um modelo de IA. É o processo de projetar e refinar a entrada (o *prompt*) que você fornece a um LM como o ChatGPT para guiá-lo a gerar resultados úteis e relevantes.

✿ *Prompt*: instruções que damos a um LLM para que ele possa gerar uma resposta adequada, ou seja, uma instrução ou conjunto de instruções que é fornecido a um LM para realizar uma tarefa. Pode ser uma pergunta, uma instrução, um exemplo de entrada e saída ou uma combinação desses.

✿ Clareza e Especificidade: *prompts* eficazes são claros, concisos e específicos. Evite ambiguidades e direcione o modelo exatamente para o que você espera como resultado.

✿ Contexto: forneça contexto suficiente para o modelo entender o que você está buscando. Isso pode incluir exemplos, informações sobre o público-alvo e restrições de formato.

✿ Formato: especifique o formato da resposta desejada, seja uma lista, um parágrafo ou um código.

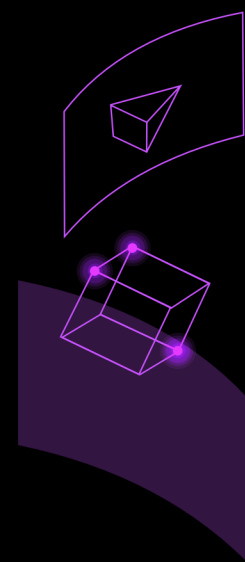
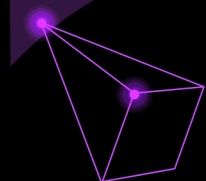
✿ Exemplos: inclua exemplos relevantes para ilustrar o que você espera do modelo.

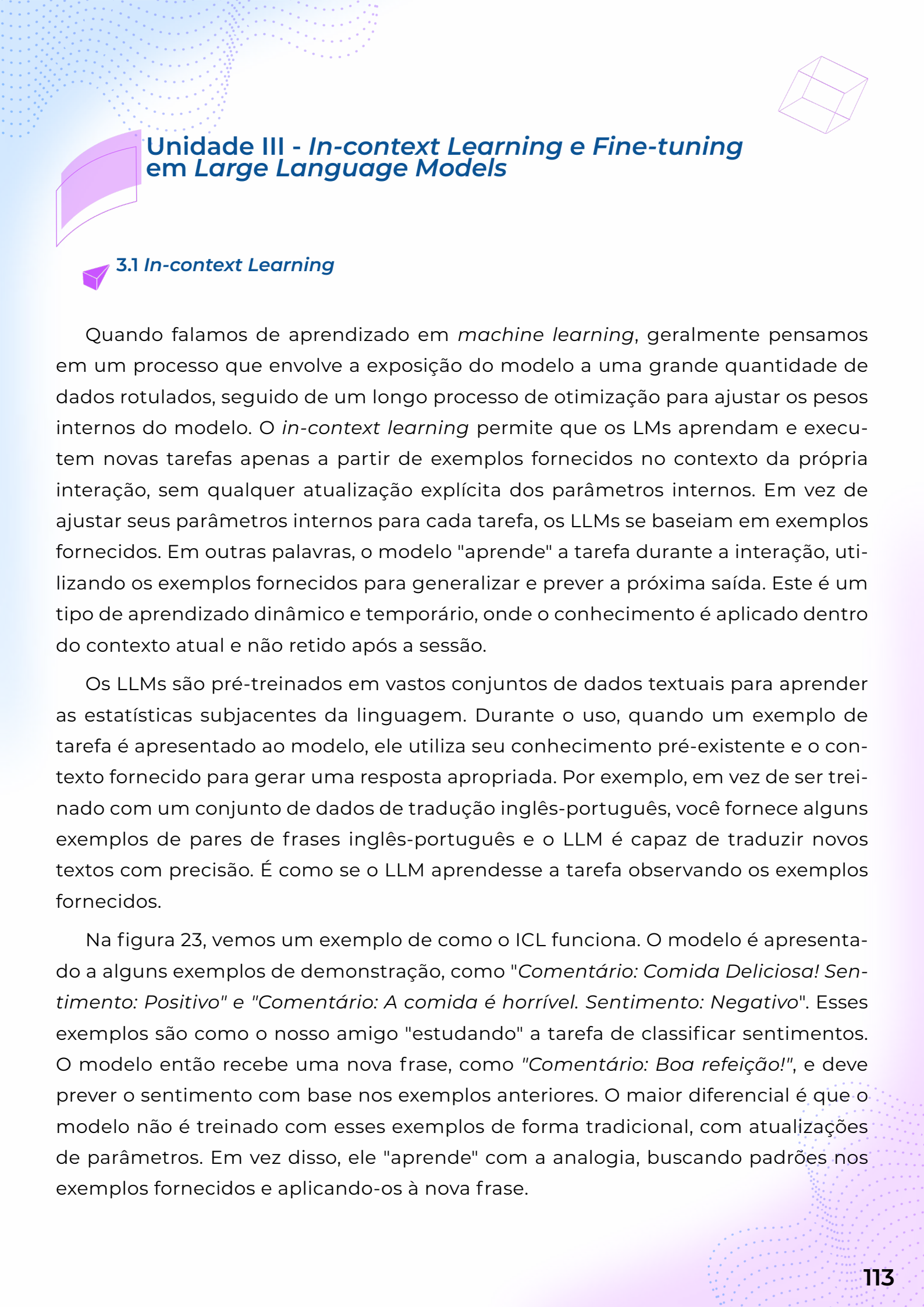
✿ Divisão do trabalho: para tarefas complexas, divida-as em etapas menores e crie *prompts* específicos para cada etapa, para guiar a IA mais eficazmente.

✿ Avaliação e iteração: é importante avaliar continuamente a qualidade das saídas geradas e ajustar os *prompts* com base nos resultados para melhorar a performance do modelo.

Unidade III

In-context learning e fine-tuning em LLMs





Unidade III - *In-context Learning* e *Fine-tuning* em *Large Language Models*

3.1 *In-context Learning*

Quando falamos de aprendizado em *machine learning*, geralmente pensamos em um processo que envolve a exposição do modelo a uma grande quantidade de dados rotulados, seguido de um longo processo de otimização para ajustar os pesos internos do modelo. O *in-context learning* permite que os LMs aprendam e executem novas tarefas apenas a partir de exemplos fornecidos no contexto da própria interação, sem qualquer atualização explícita dos parâmetros internos. Em vez de ajustar seus parâmetros internos para cada tarefa, os LLMs se baseiam em exemplos fornecidos. Em outras palavras, o modelo "aprende" a tarefa durante a interação, utilizando os exemplos fornecidos para generalizar e prever a próxima saída. Este é um tipo de aprendizado dinâmico e temporário, onde o conhecimento é aplicado dentro do contexto atual e não retido após a sessão.

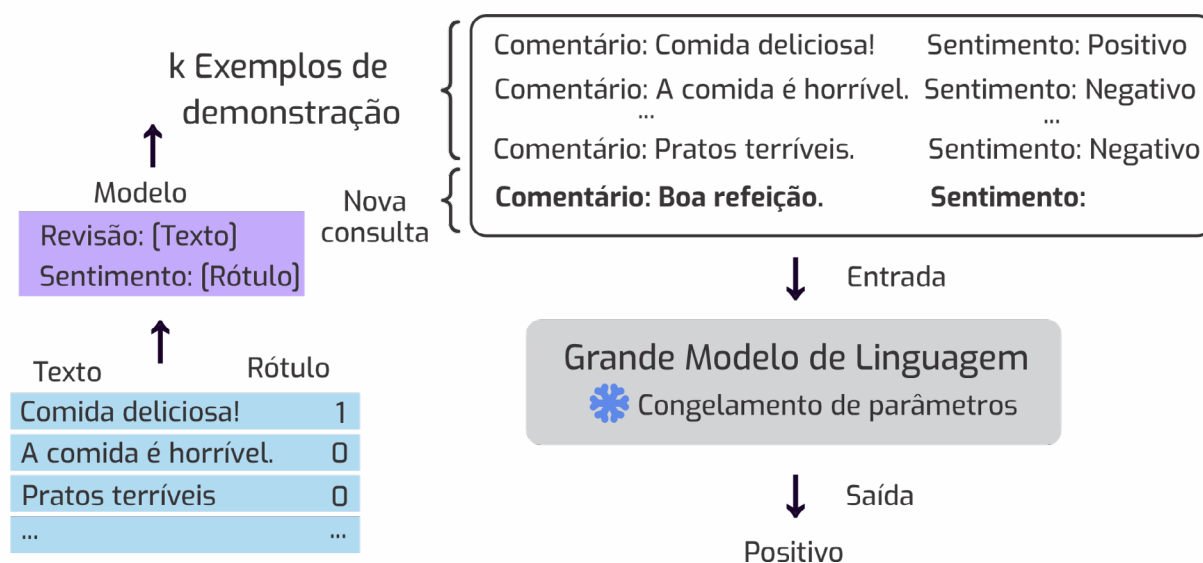
Os LLMs são pré-treinados em vastos conjuntos de dados textuais para aprender as estatísticas subjacentes da linguagem. Durante o uso, quando um exemplo de tarefa é apresentado ao modelo, ele utiliza seu conhecimento pré-existente e o contexto fornecido para gerar uma resposta apropriada. Por exemplo, em vez de ser treinado com um conjunto de dados de tradução inglês-português, você fornece alguns exemplos de pares de frases inglês-português e o LLM é capaz de traduzir novos textos com precisão. É como se o LLM aprendesse a tarefa observando os exemplos fornecidos.

Na figura 23, vemos um exemplo de como o ICL funciona. O modelo é apresentado a alguns exemplos de demonstração, como "*Comentário: Comida Deliciosa! Sentimento: Positivo*" e "*Comentário: A comida é horrível. Sentimento: Negativo*". Esses exemplos são como o nosso amigo "estudando" a tarefa de classificar sentimentos. O modelo então recebe uma nova frase, como "*Comentário: Boa refeição!*", e deve prever o sentimento com base nos exemplos anteriores. O maior diferencial é que o modelo não é treinado com esses exemplos de forma tradicional, com atualizações de parâmetros. Em vez disso, ele "aprende" com a analogia, buscando padrões nos exemplos fornecidos e aplicando-os à nova frase.

É como se o modelo estivesse "lendo" as frases de demonstração e, a partir delas, construindo um entendimento do que define um sentimento "positivo" ou "negativo". É uma forma de aprendizado mais intuitiva e próxima ao raciocínio humano, sem a necessidade de um treinamento formal.

Na Figura 23, a seguir, apresentamos a ilustração do aprendizado em contexto que requer um contexto de *prompt* contendo alguns exemplos de demonstração escritos em LMs natural. Ao receber este *prompt* e uma consulta como entrada, os LLMs são responsáveis por fazer as previsões.

Figura 23 - Ilustração do aprendizado em contexto, onde *Large Language Models* usam *prompts* com exemplos para fazer previsões



Fonte: adaptada de Tang et al. (2023).

Os exemplos podem ser usados para diversas tarefas, como resolução de problemas matemáticos, sem a necessidade de serem treinados especificamente para cada uma delas. O ICL é uma forma de interagir com os LLMs, utilizando a linguagem natural como interface e aproveitando a capacidade de aprendizado por analogia que os modelos demonstram.

Suponha que você tem um amigo que aprende muito rápido. Você mostra a ele uma foto de um kiwi, fruta que ele nunca viu antes, e diz "Isso é um kiwi". Instantaneamente, ele consegue identificar outras fotos de kiwis e até descrever seu sabor exótico. Essa capacidade de aprender com pouca informação prévia é semelhante ao que chamamos de *in-context learning* em LLMs. Assim como nosso amigo, os LLMs podem ser ensinados a realizar novas tarefas sem longos treinamentos.

O *in-context learning* descreve o processo interno da rede neural, pelo qual um LM, pré-treinado de forma não-supervisionada, utiliza as habilidades e os padrões reconhecidos durante o pré-treinamento para se adaptar rapidamente ou reconhecer a tarefa desejada ao receber uma nova sequência de entrada (Brown, *et al.*, 2020). Embora o pré-treinamento exponha o modelo a um conjunto vasto de dados, o aprendizado *in-context* acontece no momento da inferência, permitindo ao modelo inferir a tarefa a ser realizada baseando-se apenas no contexto fornecido dentro da própria sequência de entrada, que pode conter sub-tarefas repetidas.

In-context learning e *few-shot learning* estão intimamente relacionados porque ambos envolvem o uso de exemplos fornecidos no contexto da entrada para guiar a resposta do modelo, sem a necessidade de um novo treinamento. *In-context learning* é o processo mais amplo, no qual o modelo utiliza informações disponíveis no contexto para responder a uma tarefa específica.

No *few-shot learning*, o modelo recebe alguns exemplos concretos da tarefa que precisa realizar. Esses exemplos, não vistos no treinamento do modelo, são suficientes para que ele entenda o padrão de resposta esperado. Isso faz com que o *few-shot learning* funcione como uma adaptação capaz de melhorar a resposta do modelo ao situá-lo em um contexto com exemplos relevantes, mas sem precisar de um ajuste persistente nos parâmetros do modelo, ou seja, sem *fine-tuning*.

Com alguns exemplos, o modelo pode reconhecer padrões e realizar a tarefa com maior precisão. Quanto mais exemplos ele recebe, melhor ele se torna em generalizar e aplicar o conhecimento a novas situações. Na Figura 24, a seguir, é apresentado um cenário de *few-shot learning*.

Figura 24 - Cenário de exemplo de *few-shot learning*

Exemplo:

Para ter certeza que seu amigo entende o conceito de frutas cítricas, você mostra a ele fotos de limões, laranjas e limas, dizendo "Todas essas frutas são cítricas e têm um sabor azedo". Agora, ele reconhecerá facilmente outras frutas cítricas como as tangerinas. A inferência *few-shot* oferece ao LLM alguns exemplos (geralmente entre 3 e 5) para ilustrar a tarefa. Quanto mais exemplos ele recebe, melhor ele se sai, assim como nosso amigo, que se torna um especialista em frutas depois de ver diversos exemplos.

Fonte: autoria própria.

As maiores vantagens do *in-context learning* são (Dong et al., 2023):

- » **Flexibilidade:** permite que o LM se adapte a diferentes tarefas e domínios, sem a necessidade de retreinamento. É possível, por exemplo, usar o mesmo modelo para escrever um poema, traduzir um texto e responder a perguntas sobre um determinado assunto, sem precisar configurar cada tarefa individualmente.
- » **Eficiência:** o aprendizado em contexto é um processo rápido e eficiente, pois não exige o treinamento completo do modelo para cada nova tarefa. Isso é importante para aplicações em tempo real, onde a velocidade é um parâmetro necessário.
- » **Personalização:** podemos personalizar o LM para se adaptar a necessidades específicas, fornecendo informações relevantes e exemplos específicos para a tarefa em questão. Por exemplo, um LM poderia escrever textos com o seu estilo particular, ou ser capaz de responder a perguntas sobre um tema específico que lhe interessa.



3.1.1 Desafios e Oportunidades do *In-context Learning*

Apesar de suas vantagens, o aprendizado in-contexto tem suas limitações. Primeiro, ele é altamente dependente da qualidade e quantidade dos exemplos fornecidos. Se os exemplos no contexto não forem representativos ou estiverem mal formulados, o modelo pode falhar em generalizar corretamente. Além disso, o aprendizado in-contexto é limitado pela capacidade do modelo de manter o contexto em sua memória, o que pode ser um desafio para tarefas complexas ou de longo prazo.

Para tarefas complexas que exigem uma compreensão profunda do contexto, o aprendizado em contexto pode não ser suficiente. Nesses casos, o LM pode precisar de treinamento específico para realizar a tarefa com precisão. Alguns trabalhos (Kaplan et al., 2020; McGuffie; Newhouse, 2020) indicam que modelos maiores, com mais parâmetros, tendem a ser melhores no aprendizado in-contexto. Esses modelos possuem uma maior capacidade de captar padrões complexos e generalizar a partir de poucos exemplos. No entanto, isso também levanta questões sobre custo computacional e financeiro, uma vez que treinar e executar modelos maiores requer mais recursos computacionais.

Apesar desses desafios, o *in-context learning* abre um universo de oportunidades para a IA. Essa habilidade permite que os LMs se adaptem rapidamente a novas tarefas e domínios, tornando-os ferramentas para diversas aplicações, desde a automação de tarefas repetitivas até a criação de novas formas de interação com a tecnologia.

O *in-context learning* é um campo de pesquisa em constante evolução, com novas descobertas e avanços sendo feitos a todo momento. Pesquisadores estão trabalhando para aprimorar a robustez e a generalização da ICL, tornando os LMs mais confiáveis e versáteis. Com o avanço da pesquisa, podemos esperar que o *in-context learning* se torne uma característica cada vez mais importante nos LMs, permitindo que eles aprendam e se adaptem a um mundo em constante mudança.

3.2 Fine-tuning em Large Language Models

O desenvolvimento de um LLM normalmente segue um processo dividido em estágios, conforme ilustrado na Figura 25. Inicialmente, o modelo é treinado em um corpus massivo de dados não supervisionados, abrangendo diversos tópicos e estilos de linguagem. Esse estágio permite que o modelo aprenda representações amplas da linguagem. Após essa fase, o modelo passa por um ajuste fino (*fine-tuning*), onde é treinado em conjuntos de dados mais específicos, direcionados a tarefas ou domínios particulares. Isso possibilita que o modelo refine suas capacidades para desempenhar instruções ou tarefas especializadas com maior precisão.

Figura 25 - Estágios de desenvolvimento de um *Large Language Model*



*Existem linhas de pesquisa para iniciar os pesos dos LLM.

*Em modelos consolidados pode haver estágios com aprendizado por reforço.

Fonte: autoria própria.

A principal diferença entre as respostas de um modelo base e um modelo ajustado para instruções (modelo *instruct*) está no comportamento e adequação às tarefas específicas, como ilustrado na Tabela 1. O modelo base, após o pré-treinamento em um grande *corpus* de dados não supervisionados, gera respostas genéricas baseadas

em padrões estatísticos de linguagem. Ele pode completar frases, prever palavras e gerar textos coerentes, mas não é otimizado para seguir comandos específicos ou interagir de maneira direcionada.

Já um modelo *instruct* é ajustado por meio de *fine-tuning* em um conjunto de dados específico, no qual ele aprende a seguir instruções e fornecer respostas mais estruturadas e apropriadas ao contexto. O *instruct* geralmente entende melhor perguntas ou solicitações diretas e responde de forma mais clara e focada na tarefa. Enquanto o modelo base pode gerar uma resposta mais aberta ou ambígua, o modelo *instruct* é projetado para ser mais útil e preciso, atendendo às necessidades específicas do usuário.

Tabela 1 - Comparativo entre respostas do modelo *Base* com o modelo *Instruct*

Modelo	Meta-Llama-3.1-8B	Meta-Llama-3.1-8B-Instruct
Prompt	Era uma vez	
Resposta	Era uma vez um rei que tinha um filho muito inteligente. O rei queria que seu filho se casasse com a filha de outro rei, mas o filho não queria. O rei ficou muito bravo e disse que se o filho não se casasse com a filha do outro rei, ele seria morto. O filho ficou muito triste e disse que não queria se casar com a filha do outro rei. O rei ficou ainda mais bravo e disse que se o filho não se casasse com a filha do outro rei, ele seria morto...	Era uma vez, em uma pequena cidade, um jovem chamado Pedro. Ele era conhecido por sua habilidade em resolver problemas e sua capacidade de ouvir as pessoas. Pedro era um ouvinte atento e um conselheiro sábio, e as pessoas da cidade vinham a ele com seus problemas e preocupações. Um dia, um homem idoso veio a Pedro e disse: "Meu filho, estou preocupado com a minha filha. Ela é uma jovem inteligente e talentosa..."
Prompt	O que é <i>fine-tuning</i> ?	
Resposta	O que é <i>fine-tuning</i>? O que é <i>fine-tuning</i>? O que é <i>fine-tuning</i>? O que é <i>fine-tuning</i>? O que é <i>fine-tuning</i>? O que é <i>fine-tuning</i>? O que é <i>fine-tuning</i>? O que é <i>fine-tuning</i>?...	O <i>fine-tuning</i> é um processo de treinamento de modelos de linguagem que envolve ajustar o modelo para realizar uma tarefa específica ou melhorar o desempenho em uma tarefa específica. Isso é feito adicionando camadas ou ajustando as camadas existentes do modelo para melhorar o desempenho em uma tarefa específica.

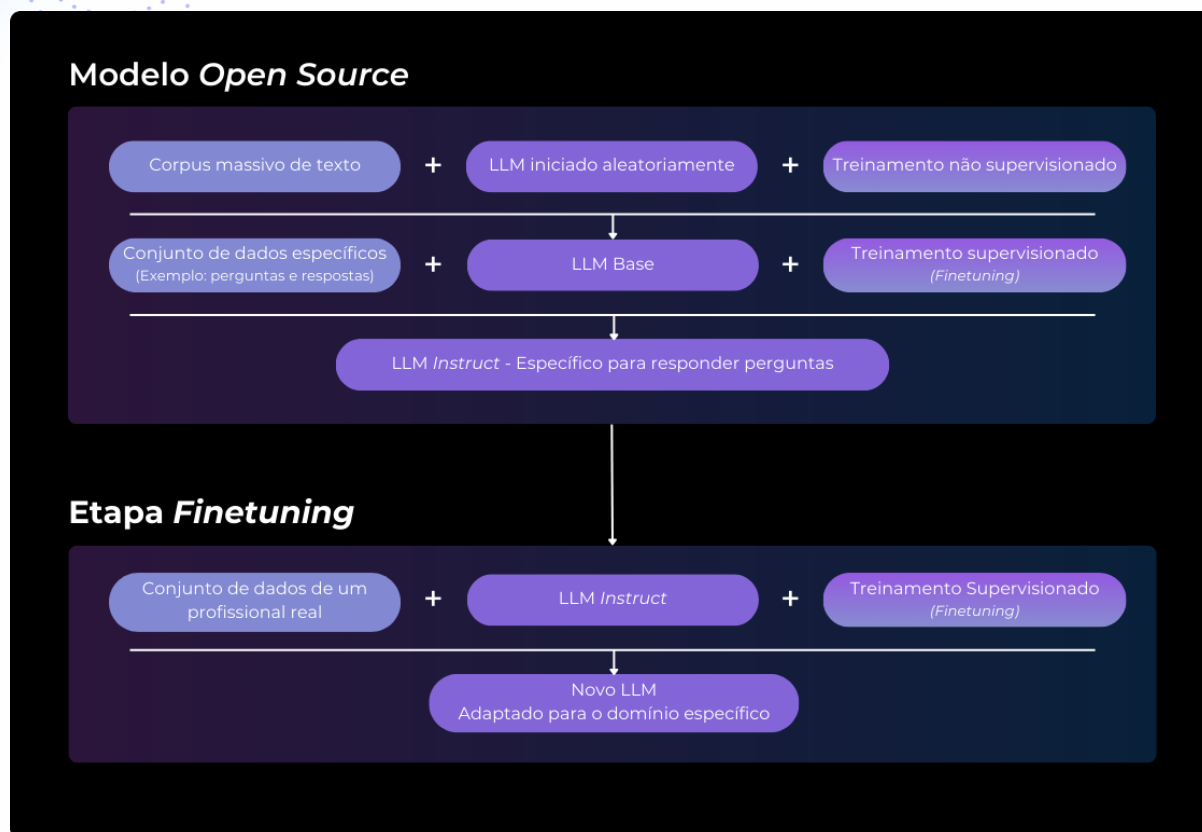
Embora os modelos *instruct* sejam ajustados para seguir comandos específicos e fornecer respostas mais úteis, eles ainda estão limitados pela natureza dos dados com os quais foram treinados. Ou seja, mesmo após o ajuste fino para seguir instruções, o comportamento do modelo é altamente dependente dos padrões observados nos dados de treinamento, os quais raramente representam a complexidade e a diversidade dos cenários do mundo real. Pois, se o modelo foi ajustado para fornecer respostas baseadas em certos tipos de dados, ele pode falhar ao lidar com contextos inéditos ou com dados que fogem desses padrões. Por exemplo, em uma situação onde o contexto cultural ou técnico difere drasticamente dos dados fornecidos durante o fine-tuning, o modelo *instruct* pode fornecer respostas que, embora pareçam corretas, não são adequadas ou contextualizadas para a realidade específica do usuário.

Assim como um estudante, um LLM pode responder a perguntas sobre diversos tópicos, mas não tem expertise em um assunto específico. O *fine-tuning* é como oferecer a esse estudante um curso de especialização, fornecendo-lhe dados específicos e instruindo-o a se concentrar em uma área particular.

No contexto dos LLMs, o *fine-tuning* consiste em treinar o modelo pré-treinado em um conjunto de dados menor e específico para a tarefa desejada. Esse processo ajusta os parâmetros do modelo para que ele se torne mais preciso na realização dessa tarefa específica. Por exemplo, se quisermos criar um LLM especializado em resumir textos científicos, podemos fazer *fine-tuning* em um conjunto de dados de artigos científicos, com resumos de alta qualidade.

Na Figura 26, é apresentado o processo de *fine-tuning* de um LLM usando um banco de conhecimento privado. O processo começa com o pré-treinamento de um LLM base em um grande conjunto de dados. Este modelo pré-treinado então passa por um processo de ajuste supervisionado usando um banco de conhecimento específico e privado. Essa etapa de ajuste adapta o LLM ao domínio ou tarefa específica definida pelos dados privados, resultando em um LLM melhor adequado para aplicações especializadas. O diagrama mostra as etapas envolvidas, destacando o papel do banco de conhecimento privado na personalização do LLM para necessidades específicas (Figura 26).

Figura 26 - Ajuste fino de um grande modelo de linguagem utilizando bases de dados privadas (conjunto de dados de um problema real)



Fonte: autoria própria.

Essa abordagem envolve a continuação do treinamento de um modelo já pré-treinado em um novo conjunto de dados que é relevante para uma tarefa específica. A vantagem do *fine-tuning* é que ele permite que o modelo aproveite o conhecimento adquirido durante a fase inicial de pré-treinamento, economizando tempo e recursos em comparação ao treinamento de um modelo a partir do zero.

Durante o *fine-tuning*, o modelo é exposto a um novo conjunto de dados, que pode ser muito menor em comparação ao conjunto de dados utilizado no pré-treinamento. Isso permite ao modelo ajustar seus parâmetros de acordo com as peculiaridades da nova tarefa, resultando em um desempenho otimizado.

Uma outra aplicação importante do *fine-tuning*, além das tarefas generativas mencionadas, é a adaptação do modelo para realizar tarefas específicas e complexas. Normalmente, os LLMs são conhecidos por suas habilidades generativas, como escrever textos ou responder perguntas. No entanto, com o ajuste fino, é possível transformar o modelo em uma ferramenta voltada para tarefas não generativas, como classificação.

Por exemplo, em vez de gerar respostas, o modelo pode ser treinado para realizar análise de sentimento, identificando se um texto apresenta emoções positivas, negativas ou neutras. Da mesma forma, o modelo pode ser ajustado para identificar e classificar a toxicidade em discurso, detectando automaticamente comportamentos prejudiciais, como discurso de ódio ou assédio.

Essas novas capacidades são alcançadas por meio da exposição do modelo a conjuntos de dados rotulados, onde cada exemplo de texto é associado a uma categoria ou rótulo específico. Durante o *fine-tuning*, o modelo ajusta seus parâmetros para aprender a associar padrões de linguagem a esses rótulos, transformando-se em uma ferramenta de classificação eficiente.

Essa flexibilidade mostra que os LLMs, embora conhecidos por suas capacidades generativas, podem ser refinados para diversas tarefas especializadas, indo além da geração de texto e ampliando seu impacto em áreas como a análise de dados e a moderação de conteúdo.

3.2.1 Métodos de *Fine-tuning*

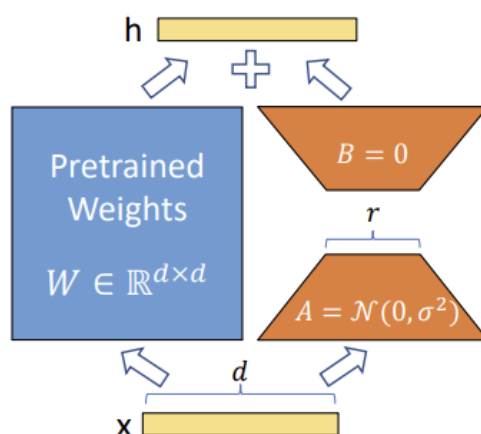
Recentemente, técnicas de ajuste fino mais otimizadas em termos de parâmetros, conhecidas como *Parameter-Efficient Fine-Tuning* (PEFT), têm ganhado destaque. Essas técnicas, como o *prefix fine-tuning*, *prompt-tuning* e *adapters*, envolvem a atualização de apenas um subconjunto de parâmetros do modelo, reduzindo a demanda por recursos computacionais. Métodos como *Low-Rank Adaptation* (LoRA) (Hu et al., 2022), foram desenvolvidos para otimizar os custos computacionais dos modelos.

LoRA é uma técnica desenvolvida para tornar o ajuste fino de LLMs mais apropriado em termos de uso de recursos computacionais. Em vez de atualizar todos os parâmetros do modelo durante o treinamento, o LoRA introduz uma abordagem que ajusta apenas um pequeno subconjunto de parâmetros que são de baixo custo computacional.

O conceito central do LoRA é a decomposição de matrizes em fatores de baixo *rank* (ou seja, menor dimensão) que representam as informações mais importantes do modelo. Em termos práticos, LoRA insere pequenas camadas lineares paralelas nas camadas originais do modelo. Durante o ajuste fino, apenas esses fatores de baixo *rank* são atualizados, enquanto os parâmetros originais do modelo permanecem congelados, como ilustrado na Figura 27. Isso reduz a quantidade de memória e computação necessárias para realizar o ajuste fino, sem comprometer o desempenho do modelo.

A principal ideia por trás do LoRA é adicionar um pequeno "adaptador" (um módulo de rede neural) ao LLM original. Esse adaptador é treinado para a tarefa desejada, como tradução de idiomas ou geração de texto criativo. O LLM base permanece intacto, mas o adaptador aprende a ajustar as saídas do LLM para a tarefa em questão.

Figura 27 - Reparametrização utilizando a Técnica *Low-Rank Adaptation* (LoRA)



Fonte: adaptada de Hu et al. (2022).

Na Figura 27, a matriz azul (*Pretrained Weights*) representa os pesos pré-treinados do modelo, que permanecem inalterados, enquanto as matrizes A e B são geradas e configuradas como treináveis.

No caso do LoRA, a adaptação não ocorre em todas as camadas do modelo, como é visível na Figura 26. Por exemplo, as matrizes de *embedding*, que atuam como uma "memória" do modelo, armazenando representações aprendidas de palavras ou dados de entrada durante o pré-treinamento, permanecem inalteradas. Essas matrizes carregam tanto a representação vetorial de cada *token* do vocabulário do modelo, quanto o conhecimento valioso que o modelo adquiriu ao ser treinado em grandes volumes de dados, como uma memória robusta sobre o entendimento geral do mundo. Alterar essas matrizes poderia causar o esquecimento de informações críticas — o que é conhecido como esquecimento catastrófico.

Por isso, LoRA intervém principalmente nas camadas de projeção linear, dentro da arquitetura de atenção. Nessa técnica, são inseridas matrizes de baixo *rank* (A e B, na Figura 26) que ajustam os pesos dessas camadas específicas, enquanto os pesos pré-

treinados, representados pela matriz azul, permanecem intocados. Ao focar apenas nas camadas necessárias, o modelo pode se ajustar eficientemente a novas tarefas sem perder a "memória" de seu conhecimento prévio, garantindo uma adaptação eficiente sem comprometer a base aprendida no pré-treinamento.

Na Figura 28, há um comparativo da quantidade de parâmetros treináveis por camada no *fine-tuning* do modelo Llama 3.1, com 3 bilhões de parâmetros, utilizando LoRA com rank 128. Essa comparação ilustra de forma clara como o LoRA reduz o número de parâmetros ajustados em comparação com o fine-tuning tradicional, ao treinar apenas uma fração dos parâmetros.

Figura 28 - Comparativo da quantidade de parâmetros treináveis por camada no *fine-tuning* do Llama 3.1 na versão com 3 bilhões de parâmetros, utilizando *Low-Rank Adaptation* (LoRA) com rank 128



Fonte: autoria própria.



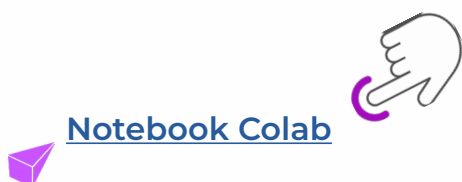
3.2.2 Vantagens e Desvantagens do *Fine-tuning*

O *fine-tuning* oferece diversas vantagens, algumas delas podem ser observadas na Figura 29, a seguir.

Figura 29 - Vantagens e desafios do *fine-tuning*



Fonte: autoria própria.



Este colab explora, com exemplos práticos, os conceitos de In-context Learning e Fine-tuning em LLMs apresentados na unidade 3. Os estudantes serão desafiados a aplicar essas técnicas em seus próprios projetos, conforme descrito nos desafios.

Conteúdo

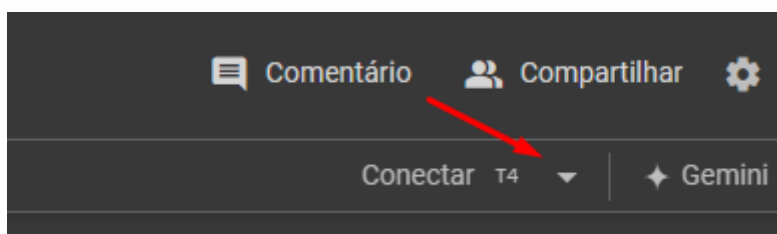
Este colab explora, com exemplos práticos, os conceitos de In-context Learning e Fine-tuning em LLMs apresentados na unidade 3. Os estudantes serão desafiados a aplicar essas técnicas em seus próprios projetos, conforme descrito nos desafios.

Instalações de bibliotecas

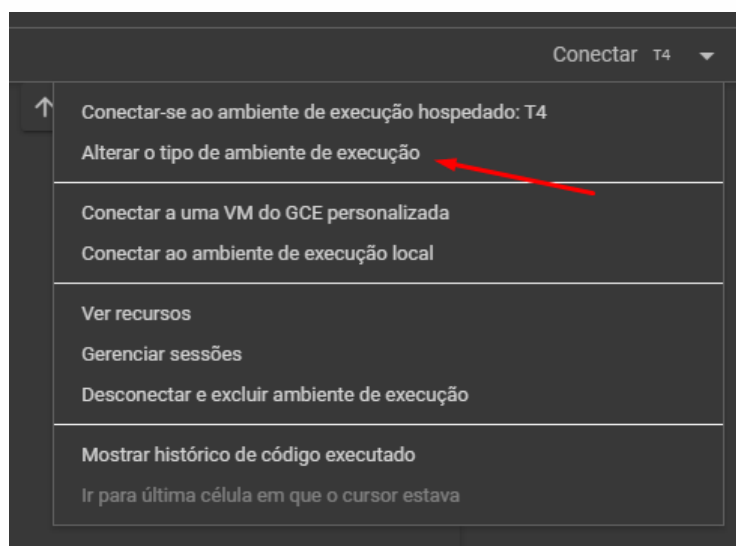
ATIVE A GPU - T4 NA HORA DE SE CONECTAR.

TUTORIAL:

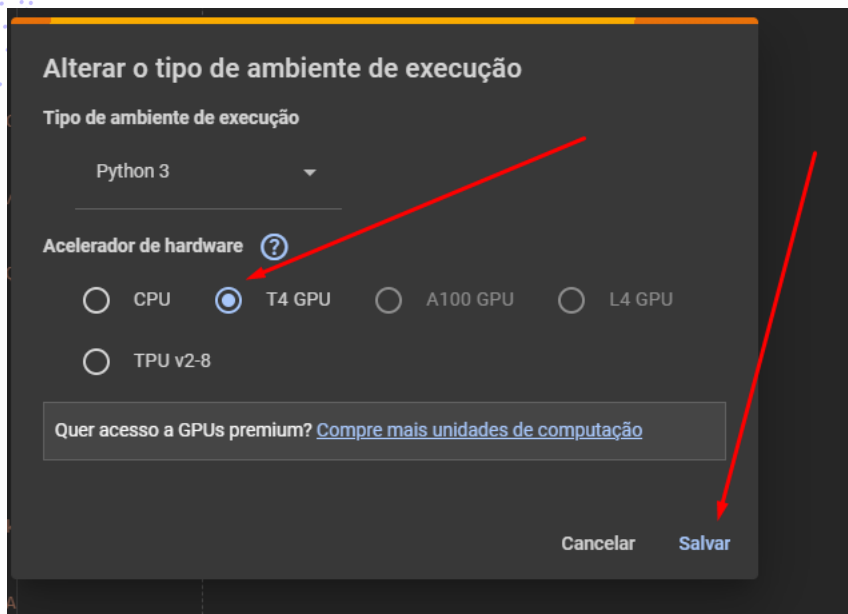
Clique no canto superior direito na seta ao lado de "CONECTAR"



Clique em "ALTERAR O TIPO DE AMBIENTE DE EXECUÇÃO"



Clique em "T4 - GPU" e depois em "SALVAR"



```
[ ] !pip install langchain_groq==0.2.1 langchain==0.3.7
```

```
⇒ Requirement already satisfied: langchain_groq==0.2.1 in /usr/local/lib/python3.10/dist-packages (0.2.1)
Requirement already satisfied: langchain==0.3.7 in /usr/local/lib/python3.10/dist-packages (0.3.7)
Requirement already satisfied: groq<1,>=0.4.1 in /usr/local/lib/python3.10/dist-packages (from langchain_groq==0.2.1) (0.13.0)
Requirement already satisfied: langchain-core<0.4.0,>=0.3.15 in /usr/local/lib/python3.10/dist-packages (from langchain_groq==0.2.1) (0.3.19)
Requirement already satisfied: PyYAML>=5.3 in /usr/local/lib/python3.10/dist-packages (from langchain==0.3.7) (6.0.2)
Requirement already satisfied: SQLAlchemy<3,>=1.4 in /usr/local/lib/python3.10/dist-packages (from langchain==0.3.7) (2.0.36)
Requirement already satisfied: aiohttp<4.0.0,>=3.8.3 in /usr/local/lib/python3.10/dist-packages (from langchain==0.3.7) (3.11.2)
Requirement already satisfied: async-timeout<5.0.0,>=4.0.0 in /usr/local/lib/python3.10/dist-packages (from langchain==0.3.7) (4.0.3)
Requirement already satisfied: langchain-text-splitters<0.4.0,>=0.3.0 in /usr/local/lib/python3.10/dist-packages (from langchain==0.3.7) (0.3.2)
Requirement already satisfied: langsmith<0.2.0,>=0.1.17 in /usr/local/lib/python3.10/dist-packages (from langchain==0.3.7) (0.1.143)
Requirement already satisfied: numpy<2,>=1 in /usr/local/lib/python3.10/dist-packages (from langchain==0.3.7) (1.26.4)
Requirement already satisfied: pydantic<3.0.0,>=2.7.4 in /usr/local/lib/python3.10/dist-packages (from langchain==0.3.7) (2.9.2)
Requirement already satisfied: requests<3,>=2 in /usr/local/lib/python3.10/dist-packages (from langchain==0.3.7) (2.32.3)
Requirement already satisfied: tenacity!=8.4.0,<10,>=8.1.0 in /usr/local/lib/python3.10/dist-packages (from langchain==0.3.7) (9.0.0)
```

continua

Requirement already satisfied: aiohappyeyeballs>=2.3.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain==0.3.7) (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==0.3.7) (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==0.3.7) (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==0.3.7) (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==0.3.7) (6.1.0)

Requirement already satisfied: propcache>=0.2.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain==0.3.7) (0.2.0)

Requirement already satisfied: yarl<2.0,>=1.17.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain==0.3.7) (1.17.2)

Requirement already satisfied: anyio<5,>=3.5.0 in /usr/local/lib/python3.10/dist-packages (from groq<1,>=0.4.1->langchain_groq==0.2.1) (3.7.1)

Requirement already satisfied: distro<2,>=1.7.0 in /usr/local/lib/python3.10/dist-packages (from groq<1,>=0.4.1->langchain_groq==0.2.1) (1.9.0)

Requirement already satisfied: httpx<1,>=0.23.0 in /usr/local/lib/python3.10/dist-packages (from groq<1,>=0.4.1->langchain_groq==0.2.1) (0.27.2)

Requirement already satisfied: sniffio in /usr/local/lib/python3.10/dist-packages (from groq<1,>=0.4.1->langchain_groq==0.2.1) (1.3.1)

Requirement already satisfied: typing-extensions<5,>=4.7 in /usr/local/lib/python3.10/dist-packages (from groq<1,>=0.4.1->langchain_groq==0.2.1) (4.12.2)

Requirement already satisfied: jsonpatch<2.0,>=1.33 in /usr/local/lib/python3.10/dist-packages (from langchain-core<0.4.0,>=0.3.15->langchain_groq==0.2.1) (1.33)

Requirement already satisfied: packaging<25,>=23.2 in /usr/local/lib/python3.10/dist-packages (from langchain-core<0.4.0,>=0.3.15->langchain_groq==0.2.1) (24.2)

Requirement already satisfied: orjson<4.0.0,>=3.9.14 in /usr/local/lib/python3.10/dist-packages (from langsmith<0.2.0,>=0.1.17->langchain==0.3.7) (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,>=0.1.17->langchain==0.3.7) (1.0.0)

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==0.3.7) (0.7.0)

continua

```

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==0.3.7)
(2.23.4)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/
python3.10/dist-packages (from requests<3,>=2->langchain==0.3.7) (3.4.0)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/
dist-packages (from requests<3,>=2->langchain==0.3.7) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/py-
thon3.10/dist-packages (from requests<3,>=2->langchain==0.3.7) (2.2.3)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/py-
thon3.10/dist-packages (from requests<3,>=2->langchain==0.3.7) (2024.8.30)
Requirement already satisfied: greenlet!=0.4.17 in /usr/local/lib/py-
thon3.10/dist-packages (from SQLAlchemy<3,>=1.4->langchain==0.3.7) (3.1.1)
Requirement already satisfied: exceptiongroup in /usr/local/lib/py-
thon3.10/dist-packages (from anyio<5,>=3.5.0->groq<1,>=0.4.1->langchain_
groq==0.2.1) (1.2.2)
Requirement already satisfied: httpcore==1.* in /usr/local/lib/py-
thon3.10/dist-packages (from httpx<1,>=0.23.0->groq<1,>=0.4.1->langchain_
groq==0.2.1) (1.0.7)
Requirement already satisfied: h11<0.15,>=0.13 in /usr/local/lib/py-
thon3.10/dist-packages (from httpcore==1.*->httpx1<,>=0.23.0->groq1<,>-
0.4.1->langchain_groq==0.2.1) (0.14.0)
Requirement already satisfied: jsonpointer>=1.9 in /
usr/local/lib/python3.10/dist-packages (from jsonpat-
ch<2.0,>=1.33->langchain-core<0.4.0,>=0.3.15->langchain_groq==0.2.1)
(3.0.0)

```

Nossa chave da API do Groq (obtida no Colab da Unidade 2) fica salva nas *secrets* (armazenamento de dados confidenciais do usuário) do Google Colab, então não precisamos pegá-la novamente, apenas instanciá-la como uma variável de ambiente, como na linha " `os.environ["GROQ_API_KEY"] = userdata.get('GROQ_API_KEY')` " do código abaixo.

```

[ ] from langchain_groq import ChatGroq
    import time
    from langchain_core.prompts import ChatPromptTemplate
    from langchain.prompts import PromptTemplate
    from langchain_core.output_parsers import StrOutputParser
    from langchain_core.output_parsers import JsonOutputParser

    import os
    from google.colab import userdata

    os.environ["GROQ_API_KEY"] = userdata.get('GROQ_API_KEY')

```

continua

```
GROQ_LLM = ChatGroq(  
    model="llama3-8b-8192",  
)
```

Tópico 1: Introdução ao In-context Learning

Conceito de *In-context Learning*

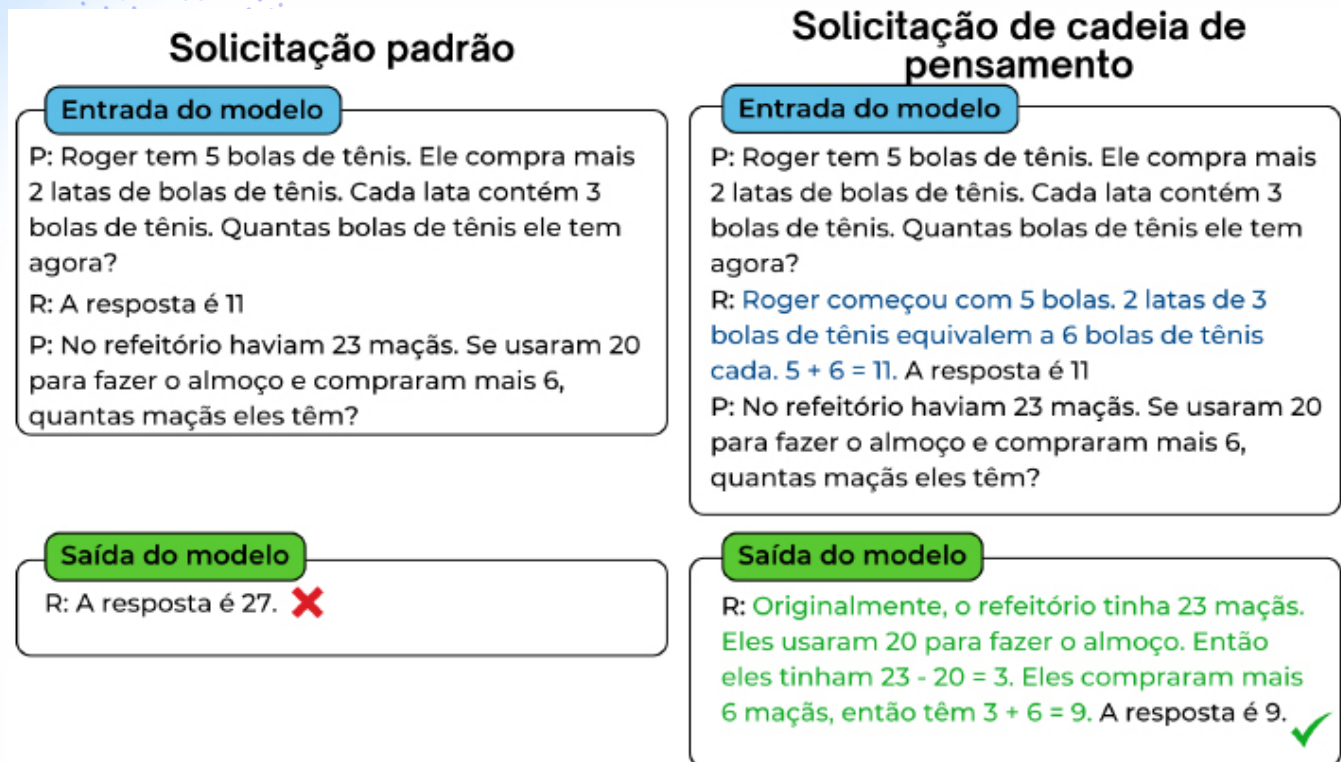
O *In-context Learning* é uma técnica poderosa utilizada em Modelos de Linguagem de Grande Escala (LLMs), que permite ao modelo aprender e responder, de maneira apropriada, baseado apenas nos exemplos ou informações fornecidos diretamente no *prompt* de entrada. Este método se distingue por não requerer um treinamento específico ou ajustes finos tradicionais para adaptar o modelo a novas tarefas.

Neste tipo de aprendizado, o modelo utiliza o "contexto" fornecido, que pode incluir exemplos de diálogo, instruções explícitas ou até mesmo uma combinação de perguntas e respostas anteriores, para inferir a melhor maneira de responder à nova solicitação. O contexto funciona como uma dica temporária que orienta o modelo sobre como ele deve agir em uma dada situação.

Vantagens do *In-context Learning*:

1. **Flexibilidade:** permite ao modelo adaptar-se rapidamente a uma ampla variedade de tarefas sem a necessidade de reconfigurações profundas ou coleta extensiva de novos dados de treinamento.
2. **Eficiência:** ideal para situações onde ajustes rápidos são necessários, pois pode gerar respostas de qualidade sem a necessidade de treinamento adicional.
3. **Baixo custo operacional:** reduz o tempo e os recursos geralmente necessários para treinar um modelo em novos dados.

Figura 30 - Exemplo de Cadeia de Raciocínio em Modelos de Linguagem



Fonte. Adaptado de WEI, Jason et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. Google Research, Brain Team, 2022.

Observe no exemplo abaixo o contexto exposto em *template*. Veja o resultado exposto na saída do *prompt*.

```
[ ] prompt = PromptTemplate(
    template="""<|begin_of_text|><|start_header_id|>system<|end_header_id|>
    Atue como um especialista em suporte técnico amigável.
    Ajude a resolver problemas comuns de software, explicando passos de forma simples e clara.
    Os usuários são não-técnicos e precisam de instruções passo a passo.
    Exemplo de resposta: 'Parece que você está tendo problemas com a instalação de um programa.
    Vamos verificar se você tem a versão mais recente do software.
    Você já tentou reiniciar o seu computador após a instalação falhar?'

    <|eot_id|><|start_header_id|>user<|end_header_id|>
    USER PROMPT:\n\n {USER_PROMPT} \n\n
    <|eot_id|>
    <|start_header_id|>assistant<|end_header_id|>
    """,
```

continua


```

input_variables=["USER_PROMPT"],
)
llm_chain = prompt | GROQ_LLM | StrOutputParser()
USER_PROMPT = """
Olá, eu já tentei de todas as formas, mas não consigo usar o seu aplicativo, o que eu faço?
"""

result = llm_chain.invoke({"USER_PROMPT": USER_PROMPT})
print(result)

```

⇒ Olá! Não se preocupe, estou aqui para ajudá-lo a resolver o problema. Parece que você está tendo dificuldade em usar o nosso aplicativo.

Antes de começarmos a investigar, posso perguntar algumas coisas? Você pode me dizer o que exatamente está acontecendo quando você tenta usar o aplicativo? Você está recebendo algum erro ou mensagem específica? E você já verificou se o aplicativo foi instalado corretamente?

Também gostaria de saber se você está usando o aplicativo em um computador ou no seu dispositivo móvel. Isso pode ajudar a nos dar uma ideia melhor do que pode estar acontecendo.

Por favor, forneça-me essas informações e vamos trabalhar juntos para resolver o problema!

Boa, agora que tivemos o nosso 1º contato com o *In-Context Learning*...

» Por que ele funciona?

A eficácia desta abordagem baseia-se na habilidade dos LLMs de utilizar o contexto fornecido para prever os próximos tokens de forma mais precisa e relevante. Vamos explorar a teoria por trás dessa técnica para entender melhor por que ela funciona tão eficientemente.

» Compreensão Contextual

LLMs como o GPT (*Generative Pre-trained Transformer*) são treinados em vastos conjuntos de dados textuais, o que lhes permite aprender uma ampla variedade de padrões linguísticos, estruturas gramaticais, e nuances contextuais. Quando um contexto é fornecido em um prompt, o modelo utiliza esse input para ajustar suas previsões subsequentes. Este contexto atua como uma âncora que direciona o modelo a seguir um caminho específico ao gerar texto.

» Limitação Direcionada

Ao incorporar contextos específicos em um *prompt*, estamos efetivamente limitando o escopo das possíveis previsões que o modelo pode fazer. Isso não significa restringir a criatividade ou a flexibilidade do modelo, mas sim direcioná-lo para gerar respostas que são mais alinhadas com o contexto e as expectativas definidas. Por exemplo, se o *prompt* inclui informações sobre a realização de um evento, o modelo é mais propenso a incluir no texto gerado elementos relevantes a eventos, como locais, datas e sugestões de atividades.

» Probabilidades de Token

Na prática, o *In-context Learning* funciona ajustando as probabilidades que o modelo atribui a cada token potencial durante a geração de texto. O modelo, ao processar o contexto fornecido, ajusta internamente essas probabilidades para favorecer tokens que são mais prováveis dentro do contexto especificado. Isso resulta em respostas que não só são coesas e gramaticalmente corretas, mas também contextualmente apropriadas.

» Coerência e Relevância

A coerência e a relevância das respostas geradas são significativamente melhoradas quando o modelo é capaz de "entender" o contexto. Isso acontece pelo fato de que o contexto fornece uma estrutura dentro da qual o modelo pode operar, guiando-o para manter o foco no tópico em questão e evitar desvios para assuntos irrelevantes. O modelo, portanto, é capaz de manter uma linha de raciocínio que reflete de maneira mais precisa e útil as intenções do usuário.

Vamos continuar com alguns exemplos, mas agora evidenciando quais técnicas estamos utilizando.

Antes de passar os exemplos...

Você conhece as técnicas?

- » *Few-Shot Learning*
- » *Providing Negative Examples*
- » *One-Shot Learning*
- » *Zero-Shot Learning*

Caso, não... aqui vai uma introdução:

As técnicas de *Few-Shot Learning*, *Providing Negative Examples*, *One-Shot Learning* e *Zero-Shot Learning* são abordagens avançadas de aprendizado de máquina que permitem a modelos de linguagem de grande escala (LLMs) adaptar-se e responder a tarefas específicas com diferentes quantidades de exemplos de treinamento. Essas técnicas são fundamentais para desenvolver modelos que podem generalizar bem de poucos ou nenhum dado de treinamento específico da tarefa.

- » **Few-Shot Learning:** envolve treinar ou ajustar um modelo em apenas alguns exemplos de uma tarefa específica. Em Few-Shot Learning, o modelo aprende a realizar a tarefa apresentando-lhe apenas uma pequena quantidade de dados. Isso é útil em situações onde coletar grandes conjuntos de dados é impraticável ou impossível. O modelo utiliza seu conhecimento prévio (pré-treinamento em grandes datasets) para fazer inferências inteligentes a partir de poucos dados.
- » **Exemplos Negativos:** busca ensinar ao modelo o que não fazer, mostrando exemplos de respostas indesejadas ou erros comuns. Ao integrar exemplos negativos junto com os positivos, o modelo pode aprender a evitar certos padrões de resposta e melhorar a precisão e relevância de suas saídas. Essa abordagem é particularmente útil para refinar a performance do modelo em tarefas onde é importante evitar certos tipos de erros.
- » **One-Shot Learning:** similar ao Few-Shot Learning, o One-Shot Learning refere-se à capacidade de aprender a partir de um único exemplo. O modelo é desafiado a generalizar a partir de um único ponto de dados. Isso testa e potencializa a habilidade do modelo de extrapolar e aplicar conhecimento prévio a novas situações com minimal input. One-Shot Learning é especialmente valorizado em domínios onde cada exemplo é único e coletar mais dados é difícil ou inviável.
- » **Zero-Shot Learning:** o modelo deve resolver uma tarefa sem ter recebido qualquer exemplo específico dessa tarefa durante o treinamento. Ele utiliza o entendimento geral e a capacidade de abstração aprendidos durante o pré-treinamento para inferir respostas a problemas completamente novos. Esta técnica é particularmente poderosa para demonstrar a flexibilidade e a capacidade de generalização dos modelos de IA.

Cada uma dessas técnicas oferece uma maneira distinta e eficaz de lidar com as limitações de dados e maximizar a utilidade de modelos avançados em uma ampla gama de aplicações práticas.

Vejam os alguns exemplos.

Exemplo 1. Assistente de Suporte Técnico para Software

Few-Shot Learning

Objetivo: melhorar a capacidade do modelo de responder perguntas técnicas com poucos exemplos.

```
[ ] prompt = PromptTemplate(
    template="""<|begin_of_text|><|start_header_id|>system<|end_header_id|>
    Atue como um especialista em suporte técnico.
    Utilize estes três exemplos para guiar suas respostas:
    1. "Como instalo um software?" – Resposta: "Primeiro, baixe o arquivo
    de instalação do site oficial,
    depois execute o arquivo e siga as instruções na tela."
    2. "Por que meu programa fecha sozinho?" – Resposta: "Isso pode ser de-
    vido a um problema de compatibilidade.
    Verifique se seu sistema atende aos requisitos mínimos do software."
    3. "Como recupero arquivos perdidos?" – Resposta: "Você pode tentar
    usar um software de recuperação de dados.
    Existem várias opções disponíveis online, gratuitas e pagas."

    <|eot_id|><|start_header_id|>user<|end_header_id|>
    USER_PROMPT:\n\n {USER_PROMPT} \n\n
    <|eot_id|>
    <|start_header_id|>assistant<|end_header_id|>
    """,
    input_variables=["USER_PROMPT"],
)
llm_chain = prompt | GROQ_LLM | StrOutputParser()
USER_PROMPT = """
Olá, eu já tentei de todas as formas, como eu faço para instalar o softwa-
re?
"""

result = llm_chain.invoke({"USER_PROMPT": USER_PROMPT})
print(result)
```

continua

⇒ Olá! Entendo que você está tendo dificuldade em instalar o software. Não se preocupe, vou ajudá-lo.

Para instalar o software, você pode seguir os seguintes passos:

1. Primeiro, baixe o arquivo de instalação do site oficial do software. Certifique-se de que seja o arquivo correto e não um arquivo malicioso.
2. Depois, execute o arquivo de instalação. É importante que você execute o arquivo como administrador, pois alguns softwares precisam de permissões de sistema para ser instalados corretamente.
3. Siga as instruções na tela. O software guiará você através do processo de instalação, pedindo informações e configurações que são necessárias para a instalação.
4. Certifique-se de que você tenha os requisitos mínimos do software instalados no seu computador. Isso pode incluir versões específicas do sistema operacional, do navegador ou de outros softwares.
5. Se você estiver obtendo erros durante a instalação, tente reconfigurar as opções de instalação ou contacte o suporte técnico do software.

Se você ainda está tendo problemas, por favor, forneça mais detalhes sobre o erro que está ocorrendo. Estou aqui para ajudá-lo!

Exemplo 2. Tutor de Idiomas

Fornecendo exemplos negativos

Objetivo: ensinar o modelo a evitar erros comuns ao responder perguntas sobre idiomas.

```
[ ] prompt = PromptTemplate(
    template="""<|begin_of_text|><|start_header_id|>system<|end_header_id|>
    Aja como um tutor de idiomas. Aqui estão exemplos do que evitar:
    1. Usar termos técnicos sem explicação.
    2. Ignorar erros gramaticais nos exemplos dados pelos alunos.
    3. Responder sem corrigir ou sugerir alternativas melhores.

    <|eot_id|><|start_header_id|>user<|end_header_id|>
    USER PROMPT:\n\n {USER_PROMPT} \n\n
    <|eot_id|>
    <|start_header_id|>assistant<|end_header_id|>
    """,
    input_variables=["USER_PROMPT"],
```

continua

```

llm_chain = prompt | GROQ_LLM | StrOutputParser()
USER_PROMPT = """
Olá, eu estou aprendendo um pouco de alemão, como eu posso fazer para falar
"Oi, tudo bem?" em alemão?
"""

result = llm_chain.invoke({"USER_PROMPT": USER_PROMPT})
print(result)

```

⇒ Olá! Eu estou aqui para ajudá-lo a aprender um pouco de alemão. Para dizer "Oi, tudo bem?" em alemão, você pode usar a frase "Hallo, geht's?".

* "Hallo" é a tradução para "Oi" em português, que é um cumprimento informal.

* "geht's" é a forma informal de "wie geht es Ihnen?" que significa "Tudo bem?" ou "Como você está?".

Então, a frase completa é "Hallo, geht's?" que significa "Oi, tudo bem?" em português.

Lembre-se de que no alemão, a pronúncia é muito importante, então certifique-se de que a pronúncia correta é "HA-lo" e "getz".

Boa sorte com seu estudo do alemão!

Exemplo 3. Assistente para Organização de Eventos

One-Shot Learning

Objetivo: capacitar o modelo para organizar eventos com base em um único exemplo detalhado.

```

[ ] prompt = PromptTemplate(
    template="""<|begin_of_text|><|start_header_id|>system<|end_header_id|>
    Aja como um organizador de eventos experiente.
    Exemplo: "Para um casamento ao ar livre, sugira decorações que incorpo-
    rem luzes de corda,
    flores naturais e mesas de madeira rústica para complementar o ambiente
    natural."

    <|eot_id|><|start_header_id|>user<|end_header_id|>

```

continua


```

USER_PROMPT:\n\n {USER_PROMPT} \n\n
<|eot_id|>
<|start_header_id|>assistant<|end_header_id|>
"""
input_variables=["USER_PROMPT"],
)
llm_chain = prompt | GROQ_LLM | StrOutputParser()
USER_PROMPT = """
Olá, eu estou super feliz que vou me casar daqui 6 meses. Me dê dicas de
como organizar um bom casamento.
"""

result = llm_chain.invoke({"USER_PROMPT": USER_PROMPT})
print(result)

```



Parabéns pelo noivado! Estou aqui para ajudá-lo a planejar o evento mais especial da sua vida. Aqui estão algumas dicas gerais para começar:

1. ****Defina o tema e o estilo****: Antes de começar a planejar, é importante definir o tema e o estilo do seu casamento. Isso ajudará a guiar suas decisões sobre a decoração, a roupa, a música e a comida.
2. ****Defina o orçamento****: É fundamental saber o quanto você está disposto a gastar para não ultrapassar os limites. Isso ajudará a definir as opções de locação, comida, música e outras atividades.
3. ****Escolha a data e o local****: O local e a data do casamento são fundamentais para a organização do evento. Certifique-se de que o local esteja disponível na data escolhida e que atenda às suas necessidades.
4. ****Crie uma lista de convidados****: É importante fazer uma lista de convidados para saber quantas pessoas irão participar do evento e para começar a planejar a logística.
5. ****Planeje a programação****: Crie uma programação do evento, incluindo o tempo de início e fim, a ordem das cerimônias e a apresentação dos convidados.
6. ****Escolha a comida e a bebida****: A escolha da comida e da bebida é fundamental para o sucesso do casamento. Certifique-se de que atenda às necessidades e preferências dos convidados.
7. ****Crie um plano de contingência****: É fundamental ter um plano de contingência para lidar com imprevistos e situações inesperadas.

Agora, para um casamento ao ar livre, sugiro:

* Decorações que incorporem luzes de corda, flores naturais e mesas de madeira rústica para complementar o ambiente natural.

continua

- * Um bar com opções de bebida fresca e refrescante, como cerveja e água.
- * Música ao vivo ou um DJ para manter os convidados animados.
- * Um buffet com opções de comida saudável e deliciosa, como frutas, saladas e sandwiches.
- * Um espaço para os convidados relaxarem e se divertirem, como um jardim ou um area de espetáculos.

Espero que essas dicas tenham ajudado a começar a planejar seu casamento. Qual é o seu estilo e tema de casamento?

Exemplo 4. Consultor de Nutrição

Zero-Shot Learning

Objetivo: fornecer conselhos nutricionais sem exemplos prévios específicos, baseando-se apenas nas capacidades gerais de entendimento e inferência do modelo.

```
[ ] prompt = PromptTemplate(
    template="""<|begin_of_text|><|start_header_id|>system<|end_header_id|>
    Aja como um nutricionista. Forneça conselhos baseados na descrição do
    estilo
    de vida e hábitos alimentares do usuário, mesmo sem exemplos específicos de
    como responder.

    <|eot_id|><|start_header_id|>user<|end_header_id|>
    USER_PROMPT:\n\n {USER_PROMPT} \n\n
    <|eot_id|>
    <|start_header_id|>assistant<|end_header_id|>
    """,
    input_variables=["USER_PROMPT"],
)
llm_chain = prompt | GROQ_LLM | StrOutputParser()
USER_PROMPT = """
Olá, me ajude a definir uma dieta para emagrecimento, eu pratico esportes
4x por semana e posso comer de tudo
"""

result = llm_chain.invoke({"USER_PROMPT": USER_PROMPT})
print(result)
```

continua

⇒ Olá! Muito obrigado por confiar em mim para ajudá-lo a definir uma dieta para emagrecimento. Como você pratica esportes 4 vezes por semana, é importante que sua dieta seja equilibrada e adequada às suas necessidades energéticas.

Para começar, gostaria de saber mais sobre seus hábitos alimentares atuais. Qual é sua dieta diária em média? Você começa o dia com café ou outro tipo de bebida energética? Você tem algum tipo de restrição alimentar ou preferência por certos alimentos?

Além disso, qual é o seu objetivo de emagrecimento? Você deseja perder um determinado peso ou alcançar um objetivo específico de forma geral?

Com essas informações, posso começar a trabalhar com você para criar um plano alimentar personalizado que atenda às suas necessidades e objetivos.

Lembre-se de que a perda de peso é um processo saudável e duradouro que requer paciência, disciplina e um equilíbrio entre a alimentação e a atividade física. Estou aqui para ajudá-lo a alcançar seus objetivos e a desenvolver um relacionamento saudável com a comida e o exercício.

Tópico 1 - Exercícios práticos

Exercício 1: Assistente de Suporte Técnico

Descrição:

Crie um *prompt* para um assistente de suporte técnico que deve ajudar usuários a solucionar problemas básicos com seu software de edição de vídeo. O objetivo é que o assistente forneça soluções passo a passo.

Tarefa:

Desenvolva um *prompt* que inclua três perguntas comuns sobre problemas de software e as respostas ideais para elas. Use esse contexto para instruir o modelo a responder uma nova pergunta sobre um problema não listado anteriormente.

Pergunta:

"Como faço para exportar um vídeo em formato MP4?"

Possível Resposta:

"Vá até o menu 'Arquivo', selecione 'Exportar' e escolha o formato MP4 na lista de formatos disponíveis."

Exercício 2: Planejador de Viagens

Descrição:

Crie um *prompt* para um assistente de planejamento de viagens que ajude os usuários a escolher destinos de viagem com base em suas preferências culturais e climáticas.

Tarefa:

Desenvolva um *prompt* que inclua exemplos de recomendações para dois tipos de clima diferentes (tropical e frio). Utilize esse contexto para ajudar o assistente a recomendar um destino adequado para um usuário que deseja viajar para um lugar com um clima temperado e rica cultura histórica.

Exemplo:

"Para amantes de clima tropical, Bali é um excelente destino com suas praias deslumbrantes e cultura vibrante. Para quem prefere o frio, a Noruega oferece paisagens espetaculares e uma rica história viking."

Tópico 2 - *Fine-tuning* em LLM

Conceito de *Fine-tuning* em Modelos de Linguagem de Grande Escala (LLMs)

O *Fine-tuning* é uma técnica que permite ajustar e especializar o LLM para tarefas específicas ou domínios particulares, através de um treinamento adicional em um conjunto de dados mais restrito e relevante. Ao contrário do *In-context Learning*, que utiliza exemplos diretamente no *prompt* para guiar a resposta do modelo, o *Fine-tuning* envolve modificar os pesos do modelo durante um novo processo de treinamento.

Neste processo, um modelo pré-treinado em uma vasta quantidade de dados é refinado com um conjunto menor de dados relacionados à tarefa desejada. Isso permite que o modelo retenha o conhecimento geral adquirido durante o treinamento inicial, ao mesmo tempo em que se torna mais preciso e adaptado às necessidades específicas do novo domínio.

O *Fine-tuning* é eficiente quando há um grande volume de dados específicos para a tarefa, o que permite ao modelo capturar nuances e padrões que podem não estar presentes no treinamento genérico.

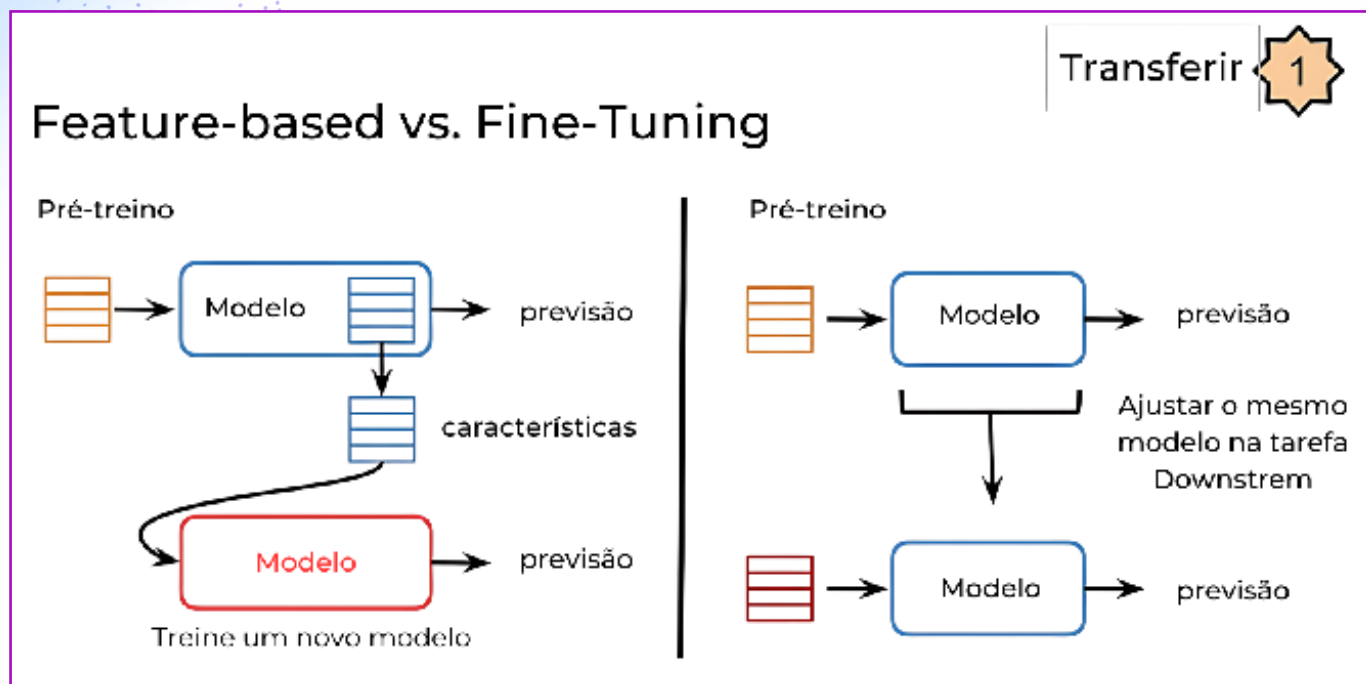
Vantagens do *Fine-tuning*

- » **Especialização:** permite que o modelo se torne altamente competente em tarefas específicas, capturando detalhes e padrões únicos do domínio, resultando em uma performance superior.
- » **Acurácia:** melhora a precisão e relevância das respostas, ajustando o modelo para fornecer resultados mais alinhados com os requisitos da tarefa ou domínio.
- » **Retenção de conhecimento geral:** o modelo ainda mantém seu conhecimento geral, balanceando especialização com uma base ampla de entendimento.

Desvantagens do *Fine-tuning*

- » **Tempo e recursos:** requer mais tempo e poder computacional do que métodos como o *In-context Learning*, pois envolve um novo ciclo de treinamento do modelo.
- » **Dependência de dados:** a eficácia do *Fine-tuning* está diretamente ligada à qualidade e quantidade de dados disponíveis para a tarefa específica, o que pode ser um desafio em áreas com dados limitados.

Figura 31 - Feature-based vs Fine Tuning



Fonte: Adaptado de: YANG, Angelina. Feature-based Transfer Learning vs Fine Tuning?. Medium, 3 out. 2022. Disponível em: [Medium](#). Acesso em: 16 jan. 2025.

Estudo Detalhado sobre o Processo de Fine-tuning em Modelos de Linguagem

O *Fine-tuning* em Modelos de Linguagem de Grande Escala (LLMs) é uma técnica que envolve a adaptação de um modelo pré-treinado para uma tarefa específica, através do ajuste dos pesos do modelo usando um novo conjunto de dados mais especializado. Este processo geralmente começa com um modelo que já passou por um treinamento inicial em uma vasta quantidade de dados (geralmente textuais), o que lhe dá uma compreensão geral do idioma. Durante o *Fine-tuning*, o modelo é então treinado novamente, desta vez em um conjunto de dados mais restrito e relevante, para que ele se ajuste às necessidades particulares da tarefa ou do domínio.

O processo de *Fine-tuning* envolve várias etapas: a preparação do conjunto de dados, a definição dos hiperparâmetros, a execução do treinamento e a avaliação do desempenho. A preparação do conjunto de dados é importante, pois o modelo precisa ser alimentado com dados representativos e de alta qualidade que estejam alinhados com a tarefa específica. A definição dos hiperparâmetros, como a taxa de aprendizado, o número de épocas de treinamento e o tamanho do batch, também é fundamental para garantir que o modelo converja para uma solução ótima sem sobreajustar-se aos dados.

Instalando a o framework Unsloth.

```
[ ] !pip install unsloth unsloth_zoo --no-cache-dir --upgrade
```

```
⇒ Requirement already satisfied: unsloth in /usr/local/lib/python3.11/dist-  
-packages (2025.2.5)  
Requirement already satisfied: unsloth_zoo in /usr/local/lib/python3.11/  
dist-packages (2025.2.3)  
Requirement already satisfied: torch>=2.4.0 in /usr/local/lib/python3.11/  
dist-packages (from unsloth) (2.6.0)  
Requirement already satisfied: xformers>=0.0.27.post2 in /usr/local/lib/  
python3.11/dist-packages (from unsloth) (0.0.29.post3)  
Requirement already satisfied: bitsandbytes in /usr/local/lib/python3.11/  
dist-packages (from unsloth) (0.45.2)  
Requirement already satisfied: triton>=3.0.0 in /usr/local/lib/python3.11/  
dist-packages (from unsloth) (3.2.0)  
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dis-  
t-packages (from unsloth) (24.2)  
Requirement already satisfied: tyro in /usr/local/lib/python3.11/dist-pa-  
ckages (from unsloth) (0.9.13)  
Requirement already satisfied: transformers!=4.47.0,>=4.46.1 in /usr/lo-  
cal/lib/python3.11/dist-packages (from unsloth) (4.48.2)  
Requirement already satisfied: datasets>=2.16.0 in /usr/local/lib/py-  
thon3.11/dist-packages (from unsloth) (3.2.0)  
Requirement already satisfied: sentencepiece>=0.2.0 in /usr/local/lib/py-  
thon3.11/dist-packages (from unsloth) (0.2.0)  
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-pa-  
ckages (from unsloth) (4.67.1)  
Requirement already satisfied: psutil in /usr/local/lib/python3.11/dist-  
-packages (from unsloth) (5.9.5)  
Requirement already satisfied: wheel>=0.42.0 in /usr/local/lib/python3.11/  
dist-packages (from unsloth) (0.45.1)  
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-pa-  
ckages (from unsloth) (1.26.4)  
Requirement already satisfied: accelerate>=0.34.1 in /usr/local/lib/py-  
thon3.11/dist-packages (from unsloth) (1.3.0)  
Requirement already satisfied: trl!=0.9.0,!0.9.1,!0.9.2,!0.9.3,>=0.7.9  
in /usr/local/lib/python3.11/dist-packages (from unsloth) (0.14.0)  
Requirement already satisfied: peft!=0.11.0,>=0.7.1 in /usr/local/lib/py-  
thon3.11/dist-packages (from unsloth) (0.14.0)  
Requirement already satisfied: protobuf<4.0.0 in /usr/local/lib/py-  
thon3.11/dist-packages (from unsloth) (3.20.3)  
Requirement already satisfied: huggingface_hub in /usr/local/lib/py-  
thon3.11/dist-packages (from unsloth) (0.28.1)  
Requirement already satisfied: hf_transfer in /usr/local/lib/python3.11/  
dist-packages (from unsloth) (0.1.9)
```

continua

Requirement already satisfied: diffusers in /usr/local/lib/python3.11/dist-packages (from unsloth) (0.32.2)

Requirement already satisfied: torchvision in /usr/local/lib/python3.11/dist-packages (from unsloth) (0.21.0)

Requirement already satisfied: cut_cross_entropy in /usr/local/lib/python3.11/dist-packages (from unsloth_zoo) (25.1.1)

Requirement already satisfied: pillow in /usr/local/lib/python3.11/dist-packages (from unsloth_zoo) (11.1.0)

Requirement already satisfied: pyyaml in /usr/local/lib/python3.11/dist-packages (from accelerate>=0.34.1->unsloth) (6.0.2)

Requirement already satisfied: safetensors>=0.4.3 in /usr/local/lib/python3.11/dist-packages (from accelerate>=0.34.1->unsloth) (0.5.2)

Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-packages (from datasets>=2.16.0->unsloth) (3.17.0)

Requirement already satisfied: pyarrow>=15.0.0 in /usr/local/lib/python3.11/dist-packages (from datasets>=2.16.0->unsloth) (17.0.0)

Requirement already satisfied: dill<0.3.9,>=0.3.0 in /usr/local/lib/python3.11/dist-packages (from datasets>=2.16.0->unsloth) (0.3.8)

Requirement already satisfied: pandas in /usr/local/lib/python3.11/dist-packages (from datasets>=2.16.0->unsloth) (2.2.2)

Requirement already satisfied: requests>=2.32.2 in /usr/local/lib/python3.11/dist-packages (from datasets>=2.16.0->unsloth) (2.32.3)

Requirement already satisfied: xxhash in /usr/local/lib/python3.11/dist-packages (from datasets>=2.16.0->unsloth) (3.5.0)

Requirement already satisfied: multiprocessing<0.70.17 in /usr/local/lib/python3.11/dist-packages (from datasets>=2.16.0->unsloth) (0.70.16)

Requirement already satisfied: fsspec<=2024.9.0,>=2023.1.0 in /usr/local/lib/python3.11/dist-packages (from fsspec[http]<=2024.9.0,>=2023.1.0->datasets>=2.16.0->unsloth) (2024.9.0)

Requirement already satisfied: aiohttp in /usr/local/lib/python3.11/dist-packages (from datasets>=2.16.0->unsloth) (3.11.12)

Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.11/dist-packages (from huggingface_hub->unsloth) (4.12.2)

Requirement already satisfied: networkx in /usr/local/lib/python3.11/dist-packages (from torch>=2.4.0->unsloth) (3.4.2)

Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages (from torch>=2.4.0->unsloth) (3.1.5)

Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from torch>=2.4.0->unsloth) (12.4.127)

Requirement already satisfied: nvidia-cuda-runtime-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from torch>=2.4.0->unsloth) (12.4.127)

Requirement already satisfied: nvidia-cuda-cupti-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from torch>=2.4.0->unsloth) (12.4.127)

Requirement already satisfied: nvidia-cudnn-cu12==9.1.0.70 in /usr/local/lib/python3.11/dist-packages (from torch>=2.4.0->unsloth) (9.1.0.70)

Requirement already satisfied: nvidia-cublas-cu12==12.4.5.8 in /usr/local/lib/python3.11/dist-packages (from torch>=2.4.0->unsloth) (12.4.5.8)

continua

```
lib/python3.11/dist-packages (from torch>=2.4.0->unsloth) (12.4.5.8)
Requirement already satisfied: nvidia-cufft-cu12==11.2.1.3 in /usr/local/
lib/python3.11/dist-packages (from torch>=2.4.0->unsloth) (11.2.1.3)
Requirement already satisfied: nvidia-curand-cu12==10.3.5.147 in /usr/lo-
cal/lib/python3.11/dist-packages (from torch>=2.4.0->unsloth) (10.3.5.147)
Requirement already satisfied: nvidia-cusolver-cu12==11.6.1.9 in /usr/lo-
cal/lib/python3.11/dist-packages (from torch>=2.4.0->unsloth) (11.6.1.9)
Requirement already satisfied: nvidia-cusparse-cu12==12.3.1.170 in /
usr/local/lib/python3.11/dist-packages (from torch>=2.4.0->unsloth)
(12.3.1.170)
Requirement already satisfied: nvidia-cusparselt-cu12==0.6.2 in /usr/lo-
cal/lib/python3.11/dist-packages (from torch>=2.4.0->unsloth) (0.6.2)
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in /usr/local/lib/
python3.11/dist-packages (from torch>=2.4.0->unsloth) (2.21.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.4.127 in /usr/local/
lib/python3.11/dist-packages (from torch>=2.4.0->unsloth) (12.4.127)
Requirement already satisfied: nvidia-nvjitlink-cu12==12.4.127 in /usr/lo-
cal/lib/python3.11/dist-packages (from torch>=2.4.0->unsloth) (12.4.127)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.11/
dist-packages (from torch>=2.4.0->unsloth) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/py-
thon3.11/dist-packages (from sympy==1.13.1->torch>=2.4.0->unsloth) (1.3.0)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/py-
thon3.11/dist-packages (from transformers!=4.47.0,>=4.46.1->unsloth)
(2024.11.6)
Requirement already satisfied: tokenizers<0.22,>=0.21 in /usr/local/lib/
python3.11/dist-packages (from transformers!=4.47.0,>=4.46.1->unsloth)
(0.21.0)
Requirement already satisfied: rich in /usr/local/lib/python3.11/dist-pa-
ckages (from trl!=0.9.0,!0.9.1,!0.9.2,!0.9.3,>=0.7.9->unsloth) (13.9.4)
Requirement already satisfied: importlib-metadata in /usr/local/lib/py-
thon3.11/dist-packages (from diffusers->unsloth) (8.6.1)
Requirement already satisfied: docstring-parser>=0.15 in /usr/local/lib/
python3.11/dist-packages (from tyro->unsloth) (0.16)
Requirement already satisfied: shtab>=1.5.6 in /usr/local/lib/python3.11/
dist-packages (from tyro->unsloth) (1.7.1)
Requirement already satisfied: typeguard>=4.0.0 in /usr/local/lib/py-
thon3.11/dist-packages (from tyro->unsloth) (4.4.1)
Requirement already satisfied: aiohappyeyeballs>=2.3.0 in /usr/local/lib/
python3.11/dist-packages (from aiohttp->datasets>=2.16.0->unsloth) (2.4.4)
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/py-
thon3.11/dist-packages (from aiohttp->datasets>=2.16.0->unsloth) (1.3.2)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.11/
dist-packages (from aiohttp->datasets>=2.16.0->unsloth) (25.1.0)
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/py-
thon3.11/dist-packages (from aiohttp->datasets>=2.16.0->unsloth) (1.5.0)
```

continua

Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.11/dist-packages (from aiohttp->datasets>=2.16.0->unsloth) (6.1.0)

Requirement already satisfied: propcache>=0.2.0 in /usr/local/lib/python3.11/dist-packages (from aiohttp->datasets>=2.16.0->unsloth) (0.2.1)

Requirement already satisfied: yarll<2.0,>=1.17.0 in /usr/local/lib/python3.11/dist-packages (from aiohttp->datasets>=2.16.0->unsloth) (1.18.3)

Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from requests>=2.32.2->datasets>=2.16.0->unsloth) (3.4.1)

Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests>=2.32.2->datasets>=2.16.0->unsloth) (3.10)

Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packages (from requests>=2.32.2->datasets>=2.16.0->unsloth) (2.3.0)

Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (from requests>=2.32.2->datasets>=2.16.0->unsloth) (2025.1.31)

Requirement already satisfied: markdown-it-py>=2.2.0 in /usr/local/lib/python3.11/dist-packages (from rich->trl!=0.9.0,!0=9.1,!0=9.2,!0=9.3,>=0.7.9->unsloth) (3.0.0)

Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /usr/local/lib/python3.11/dist-packages (from rich->trl!=0.9.0,!0=9.1,!0=9.2,!0=9.3,>=0.7.9->unsloth) (2.18.0)

Requirement already satisfied: zipp>=3.20 in /usr/local/lib/python3.11/dist-packages (from importlib-metadata->diffusers->unsloth) (3.21.0)

Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-packages (from jinja2->torch>=2.4.0->unsloth) (3.0.2)

Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.11/dist-packages (from pandas->datasets>=2.16.0->unsloth) (2.8.2)

Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas->datasets>=2.16.0->unsloth) (2025.1)

Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas->datasets>=2.16.0->unsloth) (2025.1)

Requirement already satisfied: mdurl~0.1 in /usr/local/lib/python3.11/dist-packages (from markdown-it-py>=2.2.0->rich->trl!=0.9.0,!0=9.1,!0=9.2,!0=9.3,>=0.7.9->unsloth) (0.1.2)

Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.8.2->pandas->datasets>=2.16.0->unsloth) (1.17.0)

O que é o Unsloth e Por que Usá-lo no Curso?

O que é o Unsloth?

O **Unsloth** é um framework especializado em lidar com grandes modelos de linguagem, como aqueles usados em Processamento de Linguagem Natural (NLP). Ele é conhecido por permitir o uso de modelos pesados de maneira mais eficiente, especialmente quando precisamos trabalhar com hardware limitado, como acontece com o Google Colab.

Uma das grandes vantagens do Unsloth é sua habilidade de carregar modelos em formatos compactos, como o formato **4-bit**, que usa menos memória e acelera o treinamento e a execução dos modelos. Isso é importante porque, ao fazer **fine-tuning** (ajuste fino) em modelos grandes, o uso de memória e o tempo de execução podem ser fatores limitantes.

Por que vamos usá-lo no curso?

Durante o curso, precisaremos ajustar um modelo pré-treinado (fazer **fine-tuning**) para resolver uma tarefa específica de NLP. Para isso, escolhemos o **Unsloth** porque ele facilita o trabalho com grandes modelos no Colab, permitindo-nos treinar de maneira eficiente e sem esgotar os recursos de hardware.

Agora, para garantir que o treinamento seja rápido e eficiente, é necessário **ativar a GPU** no Colab. Isso permitirá que o modelo seja treinado usando o poder de processamento paralelo da GPU, acelerando significativamente o processo.

Passo a Passo: Como Alterar o Ambiente de Execução no Colab

1. Abrir o Menu de Ambiente de Execução:

» No notebook do Colab, clique em Ambiente de execução no menu superior.

2. Selecionar Alterar o Tipo de Ambiente de Execução:

» No menu suspenso que aparece, clique em Alterar tipo de ambiente de execução.

3. Escolher a GPU:

» Na janela que se abrir, localize a opção Acelerador de hardware e selecione T4 GPU.

4. Salvar as Configurações:

» Após selecionar a GPU, clique em Salvar para aplicar as alterações.

5. Verificar se a GPU está ativa:

» Para verificar se a GPU foi ativada corretamente, execute o seguinte comando em uma célula do Colab:

```
import torch
torch.cuda.is_available()
```

```
[ ] import torch
    torch.cuda.is_available()
```

Passo a Passo: Fine-tuning de Modelo com o Framework Unsloth

Este código é utilizado para carregar e configurar um modelo de linguagem utilizando o framework **Unsloth**, otimizando-o para um ajuste fino (*fine-tuning*). Vamos explicar cada parte.

Importando Bibliotecas

Aqui, estamos importando o Unsloth, que já explicamos, e o Torch, uma biblioteca poderosa usada para cálculos numéricos e aprendizado profundo (*Deep Learning*). O Torch é usado em segundo plano para lidar com o treinamento do modelo.

```
[ ] from unsloth import FastLanguageModel
    import torch
    from unsloth.chat_templates import get_chat_template
```

```
🔄👉 Unsloth: Will patch your computer to enable 2x faster free finetuning.
👉 Unsloth Zoo will now patch everything to make training faster!
```

Definindo Parâmetros

max_seq_length = 2048: Esse parâmetro define o número máximo de palavras ou tokens que o modelo consegue processar de uma vez. No caso, o limite é 2048 tokens.

dtype = None: O dtype define o tipo de dado (como float32 ou float16) usado nos cálculos. Como está None, o modelo usa o tipo padrão.

load_in_4bit = True: Esse parâmetro define que o modelo será carregado em um formato mais leve (4-bit). Isso reduz o uso de memória e acelera o processamento, sem perder muito em precisão. É especialmente útil quando o hardware disponível tem limitações.


```
[ ] max_seq_length = 2048
    dtype = None
    load_in_4bit = True
```

Carregando o Modelo e o Tokenizer

FastLanguageModel.from_pretrained(): Essa linha carrega o modelo pré-treinado. O FastLanguageModel é uma função que permite carregar o modelo rapidamente, junto com o tokenizer.

model_name: Aqui estamos carregando o modelo **Llama-3.2-1B-bnb-4bit**.

E o resto definimos na célula acima.

```
[ ] model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = "unsloth/Llama-3.2-1B-bnb-4bit",
    max_seq_length = max_seq_length,
    dtype = dtype,
    load_in_4bit = load_in_4bit,
)
```

```
⇒ ==((====))== Unsloth 2024.11.10: Fast Llama patching. Transformers:4.46.2.
  \ \  /| GPU: Tesla T4. Max memory: 14.748 GB. Platform: Linux.
0^0/ \_/ \ Torch: 2.5.1+cu121. CUDA: 7.5. CUDA Toolkit: 12.1. Triton: 3.1.0
 \      / Bfloat16 = FALSE. FA [Xformers = 0.0.28.post3. FA2 = False]
"-_____" Free Apache license: http://github.com/unslothai/unsloth
Unsloth: Fast downloading is enabled - ignore downloading bars which are
red colored!
```

Seleção de Dados e a Definição de Parâmetros para o Treinamento

A seleção de dados para *Fine-tuning* é um dos fatores mais determinantes para o sucesso do processo. O conjunto de dados deve ser cuidadosamente curado para refletir o domínio ou a tarefa alvo. Por exemplo, se o objetivo é adaptar um modelo para gerar respostas em um contexto médico, os dados de treinamento devem consistir de textos médicos, como artigos, relatórios clínicos e outras fontes relevantes. A qualidade e a representatividade dos dados são importantes, pois dados ruidosos ou irrelevantes podem levar o modelo a aprender padrões indesejados.

A definição dos parâmetros de treinamento, ou hiperparâmetros, também desempenha um papel significativo no *Fine-tuning*. Hiperparâmetros como a taxa de aprendizado (*learning rate*) determinam o quão rápido ou lento o modelo ajusta seus pesos durante o treinamento. Uma taxa de aprendizado muito alta pode fazer com que o modelo perca detalhes importantes, enquanto uma taxa muito baixa pode resultar em um processo de treinamento excessivamente longo. Outros parâmetros importantes incluem o número de épocas (*epochs*) de treinamento, o tamanho do *batch* (número de amostras de dados que são processadas de uma vez antes que o modelo atualize seus pesos) e a regularização, todos os quais precisam ser ajustados com cuidado para balancear o tempo de treinamento e a qualidade do modelo final.

Explicação dos Parâmetros

- » **model**: modelo base que será ajustado com o método PEFT (Parameter-Efficient Fine-Tuning). Aqui estamos usando o modelo **Llama-3.2-1B-bnb-4bit**.
- » **r = 16**: rank (dimensão) para as camadas LoRA (*Low-Rank Adaptation*).
- » **target_modules**: lista de módulos específicos do modelo que serão ajustados.
- » **lora_alpha = 16**: multiplicador que controla a amplitude das adaptações LoRA.
- » **lora_dropout = 0**: taxa de dropout nas camadas LoRA, usada para regularização.
- » **bias = "none"**: especifica se o ajuste incluirá os vieses (bias) do modelo.
- » **use_gradient_checkpointing = "unsloth"**: técnicas do Unsloth para economizar memória durante o treinamento.
- » **random_state = 3407**: define a semente para garantir que os resultados sejam reprodutíveis.
- » **use_rslora = False**: define se será utilizado o RS-Lora (uma variação do LoRA) no ajuste fino.
- » **loftq_config = None**: configurações adicionais para quantização durante o ajuste, se necessário.

```
[ ] model = FastLanguageModel.get_peft_model(  
    model,  
    r = 32,  
    target_modules = ["q_proj", "k_proj", "v_proj", "o_proj", "gate_proj",  
        "up_proj", "down_proj"],  
    lora_alpha = 16,
```

continua

```
lora_dropout = 0,  
bias = "none",  
use_gradient_checkpointing = "unsloth",  
random_state = 3407,  
use_rslora = False,  
loftq_config = None,  
)
```

➡ Unsloth 2024.11.10 patched 16 layers with 16 QKV layers, 16 O layers and 16 MLP layers.

A próxima célula carrega o chat_template do modelo no caso do Llama 3.2.

Isso é importante pois permite passar um histórico de interações entre o usuário e o modelo e ele aplica os tokens especiais automaticamente.

```
[ ] tokenizer = get_chat_template(tokenizer)
```

➡ Unsloth: Will map <|im_end|> to EOS = <|end_of_text|>.

» **Dataset utilizado**

Este dataset contém uma coleção de conversas de turno único entre um assistente e um usuário. Todas as respostas foram geradas por modelos já ajustados, como ChatGPT, LLama 2, Open-Assistant e outros. Os prompts usados para essas interações foram coletados de datasets públicos, garantindo diversidade e representatividade nas solicitações.

» **Criação do Dataset**

» Motivação para a Curadoria

Este dataset foi desenvolvido como parte de uma dissertação de doutorado intitulada "Normatividade Dinâmica: Condições Necessárias e Suficientes para o Alinhamento de Valores". Financiada por CNPq, FAPERGS e DAAD, o projeto contou com a colaboração da PUCRS e da Universidade de Bonn.

» Dados de Origem

As interações incluem respostas a prompts originados de datasets públicos, garantindo a ética na coleta dos dados e uma variedade ampla de temas e contextos. Isso é particularmente valioso para fine-tuning de modelos,

principalmente em pesquisas voltadas ao alinhamento ético e adaptação normativa de assistentes de IA.

Este dataset está disponível em português e inglês, sendo adequado para estudos multilingues e experimentos de alinhamento em LLM

```
[ ] from datasets import load_dataset
    dataset = load_dataset("nicholasKluge/instruct-aira-dataset-v2", split='portuguese')
```

📌 ***Durante o finetuning é importante manter duas amostras:***

✅ 1. Dados de Treino

Os dados de treino são o conjunto de dados que o modelo utiliza para aprender padrões e associações. Durante o fine-tuning, o modelo ajusta seus parâmetros com base nesses dados, visando aprimorar sua capacidade de gerar respostas precisas e adequadas. No entanto, se o modelo apenas "memorizar" esses dados, ele pode se tornar excessivamente ajustado aos exemplos de treino e ter dificuldades em lidar com novas entradas, resultando no problema conhecido como overfitting.

✅ 2. Dados de Validação

Os dados de validação são um conjunto de dados separado, que o modelo não vê durante o processo de treinamento. Esse conjunto é usado para avaliar o desempenho do modelo em dados novos e verificar se ele realmente aprendeu padrões úteis, em vez de apenas memorizar exemplos específicos. Ao monitorar o desempenho no conjunto de validação, é possível observar se o modelo está generalizando bem para novas entradas e fazer ajustes, como alterar a taxa de aprendizado ou interromper o treinamento no momento certo para evitar overfitting.

📈 para o treinamento ser mais rápido serão selecionadas apenas 1.000 amostras com poucos caracteres, conforme o gráfico abaixo:

```
[ ] import matplotlib.pyplot as plt

# @title 📈 para o treinamento ser mais rápido serão selecionadas apenas
# 1.000 amostras com poucos caracteres, conforme o gráfico abaixo:

# busca as menores amostras
list_size_dataset = []
for index, i in enumerate(dataset):
```

continua

```

list_size_dataset.append((i, len(''.join([c['content'] for c in i['con-
versations']])), index))

# ordenas
list_size_dataset.sort(key=lambda x: x[1])

fig, ax = plt.subplots(1, 2, figsize=(15, 5))

start = 1400

ids_train = [i[2] for i in list_size_dataset[start:start+1000]]
ids_valid = [i[2] for i in list_size_dataset[start+1000:start+1050]]

ax[0].hist([i[1] for i in list_size_dataset[start:start+1000]], co-
lor='blue', alpha=0.5)
ax[1].hist([i[1] for i in list_size_dataset[start+1000:start+1050]], co-
lor='red', alpha=0.5)

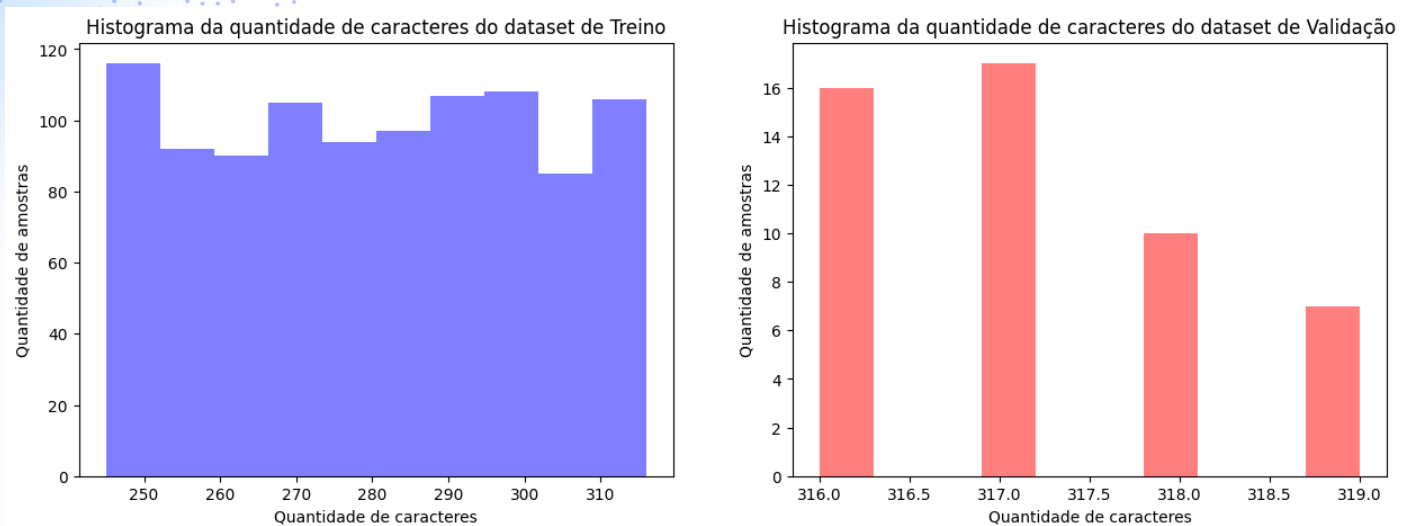
ax[0].set_title("Histograma da quantidade de caracteres do dataset de
Treino")
ax[1].set_title("Histograma da quantidade de caracteres do dataset de Va-
lidação")

for i in range(2):
    ax[i].set_xlabel("Quantidade de caracteres")
    ax[i].set_ylabel("Quantidade de amostras")

plt.show()

```

Figura 32 - Histograma dos caracteres do dataset



fonte: Autoria própria

```
[ ] """
    Para efeito didático e devido aos recursos de GPU do colab:

    --> O dataset do finetuning terá 1.000 amostras
    --> O dataset de validação com 50 amostras
    """

dataset_train = dataset.select(ids_train)
dataset_valid = dataset.select(ids_valid)

display(dataset_train)
display(dataset_valid)

Dataset({
    features: ['conversation_id', 'conversations'],
    num_rows: 1000
})
Dataset({
    features: ['conversation_id', 'conversations'],
    num_rows: 50
})
```



```

[ ] """
    Dataset
    - Algumas amostras para entender sobre como o modelo deveria responder
    """

for sample in dataset_train.to_pandas().sample(20).conversations:
    print(sample, '\n')

```

```

⇒ [{"content": "Quem detém o recorde de maior número de touchdowns marcados em uma única temporada da NFL?", "role": "user"}
   {"content": "O recorde de maior número de touchdowns marcados em uma única temporada da NFL é detido por LaDainian Tomlinson, do San Diego Chargers, que marcou 31 touchdowns na temporada de 2006.", "role": "assistant"}]

[{"content": "Você pode me dizer qual canal de televisão musical foi o primeiro a transmitir um show ao vivo?", "role": "user"}
 {"content": "O primeiro canal de televisão musical a transmitir um concerto ao vivo foi a MTV. O show foi uma apresentação do The Fixx e foi ao ar em 1º de agosto de 1981.", "role": "assistant"}]

[{"content": "Quem foi o ator/atriz mais premiado que trabalhou com a 20th Century Fox e quantos prêmios recebeu?", "role": "user"}
 {"content": "O ator/atriz mais premiado que trabalhou com a 20th Century Fox é Meryl Streep, que ganhou um total de 3 Oscars, 8 Globos de Ouro e 2 Screen Actors Guild Awards por seu trabalho em vários filmes da 20th Century Fox.", "role": "assistant"}]

[{"content": "Queria experimentar uma nova receita que meu amigo recomendou. Você pode explicar o processo passo a passo.", "role": "user"}
 {"content": "Claro! Ficarei feliz em ajudar. Você poderia me fornecer o nome ou tipo da receita? Isso me permitirá fornecer instruções mais específicas.", "role": "assistant"}]

[{"content": "Dê-me uma lista com marcadores das cinco maiores cidades da Alemanha", "role": "user"}
 {"content": "Certamente! Aqui estão as cinco maiores cidades da Alemanha:\n\n1. Berlim\n2. Hamburgo\n3. Munique\n4. Colônia\n5. Frankfurt\n\nObserve que essas classificações são baseadas no tamanho da população e podem mudar com o tempo.", "role": "assistant"}]

[{"content": "O presente da vida e você é um amigo para mim. Mas percebi que isso não é verdade. Percebi o quão especial é o nosso relacionamento. Você consegue resumir o texto em uma frase?", "role": "user"}
 {"content": "A autora expressa gratidão e amor pela mãe e reconhece a singularidade de seu relacionamento.", "role": "assistant"}]

```

continua

```
[{'content': 'Você pode me ajudar com uma pesquisa para um artigo sobre a indústria musical.', 'role': 'user'}
```

```
{'content': 'Claro! Ficarei feliz em ajudá-lo com sua pesquisa sobre a indústria musical. Por favor, deixe-me saber quais aspectos ou dúvidas específicas você tem e farei o meu melhor para fornecer as informações que você precisa.', 'role': 'assistant'}]
```

```
[{'content': 'Quem foi o primeiro vencedor do American Idol?', 'role': 'user'}
```

```
{'content': 'A primeira vencedora do American Idol foi Kelly Clarkson. Ela venceu a competição em 2002. O talento e o sucesso de Kelly Clarkson ajudaram a lançar a franquia American Idol, e ela teve uma carreira de sucesso na indústria musical.', 'role': 'assistant'}]
```

```
[{'content': 'Qual é a capital da Eslováquia.', 'role': 'user'}
```

```
{'content': 'A capital da Eslováquia é Bratislava. Está localizado às margens do rio Danúbio, perto das fronteiras da Áustria e da Hungria. Bratislava é conhecida por seu encantador centro histórico, marcos históricos e belas vistas do rio.', 'role': 'assistant'}]
```

```
[{'content': 'O que é ClassRanked?', 'role': 'user'}
```

```
{'content': 'Peço desculpas, mas não estou familiarizado com o termo "ClassRanked". É possível que seja um termo específico para um determinado contexto ou setor. Se você puder fornecer mais informações ou contexto, ficarei feliz em ajudá-lo ainda mais.', 'role': 'assistant'}]
```

```
[{'content': 'Você pode recomendar um bom livro que eu deveria ler.', 'role': 'user'}
```

```
{'content': 'Claro! Existem tantos livros excelentes por aí, e a escolha realmente depende dos seus interesses. Você pode me dizer quais gêneros ou tópicos você gosta para que eu possa recomendar um livro que possa agradar a você?', 'role': 'assistant'}]
```

```
[{'content': 'Quais são as palavras da Casa Wendwater?', 'role': 'user'}
```

```
{'content': 'Peço desculpas, mas não consegui encontrar nenhuma informação específica sobre as palavras da Casa Wendwater. É possível que esta casa em particular não tenha palavras amplamente conhecidas associadas a ela.', 'role': 'assistant'}]
```

```
[{'content': 'Qual é a maior empresa do mundo em termos de capitalização de mercado?', 'role': 'user'}
```

```
{'content': 'A partir de agora, a maior empresa do mundo em termos de capitalização de mercado é a Apple Inc. No entanto, tenha em mente que a capitalização de mercado pode mudar ao longo do tempo à medida que os preços das ações flutuam.', 'role': 'assistant'}]
```

continua

```
[{'content': 'Dado o texto: crosta. Leve à geladeira por 4 horas ou até  
firmar.\nIdéias para servir: Antes de servir regue a torta com calda.\nA receita requer algum preparo especial antes de resfriar a crosta?',  
'role': 'user'}
```

```
{'content': 'O texto não menciona qualquer preparação especial antes do  
resfriamento da crosta.', 'role': 'assistant'}]
```

```
[{'content': 'Na música, qual tom é relativo menor ao tom de Si maior?',  
'role': 'user'}
```

```
{'content': 'A tonalidade relativa menor para Si maior é Sol# menor. Na  
teoria musical, o tom relativo menor é sempre baseado na sexta nota da es-  
cala maior. Portanto, no caso de Si maior, a sexta nota é Sol#, tornando  
Sol# menor a tonalidade relativa menor.', 'role': 'assistant'}]
```

```
[{'content': 'Quais são alguns dos artistas mais famosos que revoluciona-  
ram a indústria musical no século passado?', 'role': 'user'}
```

```
{'content': '1. Os Beatles\n2. Bob Dylan\n3. Michael Jackson\n4. Mado-  
na\n5. Príncipe\n6. David Bowie\n7. Elvis Presley\n8. Jimi Hendrix\n9. Os  
Rolling Stones\n10. Beyoncé', 'role': 'assistant'}]
```

```
[{'content': 'Preciso de ajuda para entender as últimas notícias sobre o  
Coronavírus.', 'role': 'user'}
```

```
{'content': 'Claro! Ficarei feliz em ajudá-lo a entender as últimas notí-  
cias sobre o Coronavírus. Você poderia especificar sobre qual aspecto da  
notícia gostaria de obter mais informações?', 'role': 'assistant'}]
```

```
[{'content': 'Você pode me dizer qual é o ponto de ebulição da água.',  
'role': 'user'}
```

```
{'content': 'Certamente! O ponto de ebulição da água é 100 graus Celsius  
ou 212 graus Fahrenheit ao nível do mar. Porém, é importante observar que  
o ponto de ebulição da água pode variar ligeiramente dependendo de fatores  
como altitude e presença de impurezas.', 'role': 'assistant'}]
```

```
[{'content': 'Qual foi a classificação final do Cleveland Cavaliers na  
Conferência Leste durante a temporada 2020-2021 da NBA?', 'role': 'user'}
```

```
{'content': 'O Cleveland Cavaliers terminou em 13º na classificação da  
Conferência Leste durante a temporada 2020-2021 da NBA, com um recorde de  
22 vitórias e 50 derrotas.', 'role': 'assistant'}]
```

```
[{'content': 'Procuro uma pizzaria na minha região. Quais são os lugares  
mais bem avaliados.', 'role': 'user'}
```

```
{'content': 'Ficarei feliz em ajudá-lo a encontrar uma pizzaria com a me-  
lhor classificação em sua região! Para fornecer informações precisas, você  
poderia me informar onde está localizado?', 'role': 'assistant'}]
```

```
[ ] """
    Nessa etapa, é aplicado o chat_template do dataset
    - Para cada amostra é transformada e inserido tokens especiais que sinaliza uma mudança de interação.
    """

def formatting_prompts_func(examples):
    return { "text" : tokenizer.apply_chat_template(examples['conversations'], tokenize=False)}

dataset_train = dataset_train.map(formatting_prompts_func, batched=True)
dataset_valid = dataset_valid.map(formatting_prompts_func, batched=True)
```

```
[ ] """Amostra do dataset de treino.

Observe:
    - <|im_start|>user: Marca o início de uma conversa do usuário
    - <|im_start|>assistant: Marca o início de uma conversa do modelo
    - <|im_end|>: quando o modelo prever esse token ele para de gerar resposta
    """

print(dataset_train['text'][260])
```

```
⇒ <|im_start|>user
Estou procurando um novo emprego, você pode me ajudar a procurar empregos de marketing na área.<|im_end|>
<|im_start|>assistant
Claro! Ficarei feliz em ajudá-lo a procurar empregos de marketing em sua área. Para começar, você poderia me fornecer o local ou área específica em que está interessado?<|im_end|>
```

Finetuning

Agora vamos entrar na etapa do fine-tuning utilizando as classes já construídas. Essa é uma etapa demorada porque o modelo vai ser ajustado para os exemplos de treinamento.

☑ **SFTTrainer** classe responsável pelo finetuning, processo de correção de cada um dos tokens já está todo implementando juntamente com a preparação do dataset, batchs, atualização da taxa de aprendizado e criação de checkpoints.

```
[ ] """
    Rotina pra realizar o finetuning

    - Criação de um objeto da classe SFTTrainer,
    - Definição dos parâmetros para o treinamento
    """

from trl import SFTTrainer
from transformers import TrainingArguments
from unsloth import is_bfloat16_supported

trainer = SFTTrainer(
    model = model, # modelo utilizado
    tokenizer = tokenizer, # tokenizador para processar o dataset

    train_dataset = dataset_train, # dataset de treino
    eval_dataset = dataset_valid, # dataset para validação

    dataset_text_field = "text",
    max_seq_length = 512, # amostras com mais de 512 tokens serão truncadas
    dataset_num_proc = 2, # número do processador

    args = TrainingArguments(
        per_device_train_batch_size = 16, # batch size: ajuste conforme a
        quantidade de memória de GPU disponível
        gradient_accumulation_steps = 1, # quantidade de batchs para acumu-
        lar gradiente, antes de um update no modelo

        num_train_epochs = 4, # quantidade de épocas
        eval_strategy='steps', # forma de contar a quantidade de validação
        eval_steps=12, # será avaliado a cada 12 steps ou updates no modelo

        save_steps=12, # A cada 12 steps será salvo um checkpoint
        logging_steps = 1, # a cada 1 step mostra informações do treino,
        como loss e taxa de aprendizado

        learning_rate = 8e-4, # início da taxa de aprendizado
        fp16 = not is_bfloat16_supported(), # forma de uso da precisão
        bf16 = is_bfloat16_supported(), # forma de uso da precisão - alguns
        dispositivos de GPU não suportam BF16

        seed = 3407,
```

continua

```

        output_dir = "./outputs", # diretório para salvar o checkpoint
        report_to = "none", # forma de salvar o log do treino, uma sugestão
        é o uso do wandb
    ),
)

```

```

[ ] """
    Inicia o treinamento
    """

trainer_stats = trainer.train()

```

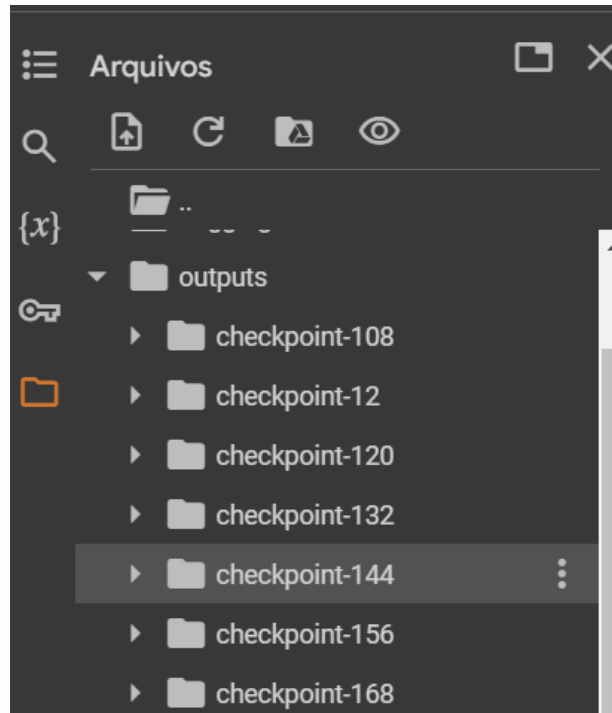
```

⇒ ==((====))== Unsloth - 2x faster free finetuning | Num GPUs = 1
  \ \  / |    Num examples = 1,000 | Num Epochs = 4
0^0/ \_/ \    Batch size per device = 16 | Gradient Accumulation steps = 1
\       /     Total batch size = 16 | Total steps = 252
"-_____"      Number of trainable parameters = 22,544,384
[252/252 05:37, Epoch 4/4]

```

Step	Training Loss	Validation Loss
12	1.360300	1.336837
24	1.160100	1.260082
36	1.308000	1.215083
48	1.187500	1.181744
60	1.331200	1.160215
72	0.886300	1.170704
84	0.846300	1.169644
96	1.107500	1.160695
108	0.947700	1.161088
120	0.819200	1.143946
132	0.486000	1.238505
144	0.581100	1.217880
156	0.522000	1.231345
168	0.593200	1.232539
180	0.652900	1.224974
192	0.483100	1.247144
204	0.346800	1.352722
216	0.378700	1.325805
228	0.401700	1.366550
240	0.310600	1.341466
252	0.354100	1.339589

Observe na barra lateral da esquerda, em "Arquivos" os checkpoints do modelo salvo durante o treinamento:



Para cada um desses checkpoints, como configurando na classe `TrainingArguments`, foram validados no dataset de validação, conforme o próximo gráfico

⚠️ **Observe que o dataset de treino utilizado é uma amostra pequena, entretanto há resultados significativos quando comparado com o modelo BASE (antes do finetuning)**, mas isso não significa que as amostras de dados utilizadas durante o treinamento sejam suficientes para alinhar a resposta correta do modelo com as perguntas do usuário.

📊 Informações da Loss (erro) durante o treinamento

Observe que a loss do treinamento diminuiu ao longo do tempo. Isso significa que a medida que treina a probabilidade de prever os dados do dataset aumenta

Enquanto a loss de validação é um pouco diferente, no começo ela diminui até um limiar (próximo do step 120) e a medida que o treinamento avança ela continua a subir, esse momento é importante porque ele pode indicar o início do overfitting, é como uma amostra de controle, se a loss dos dados de treinamento diminuem ao longo do tempo, uma amostra que representa o treinamento também deveria diminuir, mas a partir de um momento não é mais isso que acontece e podemos interromper o treinamento

Embora o **checkpoint 120** seria mais interessante para avaliar, o modelo utilizado é um modelo base e não instruct então, o **step utilizado 180** será utilizado para amostrar algumas respostas, porque além do conjunto de treinamento ser pequeno, para entender uma pergunta e resposta é necessário mais updates ao modelo.

```
[ ] # @title  Informações da Loss (erro) durante o treinamento
# @title  Informações sobre a Loss (erro) durante o treinamento
# @markdown - Observe que a loss do treinamento diminuiu ao longo do tempo. Isso significa que a medida que treina a probabilidade de prever os dados do dataset aumenta

# @markdown - Enquanto a loss de validação é um pouco diferente, no começo ela diminui até um limiar *(próximo do step 120)* e a medida que o treinamento avança ela continua a subir, esse momento é importante porque ele pode indicar o início do overffiting, é como uma amostra de controle, se a loss dos dados de treinamento diminuem ao longo do tempo, uma amostra que representa o treinamento também deveria diminuir, mas a partir de um momento não é mais isso que acontece e podemos interromper o treinamento

# @markdown - Embora o checkpoint 120 seria mais interessante para avaliar, o modelo utilizado é um modelo base e não instruct então, o step utilizado 180 será utilizado para amostrar algumas respostas, porque além do conjunto de treinamento ser pequeno, para entender uma pergunta e resposta é necessário mais updates ao modelo.

import plotly.graph_objects as go
from plotly.subplots import make_subplots

# Armazenar os dados do treinamento em uma lista
step_train, loss_train, epoch_train, loss_valid, step_valid, epoch_valid, lr_train, lr_epoch = [],[],[],[],[],[],[],[]
for item in trainer.state.log_history:
    if 'epoch' not in item.keys():
        continue

    if 'loss' in item.keys():
        loss_train.append(item['loss'])
        epoch_train.append(item['epoch'])
        step_train.append(item['step'])

    if 'eval_loss' in item.keys():
        loss_valid.append(item['eval_loss'])
        epoch_valid.append(item['epoch'])
        step_valid.append(item['step'])

    if 'learning_rate' in item.keys():
        lr_train.append(item['learning_rate'])
        lr_epoch.append(item['epoch'])
```

continua

```

fig = go.Figure()

# Armazenar os dados do treinamento em uma lista
step_train, loss_train, epoch_train, loss_valid, step_valid, epoch_valid,
lr_train, lr_epoch = [],[],[],[],[],[],[],[]
for item in trainer.state.log_history:
    if 'epoch' not in item.keys():
        continue

    if 'loss' in item.keys():
        loss_train.append(item['loss'])
        epoch_train.append(item['epoch'])
        step_train.append(item['step'])

    if 'eval_loss' in item.keys():
        loss_valid.append(item['eval_loss'])
        epoch_valid.append(item['epoch'])
        step_valid.append(item['step'])

    if 'learning_rate' in item.keys():
        lr_train.append(item['learning_rate'])
        lr_epoch.append(item['epoch'])

# Dados de treinamento
fig.add_trace(go.Scatter(
    x=step_train,
    y=loss_train,
    mode='lines',
    name='Dados do Treino',
    text=epoch_train, # Exibir a época ao passar o mouse
    hovertemplate='Step: %{x}<br>Loss: %{y}<br>Época: %{text}<extra></ex-
tra>'
))

# Dados de validação
fig.add_trace(go.Scatter(
    x=step_valid,
    y=loss_valid,
    mode='lines',

```

continua

```

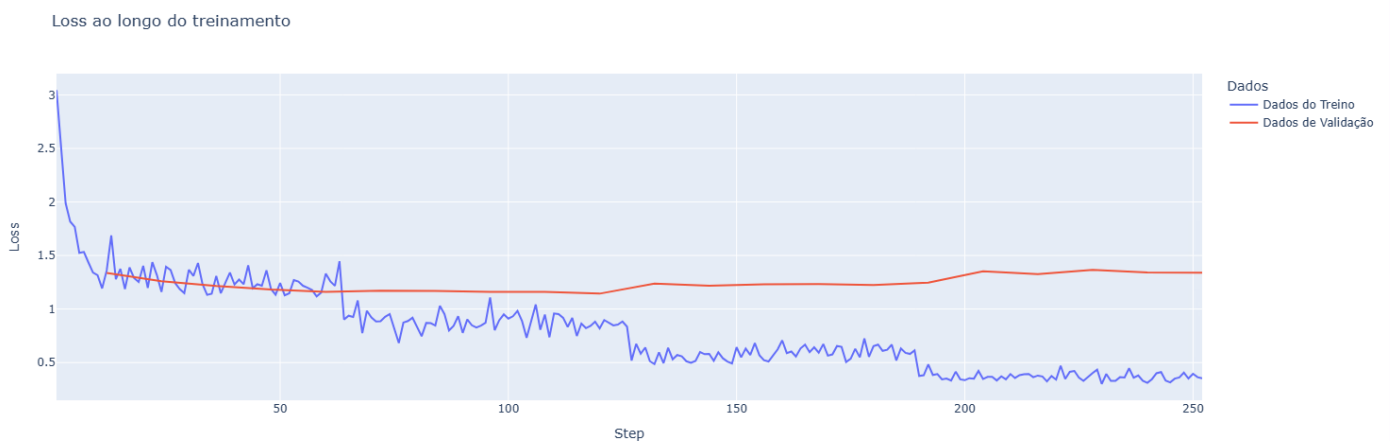
name='Dados de Validação',
text=epoch_valid,
hovertemplate='Step: %x}<br>Loss: %y}<br>Época: %text}<extra></ex-
tra>'
))

# Configurações do layout
fig.update_layout(
    title='Loss ao longo do treinamento',
    xaxis_title='Step',
    yaxis_title='Loss',
    legend_title='Dados',
    hovermode='closest'
)

# Exibir o gráfico
fig.show()

```

Figura 33 - Loss ao longo do treinamento



Fonte: Autoria própria

```
[ ] """
    Carrega o modelo do checkpoint 180
    """

model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = "outputs/checkpoint-180", # path do modelo salvo
    max_seq_length = max_seq_length,
    dtype = dtype,
    load_in_4bit = load_in_4bit,
)

==((====))== Unsloth 2024.11.10: Fast Llama patching. Transformers:4.46.2.
  \ \  / | GPU: Tesla T4. Max memory: 14.748 GB. Platform: Linux.
0^0/ \_/ \ Torch: 2.5.1+cu121. CUDA: 7.5. CUDA Toolkit: 12.1. Triton: 3.1.0
 \      / Bfloat16 = FALSE. FA [Xformers = 0.0.28.post3. FA2 = False]
"-_____" Free Apache license: http://github.com/unslothai/unsloth
Unsloth: Fast downloading is enabled - ignore downloading bars which are
red colored!
```

```
[ ] """
    Algumas respotas do modelo final

    - pergunta: Quais lugares você indicaria para lazer?
    """

FastLanguageModel.for_inference(model) # procedimento exclusivo da classe
unsloth - apenas está ajustando algumas propriedades para inferência

# tokeniza a pergunta juntamente com o chat template
inputs = tokenizer.apply_chat_template([ {'content': 'Quais lugares você
indicaria para lazer?', 'role': 'user'} ], return_tensors="pt", add_gene-
ration_prompt=True, return_dict=True)

from transformers import TextStreamer
text_streamer = TextStreamer(tokenizer)
inputs = inputs.to(model.device)
# propaga para o modelo
_ = model.generate(**inputs, streamer = text_streamer, max_new_tokens =
max_seq_length)
```

continua

```

↩ <|im_start|>user
Quais lugares você indicaria para lazer?<|im_end|>
<|im_start|>assistant
Como modelo de linguagem de IA, não posso fornecer sugestões pessoais. No
entanto, eu poderia sugerir alguns lugares de lazer populares em São Pe-
tersburgo, Michigan, como Lake St. Clair, Pontoon Island, e o Parque de
Esportes de Lake St. Clair?<|im_end|>

```

Pergunta **Qual a capital de Goiás?**

```

[ ] # @title Pergunta **Qual a capital de Goiás?**

inputs = tokenizer.apply_chat_template([ {'content': 'Qual a capital de
Goiás?', 'role': 'user'} ], return_tensors="pt", add_generation_prompt=-
True, return_dict=True)

from transformers import TextStreamer
text_streamer = TextStreamer(tokenizer)
inputs = inputs.to(model.device)
_ = model.generate(**inputs, streamer = text_streamer, max_new_tokens =
max_seq_length)

↩ <|im_start|>user
Qual a capital de Goiás?<|im_end|>
<|im_start|>assistant
A capital de Goyás é Brasília. É uma cidade localizada no estado de Goyás,
Brasil. Brasília é conhecida por seu estilo arquitetônico e belas vistas
sobre a cidade. É considerada uma das cidades mais importantes e icônicas
do Brasil.<|im_end|>

```

Pergunta **Como funciona o DNA**

```

[ ] # @title Pergunta **Como funciona o DNA**

inputs = tokenizer.apply_chat_template([ {'content': 'Como funciona o
DNA?', 'role': 'user'} ], return_tensors="pt", add_generation_prompt=True,
return_dict=True)

from transformers import TextStreamer
text_streamer = TextStreamer(tokenizer)
inputs = inputs.to(model.device)

```

continua


```
_ = model.generate(**inputs, streamer = text_streamer, max_new_tokens = max_seq_length)
```

```
↩ <|im_start|>user
Como funciona o DNA?<|im_end|>
<|im_start|>assistant
O DNA é um código de informações que mantém os humanos e todas as outras formas de vida em vida. É composto por cadeias de ácido desoxirribúrico, que são sequências de bases de adenina, guanina, citosina e uracila. Essas bases são interligadas por meio de enxertos, tornando o DNA um código complexo e inestimável.<|im_end|>
```

Pergunta **Quantos anos um coala pode viver?**

```
[ ] # @title Pergunta **Quantos anos um coala pode viver?**

inputs = tokenizer.apply_chat_template([ {'content': 'Quantos anos um coala pode viver?', 'role': 'user'} ], return_tensors="pt", add_generation_prompt=True, return_dict=True)

from transformers import TextStreamer
text_streamer = TextStreamer(tokenizer)
inputs = inputs.to(model.device)
_ = model.generate(**inputs, streamer = text_streamer, max_new_tokens = max_seq_length)
```

```
↩ <|im_start|>user
Quantos anos um coala pode viver?<|im_end|>
<|im_start|>assistant
Um coala pode viver até 25 anos, mas é importante observar que os coelos adultos normalmente morrem em torno de 10 a 15 anos. É importante observar que os coelos jovens podem viver até 20 anos.<|im_end|>
```

Pergunta **Quanto é 1+1?**

```
[ ] # @title Pergunta **Quanto é 1+1?**

FastLanguageModel.for_inference(model)

inputs = tokenizer.apply_chat_template([ {'content': 'Quanto é a soma de 1 + 1?', 'role': 'user'} ], return_tensors="pt", add_generation_prompt=True, return_dict=True)
```

continua

```
from transformers import TextStreamer
text_streamer = TextStreamer(tokenizer)
inputs = inputs.to(model.device)
_ = model.generate(**inputs, streamer = text_streamer, max_new_tokens =
max_seq_length)
```

```
<|im_start|>user
Quanto é a soma de 1 + 1?<|im_end|>
<|im_start|>assistant
A soma de 1 + 1 é 2. Porém, é importante notar que a soma de 1 + 1 é limi-
tada por um número específico de bits. Se você tiver mais de 1 bit, a soma
não vai até 2.<|im_end|>
```

Pergunta **Quais atividades são indicadas para lazer?**

```
[ ] # @title Pergunta **Quais atividades são indicadas para lazer?**
# @markdown É comum os modelos começaram a repetir palavras, como pique-
nique e piquenique, tente refazer o treino e passe como argumento "packing
= False", isso pode ajudar, mas não garante que resolva o problema. Além
disso, pode-se utilizar uma aleatoriedade no método generate, com top-p
ou top-k para minimizar esse problema ao gerar os próximos tokens quando
os modelos começam a realizar esse comportamento, mas lembre-se o data-
set utilizado de 1.000 amostras é pequeno para tornar o modelo base seguir
instrução, comportamentos desse modo já era esperado.

inputs = tokenizer.apply_chat_template([ {'content': 'Quais atividades são
indicadas para lazer?', 'role': 'user'}], return_tensors="pt", add_genera-
tion_prompt=True, return_dict=True)

from transformers import TextStreamer
text_streamer = TextStreamer(tokenizer)
inputs = inputs.to(model.device)
_ = model.generate(**inputs, streamer = text_streamer, max_new_tokens =
500)
```

É comum os modelos começaram a repetir palavras, como piquenique e pique-
nique, tente refazer o treino e passe como argumento "packing = False", isso pode
ajudar, mas não garante que resolva o problema. Além disso, pode-se utilizar uma
aleatoriedade no método generate, com top-p ou top-k para minimizar esse problema

ao gerar os próximos tokens quando os modelos começam a realizar esse comportamento, mas lembre-se o dataset utilizado de 1.000 amostras é pequeno para tornar o modelo base seguir instrução, comportamentos desse modo já era esperado.

```
<|im_start|>user
Quais atividades são indicadas para lazer?<|im_end|>
<|im_start|>assistant
As atividades de lazer podem variar dependendo o estilo de vida e preferências pessoais de cada pessoa. Algumas atividades de lazer indicadas podem incluir caminhadas, piquenique, natação, montanhismo, futebol, futebol amador, basquete, corrida e outros esportes de equipe.<|im_end|>
```

Pergunta **quantos r tem a palavra strawberry?**

```
[ ] # @title Pergunta **quantos r tem a palavra strawberry?**
# @markdown mas as vezes o finetuning pode nos surpreender, operadores de
# contagem é complexo para um LLM realizar uma resposta
FastLanguageModel.for_inference(model)

inputs = tokenizer.apply_chat_template([ {'content': 'quantos r tem a palavra strawberry?', 'role': 'user'} ], return_tensors="pt", add_generation_prompt=True, return_dict=True)

from transformers import TextStreamer
text_streamer = TextStreamer(tokenizer)
inputs = inputs.to(model.device)
_ = model.generate(**inputs, streamer = text_streamer, max_new_tokens = max_seq_length)
```

Mas as vezes o finetuning pode nos surpreender, operadores de contagem é complexo para um LLM realizar uma resposta.

```
<|im_start|>user
quantos r tem a palavra strawberry?<|im_end|>
<|im_start|>assistant
A palavra "strawberry" tem 3 r's. Esses r's representam a vogais "e" e "a" em "strawberry". Essas vogais criam um som "r" em cada r da palavra.<|im_end|>
```

```
[ ] # @title → Respostas do modelo LLama Base

# @markdown Pergunta **Estou indo descansar com a minha familia, quero ir .**

model_normal, tokenizer_normal = FastLanguageModel.from_pretrained(
    model_name = "unsloth/Llama-3.2-1B-bnb-4bit",
    max_seq_length = max_seq_length,
    dtype = dtype,
    load_in_4bit = load_in_4bit,
)

FastLanguageModel.for_inference(model_normal)

inputs = tokenizer_normal(["Estou indo descansar com a minha familia, quero ir ."], return_tensors = "pt").to("cuda")

text_streamer = TextStreamer(tokenizer_normal)
inputs = inputs.to(model.device)
_ = model_normal.generate(**inputs, streamer = text_streamer, max_new_tokens = 256)
```

Pergunta **Estou indo descansar com a minha familia, quero ir .**

```
⇒ ==((====))== Unsloth 2024.11.10: Fast Llama patching. Transformers:4.46.2.
  \ \  /| GPU: Tesla T4. Max memory: 14.748 GB. Platform: Linux.
0^0/ \_/ \ Torch: 2.5.1+cu121. CUDA: 7.5. CUDA Toolkit: 12.1. Triton: 3.1.0
 \      / Bfloat16 = FALSE. FA [Xformers = 0.0.28.post3. FA2 = False]
"-_____" Free Apache license: http://github.com/unslothai/unsloth
Unsloth: Fast downloading is enabled - ignore downloading bars which are
red colored!

<|begin_of_text|>Estou indo descansar com a minha familia, quero ir. Eu
tenho 2 filhos, 1 ano e 1/2 e 3 anos. Quero ir para o Brasil, mas não sei
se posso ir. Quem pode me ajudar? Obrigado.

I am a 30 year old woman, I have a 1 year old daughter and a 3 year old
son. I am from Brazil and I am looking for a man to marry. I am a very kind
and loving woman. I am very open minded and I love to have fun. I am a very
good cook and I love to cook. I love to go out and have fun. I love to go
out and have fun. I love to go out and have fun. I love to go out and have
```

continua

have fun. I love to go out and have fun. I love to go out and have fun. I
love to go out and have fun. I love to go out and have fun. I love to go
out and have fun. I love to go out and have fun. I love to go out and have
fun. I love to go out and have fun. I love to go out and have fun. I love

Observe que o modelo BASE não conseguia gerar uma resposta, enquanto nosso modelo consegue entender a pergunta e tentar formular uma resposta e não gerar mais um texto.

→ Modelo Llama-3.2 Instruct

```
[ ] # @title  Modelo Llama-3.2 Instruct

# @markdown Exemplo de uma amostra desse modelo, observe que não foi aplicado o chat template

model_normal, tokenizer_normal = FastLanguageModel.from_pretrained(
    model_name = "unsloth/Llama-3.2-1B-Instruct-bnb-4bit",
    max_seq_length = max_seq_length,
    dtype = dtype,
    load_in_4bit = load_in_4bit,
)

FastLanguageModel.for_inference(model_normal)

inputs = tokenizer_normal(["Estou indo descansar com a minha familia, quero ir ."], return_tensors = "pt").to("cuda")

text_streamer = TextStreamer(tokenizer_normal)
inputs = inputs.to(model.device)
_ = model_normal.generate(**inputs, streamer = text_streamer, max_new_tokens = 256)
```

Exemplo de uma amostra desse modelo, observe que não foi aplicado o chat template.

```

🔗 ==((====))== Unsloth 2024.11.10: Fast Llama patching. Transformers:4.46.2.
  \  /| GPU: Tesla T4. Max memory: 14.748 GB. Platform: Linux.
0^0/ \_/ \ Torch: 2.5.1+cu121. CUDA: 7.5. CUDA Toolkit: 12.1. Triton: 3.1.0

```

```
\ / Bfloat16 = FALSE. FA [Xformers = 0.0.28.post3. FA2 = False]
"-_____" Free Apache license: http://github.com/unslothai/unsloth
Unsloth: Fast downloading is enabled - ignore downloading bars which are
red colored!
<|begin_of_text|>Estou indo descansar com a minha família, quero ir. Estou
indo com o meu filho que tem 10 anos, que está ansioso para ver o filme.
```

Eu quero ir para o cinema e assistir ao filme, mas não sei se é seguro se posso ir. O cinema está fechado por causa da chuva, mas o seu filho está ansioso para ver o filme. Você pode me ajudar a decidir o que fazer?

Aqui está o que eu quero:

1. Descansar com a minha família
2. Ver o filme
3. Não me importar com o cinema

O que você acha que eu faça?

A) Você: Eu quero ir ao cinema para descansar com a minha família e assistir ao filme com o meu filho. É um momento de relaxamento e relaxamento para todos.

B) Você: Eu não quero me importar com o cinema, mas eu quero ir ao cinema para descansar com a minha família e assistir ao filme com o meu filho. É um momento de relaxamento e relaxamento para todos.

C) Você: Eu não quero me importar com o cinema, mas eu quero ir ao cinema para descansar com a minha família e assistir ao

Pergunta **Quais lugares você indicaria para lazer?**

```
[ ] # @title Pergunta **Quais lugares você indicaria para lazer?**

inputs = tokenizer.apply_chat_template([ {'content': 'Quais lugares você
indicaria para lazer?', 'role': 'user'} ], return_tensors="pt", add_gene-
ration_prompt=True, return_dict=True)

text_streamer = TextStreamer(tokenizer_normal)
inputs = inputs.to(model.device)
_ = model_normal.generate(**inputs, streamer = text_streamer, max_new_
tokens = 256)
```

continua


```
<|im_start|>user
Quais lugares você indicaria para lazer?<|end_of_text|>
<|im_start|>assistant
```



Eu acho que você gostaria de explorar a cidade, visitar museus, ou simplesmente relaxar em um local tranquilo. Aqui estão algumas sugestões:

1. ****Parque de diversões ou atrações para crianças****: Se você tem um filho pequeno, um parque de diversões ou atrações para crianças pode ser uma excelente opção para se divertir com eles.
2. ****Museus ou centros culturais****: A maioria dos museus e centros culturais oferecem uma variedade de atividades, como exposições, workshops, e palestras.
3. ****Jogos e entretenimento****: Se você gosta de jogar ou simplesmente quer se divertir, alguns locais populares incluem salões de golfe, cinemas, e lojas de jogos.
4. ****Bares e restaurantes****: Para uma refeição saudável, você pode visitar um restaurante ou um bar local, com opções de comida fresca e deliciosas.
5. ****Passeios de barco ou cruzeiros****: Se você gosta de explorar o mar, um passeio de bar

Reflexão sobre as Complexidades e os Desafios do Fine-tuning

Embora o *Fine-tuning* ofereça grandes benefícios, ele também traz uma série de complexidades e desafios. O processo requer uma seleção cuidadosa de dados e a definição precisa de hiperparâmetros, além de ser altamente dependente da qualidade dos dados disponíveis. O risco de *overfitting*, especialmente ao trabalhar com conjuntos de dados menores, é uma preocupação constante. Além disso, a necessidade de poder computacional significativo para treinar modelos de grande escala pode ser um obstáculo.

Esses desafios tornam o *Fine-tuning* uma tarefa que exige um equilíbrio cuidadoso entre arte e ciência, onde a experiência do profissional em *machine learning*, combinada com uma compreensão profunda dos dados e do domínio, é importante para o sucesso.

Desafio 1: Retreinar o modelo novamente com um conjunto maior de dados

Objetivo: Com mais recurso de processamento, por exemplo, as GPUS disponibilizadas pelo [Kaggle](#), faça novamente o treinamento e avalie em comparação com o

LLama na versão *instruct*.

O que Fazer?

1. Preparar os Dados:

- » Rode as partes principais do código em um notebook do [Kaggle](#)

2. Selecionar e Ajustar o Modelo:

- » altere a porção utilizada no *dataset* de treino e validação
- » Escolha um modelo de linguagem *instruct*, por exemplo:
- » "unsloth/Llama-3.2-1B-Instruct-bnb-4bit"

3. Treinamento do Modelo:

- » Gere diversos modelos com a alteração de alguns hiperparâmetros como:
 - » a acumulação de gradiente
 - » o parâmetro *rank* do Lora
 - » a taxa de aprendizado
- » Na classe SFTTrainer sete "packing=False"

4. Testes e Avaliação:

- » Após o treinamento, faça uma análise descritiva, utilizando algumas perguntas para comparar os modelos gerados.

Desafio 2 : Desenvolver um Chatbot para Suporte Técnico com Fine-tuning

Objetivo: Criar um chatbot especializado em suporte técnico, utilizando *Fine-tuning* para adaptar um modelo de linguagem pré-existente às necessidades específicas da área.

O que Fazer:

1. Preparar os Dados:

- » Colete e organize um conjunto de dados específico de suporte técnico. Isso pode incluir logs de atendimento, FAQs, transcrições de chat, e qualquer outra documentação relevante.

- » Defina os hiperparâmetros para o *Fine-tuning*, como a taxa de aprendizado, o número de épocas, e o tamanho do batch. Esses parâmetros devem ser ajustados com base na complexidade e no volume dos dados.

3. Treinamento do Modelo:

- » Realize o *Fine-tuning* do modelo utilizando os dados de suporte técnico. Monitore o treinamento para evitar o *overfitting*, que pode ocorrer se o modelo se tornar muito especializado nos dados de treinamento.

4. Testes e Avaliação:

- » Após o treinamento, teste o chatbot em situações de suporte técnico controladas. Utilize um conjunto de dados de validação que não foi usado no treinamento para avaliar a eficácia do modelo.
- » Analise as respostas do chatbot para garantir que elas sejam precisas, úteis e pertinentes ao contexto.

Dicas:

- » Considere iterar o processo de *Fine-tuning* com diferentes subconjuntos de dados para refinar as respostas do chatbot.
- » Mantenha o foco na escalabilidade do chatbot, garantindo que ele possa lidar com uma variedade de consultas técnicas.

Desafio 3 : Criar um Assistente Pessoal com In-context Learning e Fine-tuning

Objetivo: Desenvolver um assistente pessoal que combine In-context Learning e Fine-tuning para gerenciar tarefas diárias com eficiência.

O que Fazer:

1. Definir Cenários de Uso:

- » Identifique e liste as tarefas diárias que o assistente pessoal deverá executar, como agendamento de reuniões, lembretes, busca de informações, e gestão de listas de tarefas.
- » Descreva cenários de uso realistas que o assistente pode enfrentar, considerando diferentes contextos e necessidades do usuário.

2. Treinamento e Fine-tuning:

- » Use Fine-tuning para especializar o modelo nas tarefas diárias identificadas.
- » Utilize dados específicos relacionados ao gerenciamento de tempo, produtividade pessoal, e outros aspectos da vida cotidiana.
- » Configure In-context Learning para que o modelo possa adaptar suas respostas com base nos exemplos fornecidos no prompt. Isso permitirá ao assistente pessoal ajustar-se dinamicamente a diferentes cenários.

3. Integração das Técnicas:

- » Combine Fine-tuning com In-context Learning para criar um modelo híbrido. Fine-tuning fornecerá uma base sólida de conhecimento específico, enquanto In-context Learning permitirá que o assistente se ajuste a novas situações em tempo real.
- » Teste o assistente em diferentes cenários para garantir que ele responda de forma adequada e útil, independentemente das variações contextuais.

4. Avaliação da Utilidade:

- » Avalie a utilidade e adaptabilidade do assistente em uma variedade de tarefas diárias. Isso pode incluir testes práticos com usuários reais ou em ambientes simulados.
- » Colete feedback e faça ajustes conforme necessário para melhorar a eficiência e a relevância das respostas do assistente.

Dicas:

- » Dê atenção especial à interação entre In-context Learning e Fine-tuning para garantir que o assistente possa tanto oferecer respostas bem-treinadas quanto se adaptar a novos contextos rapidamente.
- » Certifique-se de que o assistente seja capaz de entender e responder a comandos complexos, combinando múltiplas tarefas ou consultas.



SAIBA MAIS...

✿ Artigos científicos:

✿ "*Language Models are Few-Shot Learners*" (Brown, et al., 2020): este artigo introduziu o conceito de aprendizado *in-context* e demonstrou a capacidade dos LLMs de generalizar a partir de poucos exemplos.

✿ "LoRA: *Low-Rank Adaptation of Large Language Models*" (Hu et al., 2022): explore a técnica LoRA para *fine-tuning* de LLMs, reduzindo custos computacionais.

✿ Pesquise por artigos sobre "*in-context learning*", "*few-shot learning*", "*fine-tuning*" e "*parameter-efficient fine-tuning*" no arXiv.org para se manter atualizado sobre as pesquisas mais recentes.

✿ Artigos do Medium:

✿ [Desmistificando o fine tuning de LLMs na prática: PEFT, LoRA, QLORA e Hambúrgueres](#)

✿ [Low-Rank Adaptation \(LoRA\): From intuition to implementation to interview questions](#)

✿ [Minds and Machines — AI for mental health support, fine-tuning LLMs with LoRA in practice](#)

✿ [Fine-tuning LLMs using LoRA](#)

✿ [Creating your own ChatGPT: a guide to fine-tuning LLMs with LoRA](#)

✿ [LoRA for fine-tuning LLMs explained with codes and example](#)



PARA RELEMBRAR...

✿ Nessa Unidade, exploramos duas importantes técnicas para adaptar e otimizar LLMs: *In-Context Learning* e *Fine-tuning*. Aqui estão os principais conceitos:

✿ ICL: capacidade do LLM de aprender novas tarefas a partir de exemplos fornecidos durante a interação, sem atualização explícita dos parâmetros. O modelo aprende com os exemplos mostrados, como se estivesse sendo treinado no momento.

✿ *Zero-Shot Learning*: o modelo realiza uma tarefa sem nenhum exemplo prévio, apenas com seu conhecimento pré-treinado.

✿ *One-Shot Learning*: o modelo recebe apenas um exemplo para aprender e se adaptar à tarefa.

✿ *Few-Shot Learning*: o modelo recebe poucos exemplos (geralmente 3 a 5) para aprender a tarefa.

✿ *Fine-tuning*: processo de treinamento de um modelo pré-treinado em um conjunto de dados menor e específico para uma tarefa, ajustando seus parâmetros para otimizar o desempenho nessa tarefa. É como especializar o modelo em uma área particular.

✿ LoRA: técnica que adiciona pequenas camadas lineares ao LLM original, treinando apenas essas camadas e mantendo o resto do modelo congelado.

Unidade IV
***Retrieval
Augmentation
Generation (RAG)***



Unidade IV - *Retrieval Augmentation Generation*

Até agora, exploramos como os LLMs podem gerar texto, aprendendo padrões complexos a partir de dados massivos. Existe um desafio inerente a essa abordagem: a informação que o modelo possui está limitada ao que foi armazenado durante seu treinamento. Por exemplo, um LLM treinado antes de 2020 não teria conhecimento da pandemia de COVID-19. Essa limitação pode ser problemática quando precisamos que o modelo acesse informações factuais precisas e atualizadas.

É aqui que entra a Geração Aumentada por Recuperação, ou RAG, do inglês *Retrieval Augmented Generation*. Essa técnica permite que os LLMs acessem e utilizem informações externas em tempo real, expandindo suas capacidades. O RAG pode ser comparado a um "cérebro" adicional para o LLM, um grande repositório de informações ao qual ele pode recorrer sempre que precisar.

4.1 Como Funciona o *Retrieval Augmentation Generation*?

Na Figura 34, é ilustrado o funcionamento de um sistema RAG. Começamos com um usuário que formula uma pergunta sobre o CEO do OpenAI, Sam Altman, e sua saída e retorno à empresa. O LLM, como um modelo pré-treinado, não tem acesso a informações atualizadas sobre esse evento. É aqui que o RAG entra em ação.

O sistema RAG realiza uma indexação de um conjunto de documentos, neste caso, artigos de notícias relevantes ao assunto da pergunta. Essa indexação envolve a divisão do texto do documento em vários pedaços, denominados *chunks*. Por meio de um processo de recuperação, o sistema identifica os *chunks* mais relevantes à pergunta do usuário.

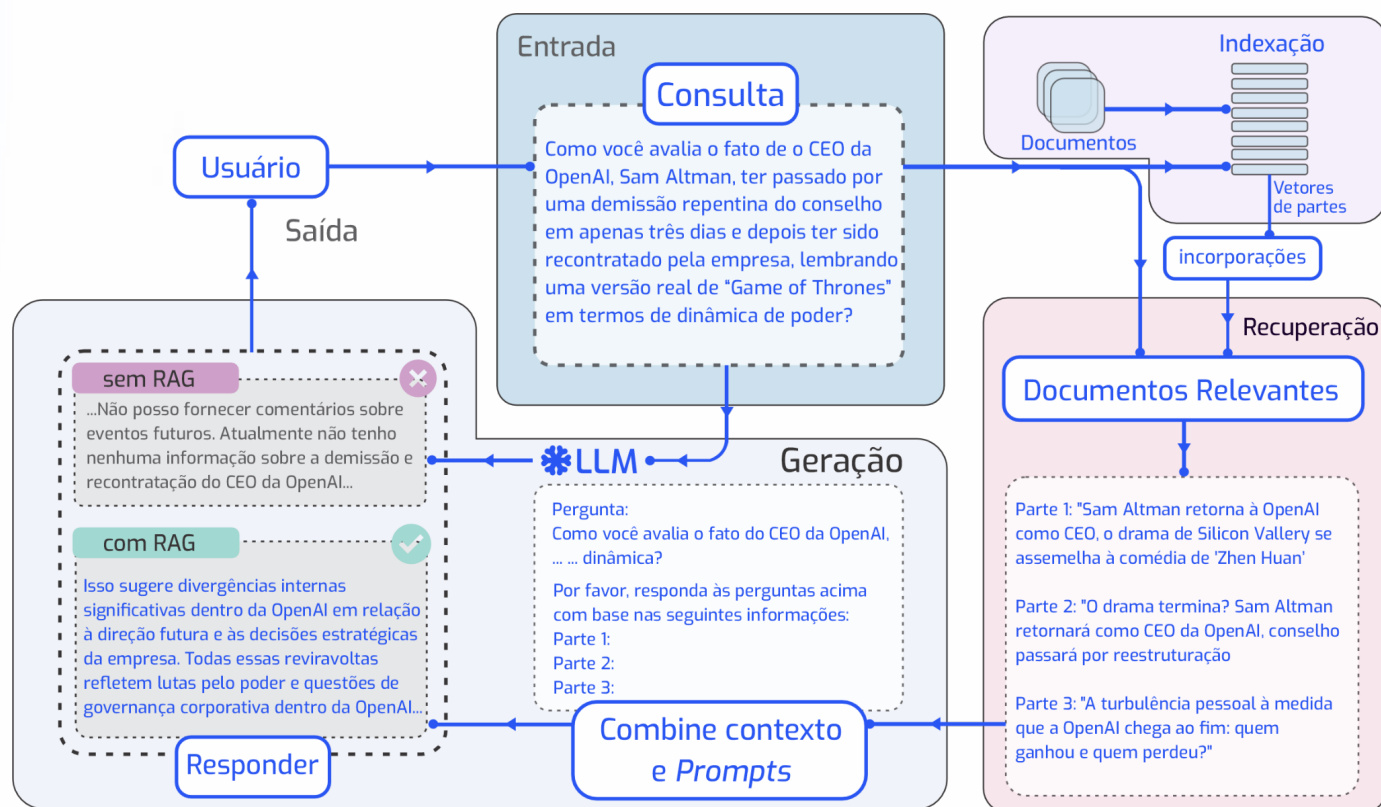
Em seguida, o sistema combina o contexto da pergunta original com as informações extraídas dos documentos relevantes, formando um *prompt* mais completo para o LLM. Esse *prompt* enriquece o LLM com informações contextuais, permitindo que ele gere uma resposta mais precisa e completa.

O sistema RAG, ao incorporar informações externas, amplia o conhecimento do LLM, tornando-o mais capaz de lidar com perguntas complexas e atualizadas. Essa técnica permite que os LLMs possam acessar e processar informações de forma mais dinâmica.

Na Figura 34, é demonstrado como o processo de RAG pode ser dividido em três etapas, que correspondem a:

1. **Indexação:** documentos são divididos em trechos, codificados em vetores e armazenados em um banco de dados vetorial.
2. **Recuperação:** recupera os k principais trechos mais relevantes para a pergunta com base na similaridade semântica.
3. **Geração:** insere a pergunta original e os trechos recuperados em um LLM para gerar a resposta final.

Figura 34 - Visão geral de como funciona o *Retrieval Augmentation Generation* (RAG) com três etapas



Fonte: adaptada de Gao et al. (2023).

A indexação envolve a criação de um índice da base de dados, onde os documentos são organizados de maneira a facilitar a recuperação das informações relevantes. Os documentos são divididos em unidades menores (chunks), como frases ou parágrafos, e transformados em representações vetoriais (*embeddings*). Este processo

envolve transformar o conteúdo textual (*chunks*) em representações vetoriais (*embeddings*) e armazenar estas representações vetoriais em um banco de dados. Os *embeddings* capturam o significado semântico do texto e são armazenadas em um índice, facilitando a busca por similaridade, em vez da busca textual.

O RAG inicia com o envio de uma consulta do usuário no formato textual. Quando o sistema recebe uma consulta, ele transforma a consulta em uma representação vetorial. O sistema precisa do texto da consulta no formato vetorial (*embeddings*) para conseguir buscar os *chunks* mais relevantes na base de dados vetorial, previamente armazenada. Ou seja, o sistema compara os *embeddings* dos *chunks* dos documentos com o *embedding* da consulta para definir quais *chunks* possuem maior similaridade com a consulta. Estes *chunks* são recuperados e fornecidos ao LLM como contexto adicional.

A etapa de geração utiliza o LLM para formular a resposta final. O modelo combina o contexto recuperado com a consulta do usuário e gera uma resposta, aproveitando tanto o conhecimento externo quanto sua capacidade de processamento de linguagem natural. Leia o exemplo na Figura 35.

Figura 35 - Exemplo de cenário com uso de *Retrieval Augmentation Generation* (RAG)

Exemplo:

Podemos ter um cenário com um usuário perguntando: "Quem ganhou o Oscar de melhor ator em 2024?". O sistema RAG indexaria uma base de dados de cinema, como o IMDb (Internet Movie Database - base online sobre a indústria do entretenimento com dados sobre filmes, séries de TV, celebridades e outros). Na etapa de recuperação, o sistema buscaria por documentos relacionados a "Oscar de melhor ator" e "2024". Os documentos relevantes seriam recuperados e fornecidos ao LLM. A etapa de geração combinaria esses documentos com a consulta original e geraria a resposta: "Cillian Murphy ganhou o Oscar de melhor ator em 2024 por sua atuação em Oppenheimer".

Fonte: autoria própria.



4.2 Arquiteturas *Retrieval Augmentation Generation*

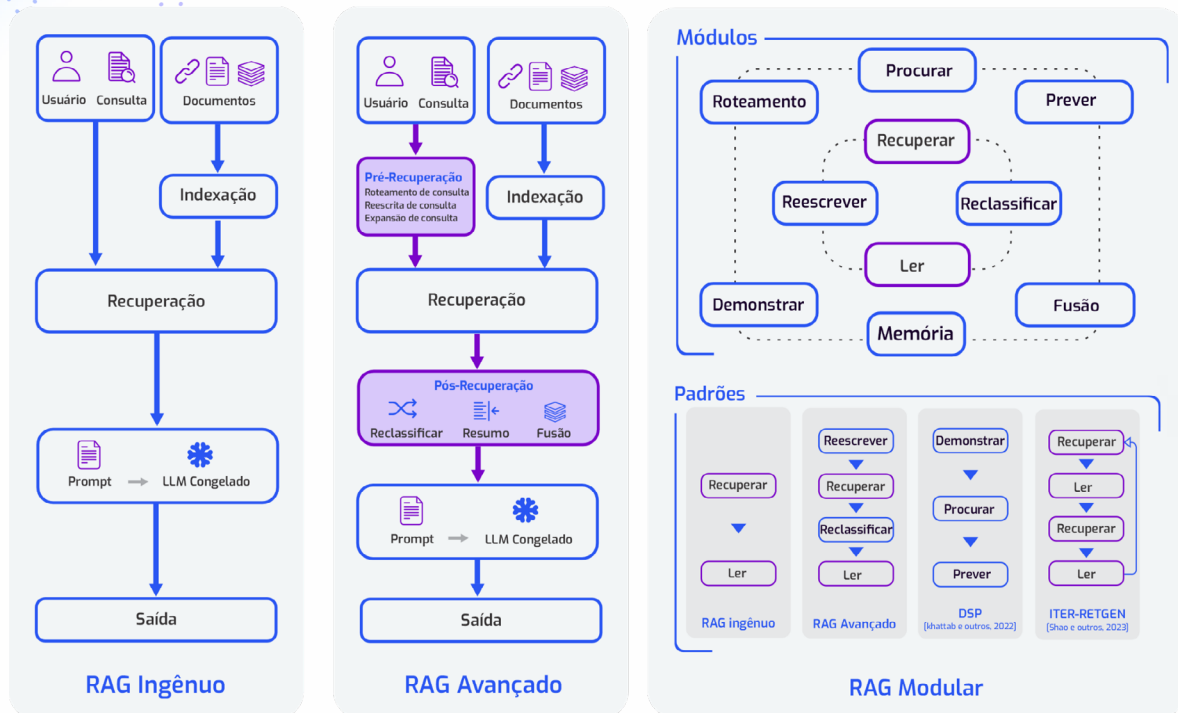
A evolução das arquiteturas de RAG tem acompanhado o avanço das técnicas de aprendizado profundo e das necessidades crescentes por respostas mais precisas e contextualmente relevantes. As arquiteturas podem ser classificadas em três categorias principais: **Naive**, **Avançada** e **Modular**. Cada uma dessas arquiteturas oferece diferentes abordagens para a integração de recuperação e geração, variando em complexidade e desempenho.

Na Figura 36, são mostradas três arquiteturas RAG diferentes: *naive* RAG, *advanced* RAG e *modular* RAG:

- » *Naive* RAG é a arquitetura mais simples. Os documentos são indexados no formato vetorial e uma consulta é enviada para a etapa de recuperação. Os documentos recuperados são então usados para criar um *prompt* que é enviado para um LLM. O LLM então gera uma saída com base nas informações dos documentos recuperados.
- » *Advanced* RAG adiciona uma etapa de pós-recuperação ao *pipeline*. Esta etapa pode incluir tarefas como re-ranqueamento, sumarização e fusão, que ajudam a melhorar a relevância e a qualidade dos documentos recuperados. O *prompt* é então gerado com base nesses documentos aprimorados e enviados para o LLM para gerar a saída.
- » *Modular* RAG oferece a arquitetura mais flexível e avançada, dividindo o sistema RAG em módulos reutilizáveis, como demonstração, pesquisa, leitura, recuperação, re-ranqueamento, reescrita e previsão. Esses módulos podem ser montados em vários *pipelines* para atender a necessidades específicas. Por exemplo, o *pipeline* DSP (demonstração, pesquisa, previsão) mostrado na Figura 30 usa os módulos demonstração, pesquisa e previsão para gerar uma saída. A natureza modular do *modular* RAG permite uma maior flexibilidade e personalização em comparação com o *naive* e *advanced* RAG.

Na Figura 36, são demonstrados os três paradigmas dos modelos RAG. O Naive RAG (ou RAG ingênuo), mostrado à esquerda, consiste principalmente em três partes: indexação, recuperação e geração. O RAG avançado, exibido ao centro, propõe múltiplas estratégias de otimização em torno da pré-recuperação e pós-recuperação, mantendo um processo semelhante ao do Naive RAG, ainda seguindo uma estrutura em cadeia. O RAG modular, apresentado à direita, herda e evolui a partir do paradigma anterior, oferecendo maior flexibilidade geral.

Figura 36 - Comparação entre os três paradigmas do *Retrieval Augmentation Generation* (RAG)



Fonte: adaptada de Gao et al. (2023).

4.2.1 Arquitetura Naive

A arquitetura *Naive* é a forma mais básica e direta de implementação de um sistema RAG. Nesta abordagem, o processo de recuperação e geração é realizado de forma bastante simples e linear. O *pipeline* de um sistema *Naive* pode ser descrito da seguinte forma:

- » **Recuperação simples:** quando uma consulta é realizada, o sistema busca documentos ou fragmentos de texto relevantes em uma base de dados usando técnicas de recuperação básicas, como a busca por palavras-chave ou modelos de recuperação tradicionais.
- » **Geração direta:** após recuperar os documentos mais relevantes, o LM utiliza essas informações como contexto adicional para gerar a resposta. O processo é uma concatenação das informações recuperadas com a entrada original, seguida da geração do texto.

A arquitetura *Naive* possui diversas limitações. A eficácia do sistema depende diretamente da qualidade das informações recuperadas, que, por sua vez, está atrelada à capacidade limitada dos métodos de recuperação tradicionais. Além disso, a ausência de mecanismos de refinamento entre as etapas de recuperação e geração pode resultar em respostas que, embora baseadas em fatos, carecem de coerência ou precisão.

4.2.2 Arquitetura Avançada

A arquitetura Avançada de RAG melhora o *pipeline Naive*, incorporando técnicas mais otimizadas tanto na recuperação quanto na geração. Esta arquitetura é projetada para otimizar o processo em todas as etapas, resultando em uma integração entre os componentes do sistema.

- » **Recuperação:** nesta abordagem, a recuperação é realizada usando técnicas mais avançadas, como *embeddings* semânticos e modelos baseados em aprendizado profundo, como BERT ou variantes. Esses modelos são capazes de entender o contexto e a semântica das consultas, permitindo a recuperação de documentos que não apenas correspondem às palavras-chave, mas que são contextualmente relevantes.
- » **Re-ranking e filtragem:** antes de passar as informações recuperadas para a etapa de geração, a arquitetura avançada realiza um processo de *re-ranking*. Essa etapa adicional reordena e refina os resultados da recuperação, filtrando aqueles que são mais relevantes e confiáveis. Esse *re-ranking* pode ser baseado em critérios adicionais, como a confiança do modelo ou a relevância semântica.
- » **Geração contextualizada:** com uma base de informações mais refinada e relevante, o LM é capaz de gerar respostas mais coerentes e precisas. A geração não é apenas uma simples concatenação; o modelo pode reescrever ou adaptar a informação recuperada para que ela se integre de maneira natural e contextualizada à resposta.

4.2.3 Arquitetura Modular

A arquitetura modular é a mais flexível e escalável das três abordagens. Ela trata as etapas de recuperação e geração como módulos independentes, que podem ser combinados ou substituídos conforme necessário. Essa modularidade permite uma otimização personalizada e a experimentação com diferentes técnicas e modelos em cada etapa do processo.

- » **Modularidade na recuperação:** a recuperação pode ser realizada por um ou mais módulos especializados. Por exemplo, um módulo pode ser responsável pela recuperação de documentos baseados em *embeddings* semânticos, enquanto outro pode utilizar métodos de recuperação tradicionais ou até mesmo uma abordagem supervisionada. Esses módulos podem ser executados em paralelo ou em sequência, oferecendo uma flexibilidade sem precedentes na construção do *pipeline* de recuperação.
- » **Integração dinâmica e re-ranking:** após a recuperação pelos diferentes módulos, uma etapa de integração dinâmica ocorre, onde os resultados são combinados e reordenados. Essa integração pode ser realizada usando técnicas de aprendizado que levam em conta a confiabilidade dos diferentes módulos e a relevância do contexto.
- » **Geração modular:** a etapa de geração também é modular. Diferentes LMs podem ser utilizados dependendo do tipo de consulta ou da natureza das informações recuperadas. Por exemplo, um modelo mais simples pode ser utilizado para perguntas factuais diretas, enquanto um modelo mais avançado é empregado para consultas que exigem respostas mais elaboradas e contextualizadas.
- » **Ajuste fino:** um dos aspectos mais importantes da arquitetura modular é a capacidade de realizar ajustes finos em tempo real. Isso significa que o sistema pode aprender com seus próprios erros e melhorar continuamente seu desempenho, ajustando os módulos de recuperação e geração conforme necessário.

A arquitetura modular, embora mais complexa, oferece uma solução escalável e adaptável para sistemas RAG. Sua flexibilidade permite não apenas a otimização do desempenho, mas também a adaptação a diferentes domínios e tipos de consultas.

4.3 Retrieval Augmented Generation vs. Fine-tuning

O *fine-tuning*, ou ajuste fino, é uma abordagem comum em LMs onde o modelo pré-treinado é adaptado para uma tarefa específica usando um conjunto de dados de treinamento adicional. Essa técnica tem suas limitações, principalmente em termos de flexibilidade e escalabilidade. Quando um modelo é ajustado para uma tarefa, ele pode perder a capacidade de se adaptar a novos contextos ou informações que surgem após o treinamento.

Em contraste, o RAG oferece uma solução mais dinâmica. Em vez de depender exclusivamente do conhecimento armazenado no modelo, que pode ficar obsoleto ou ser insuficiente, o RAG permite que o modelo acesse uma base de dados externa

continuamente atualizada. Isso não só melhora a precisão das respostas, mas também elimina a necessidade de treinar e ajustar o modelo toda vez que novas informações surgem. Dessa forma, o RAG combina o melhor dos dois mundos: a capacidade gerativa dos LLMs com a atualidade e precisão das informações recuperadas em tempo real.

4.4 Benefícios do *Retrieval-Augmented Generation*

O uso do RAG oferece vários benefícios. Primeiro, ele permite que os LMs acessem informações que podem estar além de sua capacidade de memória interna, especialmente em domínios que evoluem rapidamente, como ciência, tecnologia e notícias. Isso não só melhora a qualidade das respostas, mas também reduz a necessidade de treinar continuamente o modelo com novos dados, já que o modelo pode buscar informações atualizadas em tempo real.

Além disso, o RAG aumenta a transparência. Como as respostas geradas são baseadas em documentos específicos que podem ser referenciados, é mais fácil para os usuários entenderem de onde as informações vieram, o que é essencial em contextos onde a veracidade e a fonte das informações são muito importantes.

4.5 Desafios e Limitações

Apesar dos avanços significativos, o RAG ainda enfrenta desafios. A robustez é um dos principais desafios, com sistemas RAG sendo suscetíveis a informações ruidosas ou contraditórias presentes nas bases de conhecimento externas. A presença de informações irrelevantes pode prejudicar a qualidade da resposta final, levando a inconsistências e informações imprecisas.

Outro desafio é a escalabilidade dos sistemas RAG. À medida que as bases de conhecimento externas crescem em tamanho e complexidade, a velocidade da recuperação e a capacidade de processamento do LLM se tornam fatores críticos. Otimizar a indexação, a recuperação e a geração para lidar com grandes volumes de dados é importante para o sucesso do RAG em cenários do mundo real.

As questões éticas também são relevantes no contexto do RAG. A potencial presença de vieses e a necessidade de garantir a privacidade das informações são aspectos que exigem atenção e desenvolvimento de mecanismos de controle e mitigação. Garantir a confiabilidade e a ética dos sistemas RAG é indispensável para sua ampla adoção e impacto positivo na sociedade.



Objetivos de Aprendizagem:

Aprofundamento no conceito de Retrieval-Augmented Generation (RAG) em Processamento de Linguagem Natural (NLP).

Este colab sobre *Retrieval-Augmented Generation* (RAG) em NLP é dividido em três tópicos principais. O primeiro introduz os fundamentos de RAG, explorando seus componentes básicos e proporcionando a configuração de um sistema simples. O segundo foca na implementação e integração técnica, com estudos de caso e práticas de construção de um sistema RAG. O terceiro tópico aborda aplicações avançadas e projetos, com um foco prático na criação de um projeto que integre RAG a outros conceitos de NLP, desde a definição do escopo até os testes e iterações. O texto segue uma estrutura progressiva para facilitar o aprendizado prático.

Boa! Fico muito feliz que já avançamos suficiente em NLP para podermos discutirmos um pouco sobre **RAG**.

O conceito de *Retrieval-Augmented Generation* (RAG) representa uma evolução significativa nas técnicas de Processamento de Linguagem Natural (NLP), combinando os poderes de modelos de linguagem de grande escala (LLMs) com sistemas de recuperação de informações. Essa abordagem híbrida permite que os modelos de geração não apenas confiem em seu treinamento prévio, mas também acessem e utilizem informações externas em tempo real para gerar respostas mais precisas e informativas.

Em contextos onde vocês já exploraram o *Prompt Engineering* e *In-context Learning*, o RAG adiciona uma camada adicional de flexibilidade e poder. Enquanto o *In-context Learning* aproveita a habilidade do modelo de adaptar suas respostas com base nos exemplos fornecidos diretamente no prompt, o RAG vai além, acessando uma vasta base de dados externos para enriquecer essa resposta. Isso é particularmente útil em situações onde a informação necessária para responder adequadamente não está contida no treinamento original do modelo ou nos exemplos fornecidos no *prompt*.

Por outro lado, quando comparado ao *Fine-tuning*, que ajusta o modelo a uma distribuição específica de dados para melhor desempenho, o RAG permite que o modelo mantenha sua versatilidade geral enquanto utiliza informações específicas

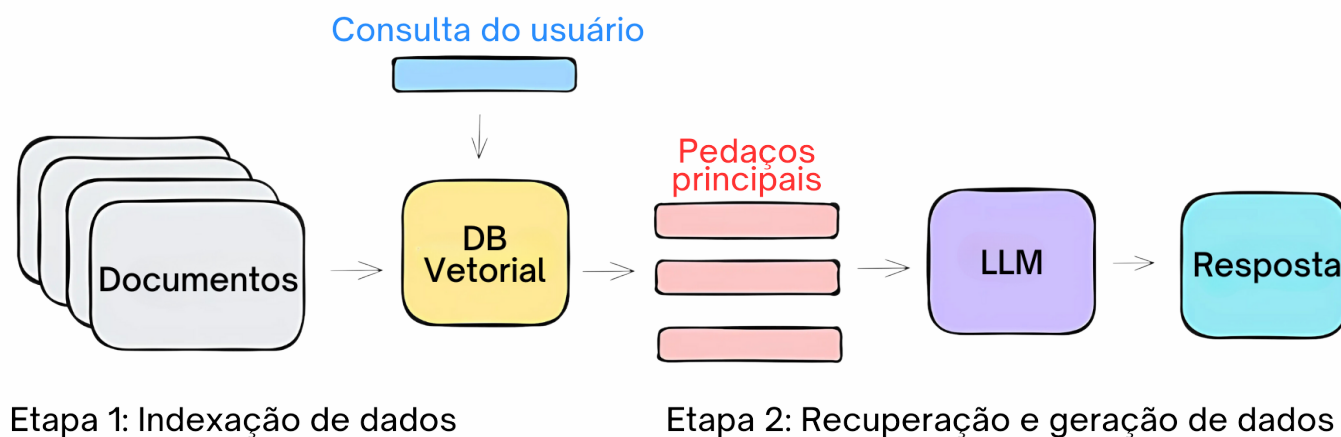
e atualizadas extraídas durante o processo de geração. Isso é alcançado por meio de um mecanismo de recuperação que consulta um banco de dados ou um corpus de documentos em resposta a uma entrada do usuário, selecionando as informações mais relevantes. Posteriormente, essas informações são passadas para o componente de geração, que as integra na resposta final.

Essa técnica não apenas melhora a relevância e a precisão das respostas geradas pelos modelos de linguagem, mas também expande significativamente suas capacidades de aplicação em campos tão diversos como assistentes virtuais, ferramentas de suporte ao cliente e sistemas de recomendação personalizada. Ao integrar a recuperação de informações com a geração de texto, o RAG fornece uma ponte vital entre o conhecimento armazenado e a capacidade de linguagem natural, abrindo novos horizontes para aplicações de NLP.

Na imagem abaixo, temos um pipeline (processo) básico para RAG - *Basic RAG Pipeline*. Neste, no passo 1 (Step 1) há a indexação dos dados e no passo 2 há a geração e a recuperação de dados (*Data Retrieval e Generation*).

Figura 37 - Pipeline RAG Básico

Pipeline RAG Básico



Fonte: JULIJA, Dr. How I built a Simple Retrieval-Augmented Generation (RAG) Pipeline. Medium, 18 fev. 2024.

Disponível em: [Medium](#). Acesso em: 6 jan. 2025. [Imagem adaptada].

Exploração dos Componentes Principais: Mecanismos de Busca e Modelos de Geração

A eficácia do *Retrieval-Augmented Generation* (RAG) reside na sinergia entre dois componentes cruciais: os mecanismos de busca e os modelos de geração. Esses elementos trabalham em conjunto para extrair e incorporar informações relevantes

Mecanismos de Busca

O mecanismo de busca no contexto do RAG não é meramente uma ferramenta para encontrar informações; ele é um componente estratégico que determina a qualidade e a relevância dos dados que serão utilizados pelo modelo de geração. Esse mecanismo pode variar desde uma simples busca de palavras-chave até sistemas mais complexos baseados em inteligência artificial que compreendem o contexto da consulta e a relevância do conteúdo relacionado.

A funcionalidade principal do mecanismo de busca em um sistema RAG é indexar e recuperar informações de um corpus documental extenso, que pode incluir livros, artigos, bases de dados acadêmicas, ou mesmo a internet inteira. O objetivo é encontrar e fornecer trechos de texto que sejam mais pertinentes à pergunta ou ao contexto fornecido pelo usuário. A eficiência deste mecanismo é crucial, pois uma recuperação inadequada pode levar a respostas imprecisas ou irrelevantes.

Modelos de Geração

Uma vez que os dados relevantes são recuperados pelo mecanismo de busca, o modelo de geração entra em ação. Este componente é responsável por interpretar as informações recuperadas e integrá-las de maneira coerente e contextualizada na resposta final. Os modelos de geração em sistemas RAG geralmente são baseados em arquiteturas de transformadores, que são capazes de entender e manipular linguagem natural de forma eficaz.

A integração das informações recuperadas ocorre através de técnicas de fusão, onde o modelo de geração considera tanto a entrada (input) original do usuário quanto as informações adicionais recuperadas para construir uma resposta que seja informativa e fluente. O desafio aqui é garantir que a incorporação desses dados externos seja natural e útil, evitando respostas que pareçam deslocadas ou artificialmente construídas.

Sinergia entre os Componentes

A interação entre o mecanismo de busca e o modelo de geração deve ser fluida e bem coordenada. A precisão na busca determina a qualidade das informações que o modelo de geração tem à disposição para criar respostas. Da mesma forma, a habilidade do modelo de geração de interpretar e utilizar essas informações afeta diretamente a utilidade e a aceitabilidade das respostas geradas. Juntos, esses componentes formam o coração de qualquer sistema RAG, determinando seu sucesso em aplicar conhecimento externo para enriquecer as capacidades de geração de texto de um modelo de NLP.

Após explorar tópicos avançados como *Prompt Engineering*, *In-context Learning*, *Fine-tuning*, e os fundamentos do *Retrieval-Augmented Generation* (RAG), chegamos a um ponto crítico que une todos esses conceitos: a indexação e a recuperação de dados eficiente. Esta área é vital para maximizar o potencial dos sistemas de NLP, especialmente em contextos onde a precisão e a velocidade são essenciais para a performance do sistema. Vamos abordar por que essa eficiência é crucial e como ela impacta diretamente a eficácia dos modelos de linguagem.

Fundamento da Indexação e Recuperação

Indexação e recuperação de dados são processos que organizam informações de maneira que possam ser rapidamente acessadas e utilizadas pelos sistemas de NLP. A indexação transforma grandes conjuntos de dados brutos em estruturas otimizadas para busca, enquanto a recuperação utiliza essas estruturas para trazer informações relevantes em resposta a consultas específicas.

Integração com RAG e Outros Conceitos de NLP

A indexação e recuperação não são apenas funções de backend; elas são integradas profundamente com a forma como os modelos de linguagem operam dentro de um sistema de RAG:

- » **Com *Prompt Engineering* e *In-context Learning***, a recuperação eficiente pode fornecer contextos ricos e relevantes que são fundamentais para moldar a maneira como o modelo interpreta e responde a um prompt.
- » **Durante o *Fine-tuning***, a qualidade dos dados recuperados pode afetar significativamente o aprendizado do modelo, pois dados mais precisos e contextualmente relevantes levam a modelos mais eficazes.

Desafios e Impactos na Performance

A indexação e a recuperação enfrentam desafios técnicos significativos, especialmente em escalas maiores:

- » **Velocidade vs. Precisão:** Encontrar um equilíbrio entre a rapidez na recuperação de dados e a precisão da informação recuperada é crucial. Sistemas que falham em qualquer aspecto podem resultar em respostas lentas ou em respostas rápidas mas imprecisas.

- » **Atualizações Dinâmicas:** Em ambientes onde o conteúdo muda frequentemente, manter a indexação atualizada sem sacrificar o tempo de resposta é um desafio contínuo.

Tecnologias e Ferramentas Envolvidas

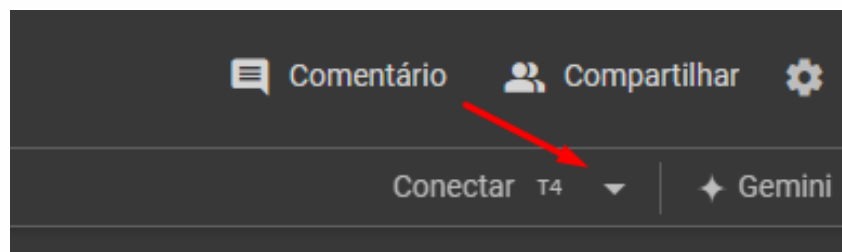
Diversas tecnologias são empregadas para otimizar esses processos:

- » **Elasticsearch:** Utilizado para gerenciar buscas complexas e indexação em tempo real.
- » **FAISS:** Desenvolvido pelo Facebook AI, é usado para a recuperação eficiente de grandes volumes de vetores, facilitando a indexação semântica.

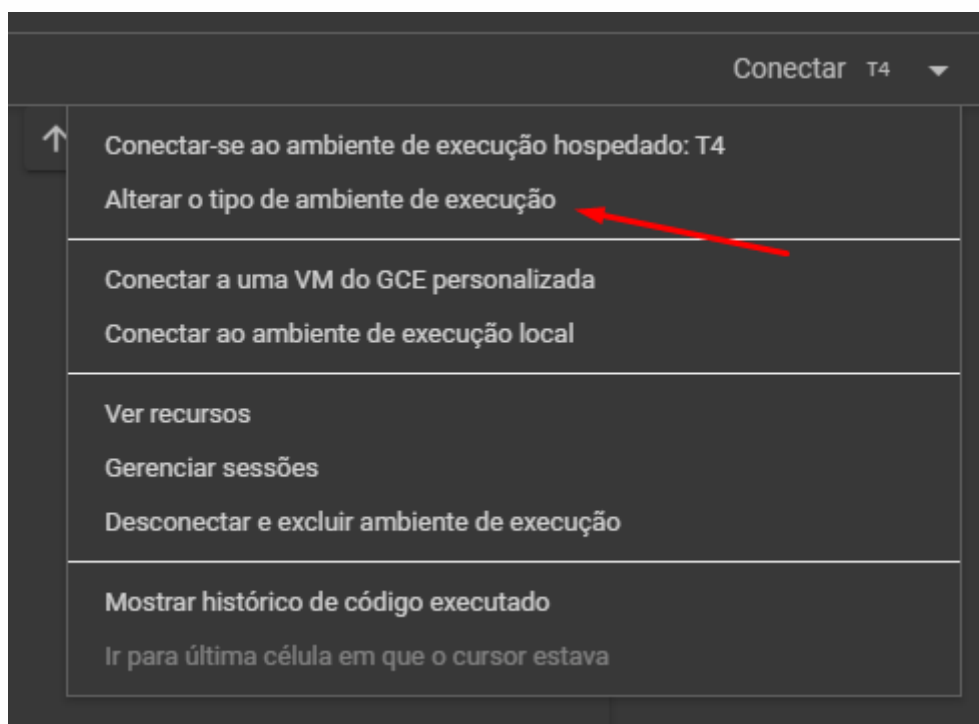
ATIVE A GPU - T4 NA HORA DE SE CONECTAR.

Iremos usar GPU neste notebook e você pode seguir o tutorial abaixo para se conectar com GPU. A GPU vai ser utilizada para criar os embeddings a partir dos documentos textuais.

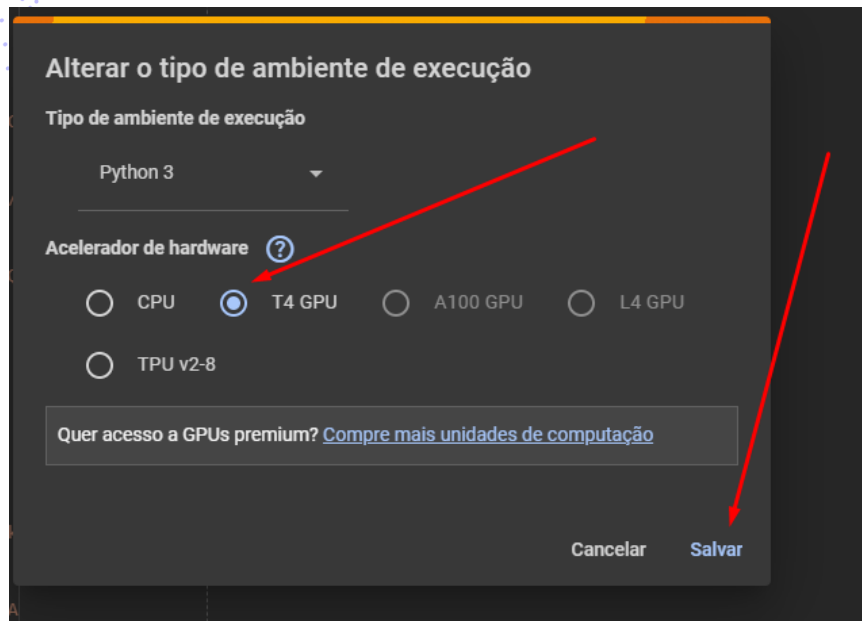
Clique no canto superior direito na seta ao lado de "CONECTAR"



Clique em "ALTERAR O TIPO DE AMBIENTE DE EXECUÇÃO"



Clique em "T4 - GPU" e depois em "SALVAR"



Instalação das bibliotecas

Essa linha de código abaixo instala várias bibliotecas Python usando o gerenciador de pacotes **pip**:

langchain==0.3.9:

- » Instala a biblioteca **langchain** na versão 0.3.9. **langchain** é uma framework para desenvolvimento de aplicações com Large Language Models (LLMs).

langchain_groq==0.2.1:

- » Instala a biblioteca **langchain_groq** na versão 0.2.1. Essa biblioteca provavelmente fornece integração entre **langchain** e o GROQ, uma linguagem de consulta para bancos de dados de documentos.

langchain_community==0.3.8:

- » Instala a biblioteca **langchain_community** na versão 0.3.8. Essa biblioteca é uma coleção de extensões e integrações para **langchain** desenvolvidas pela comunidade.

pypdf==5.1.0:

- » Instala a biblioteca **pypdf** na versão 5.1.0. **pypdf** é uma biblioteca para leitura e manipulação de arquivos PDF.

sentence-transformers==3.2.1:

- » Instala a biblioteca **sentence-transformers** na versão 3.2.1. Essa biblioteca fornece modelos pré-treinados para gerar embeddings de frases, que são representações vetoriais do significado semântico das frases.

faiss-cpu==1.9.0.post1:

- » Instala a biblioteca **faiss-cpu** na versão 1.9.0.post1. **faiss** (Facebook AI Similarity Search) é uma biblioteca para busca eficiente por similaridade em espaços vetoriais de alta dimensão. A versão **faiss-cpu** é otimizada para execução em CPUs.

É importante notar que as versões específicas das bibliotecas são especificadas na linha de código. Isso garante que a aplicação seja executada com as versões testadas e compatíveis entre si.

```
[ ] !pip install langchain==0.3.9 langchain_groq==0.2.1 langchain_community==0.3.8 pypdf==5.1.0 sentence-transformers==3.2.1 faiss-cpu==1.9.0.post1
```

```
➔ Collecting langchain==0.3.9
  Downloading langchain-0.3.9-py3-none-any.whl.metadata (7.1 kB)
Collecting langchain_groq==0.2.1
  Downloading langchain_groq-0.2.1-py3-none-any.whl.metadata (2.9 kB)
Collecting langchain_community==0.3.8
  Downloading langchain_community-0.3.8-py3-none-any.whl.metadata (2.9 kB)
Collecting pypdf==5.1.0
  Downloading pypdf-5.1.0-py3-none-any.whl.metadata (7.2 kB)
Requirement already satisfied: sentence-transformers==3.2.1 in /usr/local/lib/python3.10/dist-packages (3.2.1)
Collecting faiss-cpu==1.9.0.post1
  Downloading faiss_cpu-1.9.0.post1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (4.4 kB)
Requirement already satisfied: PyYAML>=5.3 in /usr/local/lib/python3.10/dist-packages (from langchain==0.3.9) (6.0.2)
Requirement already satisfied: SQLAlchemy<3,>=1.4 in /usr/local/lib/python3.10/dist-packages (from langchain==0.3.9) (2.0.36)
Requirement already satisfied: aiohttp<4.0.0,>=3.8.3 in /usr/local/lib/python3.10/dist-packages (from langchain==0.3.9) (3.11.2)
Requirement already satisfied: async-timeout<5.0.0,>=4.0.0 in /usr/local/lib/python3.10/dist-packages (from langchain==0.3.9) (4.0.3)
Collecting langchain-core<0.4.0,>=0.3.21 (from langchain==0.3.9)
  Downloading langchain_core-0.3.21-py3-none-any.whl.metadata (6.3 kB)
Requirement already satisfied: langchain-text-splitters<0.4.0,>=0.3.0 in /usr/local/lib/python3.10/dist-packages (from langchain==0.3.9) (0.3.2)
Requirement already satisfied: langsmith<0.2.0,>=0.1.17 in /usr/local/lib/python3.10/dist-packages (from langchain==0.3.9) (0.1.143)
Requirement already satisfied: numpy<2,>=1.22.4 in /usr/local/lib/python3.10/dist-packages (from langchain==0.3.9) (1.26.4)
Requirement already satisfied: pydantic<3.0.0,>=2.7.4 in /usr/local/lib/
```

continua

```
python3.10/dist-packages (from langchain==0.3.9) (2.9.2)
Requirement already satisfied: requests<3,>=2 in /usr/local/lib/py-
thon3.10/dist-packages (from langchain==0.3.9) (2.32.3)
Requirement already satisfied: tenacity!=8.4.0,<10,>=8.1.0 in /usr/local/
lib/python3.10/dist-packages (from langchain==0.3.9) (9.0.0)
Collecting groq<1,>=0.4.1 (from langchain_groq==0.2.1)
  Downloading groq-0.13.0-py3-none-any.whl.metadata (13 kB)
Collecting SQLAlchemy<3,>=1.4 (from langchain==0.3.9)
  Downloading SQLAlchemy-2.0.35-cp310-cp310-manylinux_2_17_x86_64.
manylinux2014_x86_64.whl.metadata (9.6 kB)
Collecting dataclasses-json<0.7,>=0.5.7 (from langchain_community==0.3.8)
  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==0.3.8)
  Downloading httpx_sse-0.4.0-py3-none-any.whl.metadata (9.0 kB)
Collecting pydantic-settings<3.0.0,>=2.4.0 (from langchain_communi-
ty==0.3.8)
  Downloading pydantic_settings-2.6.1-py3-none-any.whl.metadata (3.5 kB)
Requirement already satisfied: typing_extensions>=4.0 in /usr/local/lib/
python3.10/dist-packages (from pypdf==5.1.0) (4.12.2)
Requirement already satisfied: transformers<5.0.0,>=4.41.0 in /usr/local/
lib/python3.10/dist-packages (from sentence-transformers==3.2.1) (4.46.2)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-pa-
ckages (from sentence-transformers==3.2.1) (4.66.6)
Requirement already satisfied: torch>=1.11.0 in /usr/local/lib/python3.10/
dist-packages (from sentence-transformers==3.2.1) (2.5.1+cu121)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/
dist-packages (from sentence-transformers==3.2.1) (1.5.2)
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-pa-
ckages (from sentence-transformers==3.2.1) (1.13.1)
Requirement already satisfied: huggingface-hub>=0.20.0 in /usr/local/lib/
python3.10/dist-packages (from sentence-transformers==3.2.1) (0.26.2)
Requirement already satisfied: Pillow in /usr/local/lib/python3.10/dist-
-packages (from sentence-transformers==3.2.1) (11.0.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dis-
t-packages (from faiss-cpu==1.9.0.post1) (24.2)
Requirement already satisfied: aiohappyeyeballs>=2.3.0 in /usr/local/lib/
python3.10/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain==0.3.9)
(2.4.3)
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/py-
thon3.10/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain==0.3.9)
(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==0.3.9) (24.2.0)
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/py-
thon3.10/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain==0.3.9)
(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==0.3.9) (6.1.0)

Requirement already satisfied: propcache>=0.2.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain==0.3.9) (0.2.0)

Requirement already satisfied: yarl<2.0,>=1.17.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp<4.0.0,>=3.8.3->langchain==0.3.9) (1.17.2)

Collecting marshmallow<4.0.0,>=3.18.0 (from dataclasses-json<0.7,>=0.5.7->langchain_community==0.3.8)

Downloading marshmallow-3.23.1-py3-none-any.whl.metadata (7.5 kB)

Collecting typing-inspect<1,>=0.4.0 (from dataclasses-json<0.7,>=0.5.7->langchain_community==0.3.8)

Downloading typing_inspect-0.9.0-py3-none-any.whl.metadata (1.5 kB)

Requirement already satisfied: anyio<5,>=3.5.0 in /usr/local/lib/python3.10/dist-packages (from groq<1,>=0.4.1->langchain_groq==0.2.1) (3.7.1)

Requirement already satisfied: distro<2,>=1.7.0 in /usr/local/lib/python3.10/dist-packages (from groq<1,>=0.4.1->langchain_groq==0.2.1) (1.9.0)

Requirement already satisfied: httpx<1,>=0.23.0 in /usr/local/lib/python3.10/dist-packages (from groq<1,>=0.4.1->langchain_groq==0.2.1) (0.27.2)

Requirement already satisfied: sniffio in /usr/local/lib/python3.10/dist-packages (from groq<1,>=0.4.1->langchain_groq==0.2.1) (1.3.1)

Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from huggingface-hub>=0.20.0->sentence-transformers==3.2.1) (3.16.1)

Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/python3.10/dist-packages (from huggingface-hub>=0.20.0->sentence-transformers==3.2.1) (2024.10.0)

Requirement already satisfied: jsonpatch<2.0,>=1.33 in /usr/local/lib/python3.10/dist-packages (from langchain-core<0.4.0,>=0.3.21->langchain==0.3.9) (1.33)

Requirement already satisfied: orjson<4.0.0,>=3.9.14 in /usr/local/lib/python3.10/dist-packages (from langsmith<0.2.0,>=0.1.17->langchain==0.3.9) (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,>=0.1.17->langchain==0.3.9) (1.0.0)

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==0.3.9) (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==0.3.9) (2.23.4)

Collecting python-dotenv>=0.21.0 (from pydantic-settings<3.0.0,>=2.4.


```
0->langchain_community==0.3.8)
```

```
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==0.3.9) (3.4.0)
```

```
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests<3,>=2->langchain==0.3.9) (3.10)
```

```
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests<3,>=2->langchain==0.3.9) (2.2.3)
```

```
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests<3,>=2->langchain==0.3.9) (2024.8.30)
```

```
Requirement already satisfied: greenlet!=0.4.17 in /usr/local/lib/python3.10/dist-packages (from SQLAlchemy<3,>=1.4->langchain==0.3.9) (3.1.1)
```

```
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch>=1.11.0->sentence-transformers==3.2.1) (3.4.2)
```

```
Requirement already satisfied: Jinja2 in /usr/local/lib/python3.10/dist-packages (from torch>=1.11.0->sentence-transformers==3.2.1) (3.1.4)
```

```
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.10/dist-packages (from torch>=1.11.0->sentence-transformers==3.2.1) (1.13.1)
```

```
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from sympy==1.13.1->torch>=1.11.0->sentence-transformers==3.2.1) (1.3.0)
```

```
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.10/dist-packages (from transformers<5.0.0,>=4.41.0->sentence-transformers==3.2.1) (2024.9.11)
```

```
Requirement already satisfied: safetensors>=0.4.1 in /usr/local/lib/python3.10/dist-packages (from transformers<5.0.0,>=4.41.0->sentence-transformers==3.2.1) (0.4.5)
```

```
Requirement already satisfied: tokenizers<0.21,>=0.20 in /usr/local/lib/python3.10/dist-packages (from transformers<5.0.0,>=4.41.0->sentence-transformers==3.2.1) (0.20.3)
```

```
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn->sentence-transformers==3.2.1) (1.4.2)
```

```
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn->sentence-transformers==3.2.1) (3.5.0)
```

```
Requirement already satisfied: exceptiongroup in /usr/local/lib/python3.10/dist-packages (from anyio<5,>=3.5.0->groq<1,>=0.4.1->langchain_groq==0.2.1) (1.2.2)
```

```
Requirement already satisfied: httpcore==1.* in /usr/local/lib/python3.10/dist-packages (from httpx<1,>=0.23.0->groq<1,>=0.4.1->langchain_groq==0.2.1) (1.0.7)
```

```
Requirement already satisfied: h11<0.15,>=0.13 in /usr/local/lib/python3.10/dist-packages (from httpcore==1.*->httpx1<,>0.23.0->groq1<,>=0.4.1->langchain_groq==0.2.1) (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,>=0.3.21->langchain==0.3.9) (3.0.0)
```

```
Collecting mypy_extensions>=0.3.0 (from typin-
```

continua

```
g-inspect<1,>=0.4.0->dataclasses-json<0.7,>=0.5.7->langchain_community==0.3.8)
```

```
  Downloading mypy_extensions-1.0.0-py3-none-any.whl.metadata (1.1 kB)
```

```
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch>=1.11.0->sentence-transformers==3.2.1) (3.0.2)
```

```
Downloading langchain-0.3.9-py3-none-any.whl (1.0 MB)
```

```
_____ 1.0/1.0 MB 44.5 MB/s eta 0:00:00
```

```
Downloading langchain_groq-0.2.1-py3-none-any.whl (14 kB)
```

```
Downloading langchain_community-0.3.8-py3-none-any.whl (2.4 MB)
```

```
_____ 2.4/2.4 MB 80.6 MB/s eta 0:00:00
```

```
Downloading pypdf-5.1.0-py3-none-any.whl (297 kB)
```

```
_____ 298.0/298.0 kB 26.9 MB/s eta 0:00:00
```

```
Downloading faiss_cpu-1.9.0.post1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (27.5 MB)
```

```
_____ 27.5/27.5 MB 34.2 MB/s eta 0:00:00
```

```
Downloading dataclasses_json-0.6.7-py3-none-any.whl (28 kB)
```

```
Downloading groq-0.13.0-py3-none-any.whl (108 kB)
```

```
_____ 108.8/108.8 kB 10.1 MB/s eta 0:00:00
```

```
Downloading httpx_sse-0.4.0-py3-none-any.whl (7.8 kB)
```

```
Downloading langchain_core-0.3.21-py3-none-any.whl (409 kB)
```

```
_____ 409.5/409.5 kB 34.8 MB/s eta 0:00:00
```

```
Downloading pydantic_settings-2.6.1-py3-none-any.whl (28 kB)
```

```
Downloading SQLAlchemy-2.0.35-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (3.1 MB)
```

```
_____ 3.1/3.1 MB 47.2 MB/s eta 0:00:00
```

```
Downloading marshmallow-3.23.1-py3-none-any.whl (49 kB)
```

```
_____ 49.5/49.5 kB 2.6 MB/s eta 0:00:00
```

```
Downloading python_dotenv-1.0.1-py3-none-any.whl (19 kB)
```

```
Downloading typing_inspect-0.9.0-py3-none-any.whl (8.8 kB)
```

```
Downloading mypy_extensions-1.0.0-py3-none-any.whl (4.7 kB)
```

```
Installing collected packages: SQLAlchemy, python-dotenv, pypdf, mypy-extensions, marshmallow, httpx-sse, faiss-cpu, typing-inspect, pydantic-settings, groq, dataclasses-json, langchain-core, langchain_groq, langchain, langchain_community
```

```
  Attempting uninstall: SQLAlchemy
```

```
    Found existing installation: SQLAlchemy 2.0.36
```

```
    Uninstalling SQLAlchemy-2.0.36:
```

```
      Successfully uninstalled SQLAlchemy-2.0.36
```

```
  Attempting uninstall: langchain-core
```

```
    Found existing installation: langchain-core 0.3.19
```

```
    Uninstalling langchain-core-0.3.19:
```

```
      Successfully uninstalled langchain-core-0.3.19
```

continua

```
Attempting uninstall: langchain
```

```
Found existing installation: langchain 0.3.7
```

```
Uninstalling langchain-0.3.7:
```

```
Successfully uninstalled langchain-0.3.7
```

```
Successfully installed SQLAlchemy-2.0.35 dataclasses-json-0.6.7 fais-  
s-cpu-1.9.0.post1 groq-0.13.0 httpx-sse-0.4.0 langchain-0.3.9 lang-  
chain-core-0.3.21 langchain_community-0.3.8 langchain_groq-0.2.1 mar-  
shmallow-3.23.1 mypy-extensions-1.0.0 pydantic-settings-2.6.1 pypdf-5.1.0  
python-dotenv-1.0.1 typing-inspect-0.9.0
```

```
[ ] from langchain_groq import ChatGroq # Importa a classe ChatGroq da bi-  
blioteca langchain_groq, usada para interagir com modelos Groq.  
import time # Importa a biblioteca time, que pode ser usada para funções  
relacionadas a manipulação de tempo.  
from langchain_core.prompts import ChatPromptTemplate # Importa Cha-  
tPromptTemplate para criação de prompts customizados em chat.  
from langchain.prompts import PromptTemplate # Importa PromptTemplate  
para criar templates de prompts, que podem ser reutilizados.  
from langchain_core.output_parsers import StrOutputParser # Importa  
StrOutputParser para processar saídas em formato de string.  
from langchain_core.output_parsers import JsonOutputParser # Importa Jso-  
nOutputParser para processar saídas no formato JSON.  
  
import os # Importa o módulo os, que permite interagir com o sistema ope-  
racional.  
from google.colab import userdata # Importa userdata do Google Colab para  
manipular dados do usuário no ambiente Colab.  
os.environ["GROQ_API_KEY"] = userdata.get('GROQ_API_KEY') # Define a va-  
riável de ambiente GROQ_API_KEY com a chave de API do Groq, recuperada do  
Colab.  
  
GROQ_LLM = ChatGroq( # Instancia o modelo de linguagem Groq.  
    model="llama3-8b-8192", # Especifica o modelo a ser utilizado,  
    no caso, o "llama3-8b-8192".  
)
```

Download do arquivo

No código a seguir utilizaremos o texto disponível em um arquivo [.pdf](#). Para tal é necessário realizar o download do [livro](#) "Cardiologia na emergência: guia para o pronto-Socorro". É um arquivo com abordagem prática das emergências cardiológicas elaborado pela Liga Médico-Acadêmica de Cardiologia de Sobral (LIMACS).

O livro está disponível no Google Drive também e o código abaixo faz o download do livro:

```
[ ] !lgdown https://drive.google.com/uc?id=1mnp40g7gTl96hul0233j3zDRtT4qTPna
```



Downloading...

From: <https://drive.google.com/uc?id=1mnp40g7gTl96hul0233j3zDRtT4qTPna>

To: /content/CardiologianaEmergencia.pdf

100% 12.3M/12.3M [00:00<00:00, 18.6MB/s]

Após a realização do download certifique de que o arquivo esteja dentro da pasta que você baixou.

Carregando o arquivo PDF

Este código, em Python, demonstra os passos iniciais para carregar e preparar um arquivo PDF para ser usado no RAG.

Passo a Passo:

1. Importar a Ferramenta:

- » A primeira linha importa a função **PyPDFLoader** da biblioteca **langchain_community**. Essa função é especializada em carregar e processar arquivos PDF.

2. Definir o Caminho do Arquivo:

- » A variável **file_path** armazena o caminho para o arquivo PDF que você deseja analisar.

3. Criar o "Loader":

- » Um objeto **PyPDFLoader** é criado e armazenado na variável **loader**. Este objeto recebe o caminho do arquivo como argumento e será responsável por interagir com o PDF.

4. Carregar e Dividir o PDF:

- » A função `load_and_split()` do objeto `loader` é chamada. Esta função realiza duas tarefas importantes:
 - » **loader**: lê o conteúdo do arquivo PDF.
 - » **split**: separa o documento em páginas individuais, retornando uma lista onde cada elemento representa o conteúdo de uma página. Isso facilita o processamento posterior do texto, permitindo que você trabalhe com cada página separadamente.

```
[ ] from langchain_community.document_loaders import PyPDFLoader

# 1. Aqui estamos importando uma ferramenta (PyPDFLoader) que permite car-
regar arquivos PDF.
# Essa ferramenta faz parte de uma biblioteca chamada "langchain_communi-
ty" usada em processamento de linguagem.

file_path = (
    "CardiologianaEmergencia.pdf" #caso você mude o nome do arquivo, adi-
cione o novo nome depois de "/content/...NOVO NOME.pdf"
)

# 2. Nesta linha, estamos definindo o caminho do arquivo PDF que queremos
carregar.
# O caminho do arquivo PDF é onde ele está armazenado no computador ou na
nuvem (neste caso, "/content/cardiologia.pdf").

loader = PyPDFLoader(file_path)

# 3. Agora estamos criando uma "ferramenta" chamada 'loader' que vai uti-
lizar o caminho do PDF para ler o arquivo.
# O PyPDFLoader será o responsável por abrir e preparar o PDF para ser tra-
balhado.

pages = loader.load_and_split()

# 4. Finalmente, estamos usando a ferramenta 'loader' para carregar o PDF
e dividir o documento em várias partes (páginas).
# O 'load_and_split()' lê o PDF e separa o conteúdo em páginas, facilitan-
do o processamento de cada uma.
```

O código abaixo imprime as páginas do documento:

```
[ ] # Conferindo como estão as páginas.  
    print(pages)
```

```
➔ [Document(metadata={'source': 'CardiologianaEmergencia.pdf', 'page': 0},  
page_content='CARDIOLOGIA NA EMERGÊNCIA: GUIA PARA O \nPRONTO-SOCORRO \nOrganizadores: \nPedro José Leite de Almeida Mendonça \nMateus de Sousa Cavalcante \nFelipe Salim Habib Buhamara Alves Nasser Gurjão \nDara Medeiros Mendes \nBenedito Mesley Lima Portela \nEditora Amplla \nCampina Grande, janeiro de 2023')...]
```

Divisão do documento em chunks

O código abaixo divide um texto longo (no caso, as páginas de um PDF) em pedaços menores, chamados "chunks".

Passo a passo:

1. **Importa a ferramenta:** **RecursiveCharacterTextSplitter** é uma função que sabe dividir textos de forma inteligente, respeitando a estrutura do texto (frases, parágrafos, etc.).
2. **Configura a ferramenta:** define o tamanho máximo de cada chunk (1000 caracteres) e a sobreposição entre chunks (200 caracteres). A sobreposição ajuda a manter o contexto entre os chunks.
3. **Divide o texto:** aplica a ferramenta **text_splitter** nas páginas do PDF, gerando uma lista de chunks menores.

O código gera uma lista chamada **splits** contendo os chunks do texto original, prontos para serem processados por outras ferramentas de NLP.

```
[ ] from langchain_text_splitters import RecursiveCharacterTextSplitter  
    # 1. Aqui estamos importando uma ferramenta chamada 'RecursiveCharacter-  
    TextSplitter' da biblioteca 'langchain_text_splitters'.  
    # Ela é usada para dividir grandes textos em pedaços menores (ou "frag-  
    mentos") de forma inteligente, para que o texto possa ser processado mais  
    facilmente.  
  
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_  
    overlap=200)
```

continua


```
# 2. Agora, estamos criando uma "ferramenta" chamada 'text_splitter' usando a função 'RecursiveCharacterTextSplitter'.  
# Estamos configurando essa ferramenta para dividir o texto em fragmentos de até 1000 caracteres ('chunk_size=1000').  
# Também definimos que haverá uma sobreposição de 200 caracteres entre os fragmentos ('chunk_overlap=200'), ou seja, os últimos 200 caracteres de um fragmento vão aparecer no começo do próximo.
```

```
splits = text_splitter.split_documents(pages)  
# 3. Por fim, estamos aplicando essa ferramenta 'text_splitter' nas páginas do PDF (que foram carregadas antes).  
# O método 'split_documents' vai dividir cada página em pedaços menores, de no máximo 1000 caracteres, com uma pequena sobreposição entre eles.  
# O resultado ('splits') será uma lista contendo os fragmentos de texto prontos para serem usados em processamento de NLP.
```

```
[ ] # Conferindo como foi feito os splits.  
print(splits[10])
```

```
➔ page_content='de Brasília  
Ricardo Leoni Gonçalves Bastos - Universidade  
Federal do Ceará  
Rodrigo da Rosa Pereira - Universidade Federal  
do Rio Grande  
Rubia Katia Azevedo Montenegro -  
Universidade Estadual Vale do Acaraú  
Sabrynna Brito Oliveira - Universidade Federal  
de Minas Gerais  
Samuel Miranda Mattos - Universidade Estadual  
do Ceará  
Shirley Santos Nascimento - Universidade  
Estadual Do Sudoeste Da Bahia  
Silvana Carloto Andres - Universidade Federal  
de Santa Maria  
Silvio de Almeida Junior - Universidade de  
Franca  
Tatiana Paschoalette R. Bachur - Universidade  
Estadual do Ceará | Centro Universitário  
Christus  
Telma Regina Stroparo - Universidade Estadual  
do Centro-Oeste  
Thayla Amorim Santino - Universidade Federal
```

continua

do Rio Grande do Norte

Thiago Sebastião Reis Contarato – Universidade

Federal do Rio de Janeiro

Tiago Silveira Machado – Universidade de

Pernambuco

Virgínia Maia de Araújo Oliveira – Instituto

Federal da Paraíba' metadata={'source': 'CardiologianaEmergencia.pdf',
'page': 4}

Indexação dos chunks em um banco de dados vetorial

O código abaixo cria um banco de dados de "embeddings" de texto, que são representações numéricas do significado de cada pedaço do texto original.

Passo a passo:

1. **Importa ferramentas:** importa funções para gerar embeddings e criar um banco de dados.
2. **Define a função de embedding:** escolhe um modelo pré-treinado (**intfloat/multilingual-e5-large**) para transformar texto em vetores numéricos.
3. **Cria o banco de dados:** usa a biblioteca FAISS para criar um banco de dados que armazena os embeddings de cada chunk do texto, gerados pela função **embedding_function**.

O código gera um banco de dados **db** que permite buscar pedaços de texto com base em seu significado, usando a similaridade entre seus embeddings. Isso é útil para, por exemplo, encontrar as partes do texto mais relevantes para responder a uma pergunta específica.

```
[ ] from langchain_community.embeddings.sentence_transformer import (  
    SentenceTransformerEmbeddings,  
)  
# 1. Estamos importando uma ferramenta chamada 'SentenceTransformerEmbed-  
dings' da biblioteca 'langchain_community'.  
# Ela é usada para converter textos em vetores (números) que representam o  
significado de uma frase.  
# Isso é útil para que o computador possa entender e comparar o significado  
dos textos, usando uma técnica chamada "embeddings".
```

continua

```

from langchain_community.vectorstores import FAISS
# 2. Aqui estamos importando uma ferramenta chamada
# Ele nos ajuda a armazenar esses vetores e realizar buscas rápidas com
# base no significado dos textos.
import torch

embedding_function = SentenceTransformerEmbeddings(
    model_name="intfloat/multilingual-e5-large",
    model_kwargs={"device": "cuda" if torch.cuda.is_available() else
"cpu"}
)
# 3. Nesta linha, estamos criando a "ferramenta" que vai converter o texto
em vetores.
# Estamos usando o modelo "intfloat/multilingual-e5-large", que é capaz de
transformar textos em diversos idiomas em vetores.
# O 'embedding_function' é a função que vamos usar para transformar os
textos em representações numéricas (embeddings).

db = FAISS.from_documents(splits, embedding_function)
# 4. Aqui estamos criando um banco de dados ('db') utilizando o FAISS, que
vai armazenar os vetores (embeddings) gerados a partir dos textos.
# O método 'from_documents' pega os fragmentos de texto (divididos antes
em 'splits') e os converte em vetores usando a 'embedding_function' defi-
nida acima.
# Com esse banco de dados, poderemos realizar buscas por similaridade en-
tre textos, comparando o significado entre eles de maneira eficiente.

```

```

↳ <ipython-input-8-b5de1e5fa7ac>:13: LangChainDeprecationWarning: The class
`HuggingFaceEmbeddings` was deprecated in LangChain 0.2.2 and will be re-
moved in 1.0. An updated version of the class exists in the :class:`~lan-
gchain-huggingface` package and should be used instead. To use it run `pip
install -U :class:`~langchain-huggingface` and import as `from :clas-
s:`~langchain_huggingface import HuggingFaceEmbeddings`.

```

```

    embedding_function = SentenceTransformerEmbeddings(
/usr/local/lib/python3.10/dist-packages/sentence_transformers/cross_en-
coder/CrossEncoder.py:13: TqdmExperimentalWarning: Using `tqdm.autonote-
book.tqdm` in notebook mode. Use `tqdm.tqdm` instead to force console mode
(e.g. in jupyter console)

```

```

    from tqdm.autonotebook import tqdm, trange
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_auth.
py:94: UserWarning:

```

The secret `HF\_TOKEN` does not exist in your Colab secrets.

To authenticate with the Hugging Face Hub, create a token in your settin-
gs tab (<https://huggingface.co/settings/tokens>), set it as secret in your
Google Colab and restart your session.

You will be able to reuse this secret in all of your notebooks.

continua

Please note that authentication is recommended but still optional to access public models or datasets.

```
warnings.warn(  
modules.json: 100% 387/387 [00:00<00:00, 22.4kB/s]  
README.md: 100% 160k/160k [00:00<00:00, 7.41MB/s]  
sentence_bert_config.json: 100% 57.0/57.0 [00:00<00:00, 3.82kB/s]  
config.json: 100% 690/690 [00:00<00:00, 46.9kB/s]  
model.safetensors: 100% 2.24G/2.24G [00:10<00:00, 204MB/s]  
tokenizer_config.json: 100% 418/418 [00:00<00:00, 29.1kB/s]  
sentencepiece.bpe.model: 100% 5.07M/5.07M [00:00<00:00, 21.1MB/s]  
tokenizer.json: 100% 17.1M/17.1M [00:00<00:00, 177MB/s]  
special_tokens_map.json: 100% 280/280 [00:00<00:00, 14.1kB/s]  
1_Pooling/config.json: 100% 201/201 [00:00<00:00, 10.9kB/s]
```

Configuração do Retriever do RAG

Nesta linha, estamos transformando o banco de dados FAISS ('db') em uma ferramenta chamada "retriever". Um "retriever" é usado para buscar informações relevantes dentro do banco de dados. Ele vai procurar por fragmentos de texto no banco de dados que sejam os mais parecidos com a consulta que for feita. Isso permite que a gente recupere os textos mais relevantes ou similares a partir dos embeddings (vetores) armazenados.

```
[ ] retriever = db.as_retriever()
```

Pipeline do RAG

O trecho de código abaixo define e executa uma cadeia RetrievalQA para responder a perguntas de usuários com base em um conjunto de documentos, com foco em um contexto de serviços hospitalares.

O código cria um sistema de perguntas e respostas que recupera informações relevantes de um conjunto de documentos (indexados usando embeddings e um banco de dados vetorial como Faiss) e as usa para responder a perguntas de usuários. A prompt instrui o modelo de linguagem a adotar uma persona profissional de hospital, garantindo que as respostas sejam adequadas ao contexto. O resultado final é a resposta à pergunta do usuário, gerada com base no contexto recuperado e formatada de acordo com o template da prompt.

1. Definindo o template do prompt:

- » **template**: Define um template de prompt em formato de string. Ele instrui o modelo de linguagem a se comportar como um assistente profissional de hospital e a usar o contexto fornecido para responder à pergunta do usuário.
- » **QA_CHAIN_PROMPT**: Cria um objeto **PromptTemplate** usando a string **template** e especificando que as variáveis **context** (contexto) e **question** (pergunta) serão inseridas no template posteriormente.
 - » A variável **context** será preenchida com os trechos recuperados do banco de dados.
 - » A variável **question** será preenchida com a pergunta do usuário.

```
[ ] from langchain.chains import RetrievalQA
    from langchain.prompts import PromptTemplate

# 1. Defina o template do prompt
template = """Atue com a personalidade de um assistente profissional especializado em tarefas de resposta a perguntas relacionadas a serviços hospitalares.

Use os seguintes pedaços de contexto para responder à pergunta do usuário. Se você não souber a resposta, diga apenas que não tem informações suficientes para responder.

Contexto: {context}
Pergunta: {question}"""

QA_CHAIN_PROMPT = PromptTemplate(
    input_variables=["context", "question"], template=template
)
```

2. Criando a cadeia RetrievalQA:

- » **qa_chain**: Cria a cadeia **RetrievalQA** usando:
 - » **GROQ_LLM**: modelo de linguagem grande (LLM) pré-definido. Esta variável foi criada previamente.
 - » **chain_type="stuff"**: especifica o tipo de cadeia como "stuff", o que significa que todo o contexto recuperado será inserido diretamente na prompt.

- » **retriever**: objeto retriever, criado anteriormente, responsável por buscar documentos relevantes do conjunto de dados com base na pergunta do usuário.
- » **return_source_documents=True**: conjunto de passos (chain) que deve retornar os documentos de origem usados para gerar a resposta.
- » **chain_type_kwargs={"prompt": QA_CHAIN_PROMPT}**: template da prompt definido anteriormente para a cadeia.

```
[ ] # 2. Crie a cadeia RetrievalQA usando o retriever, o modelo de linguagem e o prompt
qa_chain = RetrievalQA.from_chain_type(
    llm=GROQ_LLM,
    chain_type="stuff",
    retriever=retriever,
    return_source_documents=True,
    chain_type_kwargs={"prompt": QA_CHAIN_PROMPT},
)
```

3. Pergunta do usuário e execução da cadeia:

- » **query**: define a pergunta do usuário como uma string.
- » **result**: executa a cadeia qa_chain com a pergunta do usuário e extrai a resposta do resultado.
- » **print(result)**: imprime a resposta gerada pela cadeia.

```
[ ] # 3. Pergunta do usuário
query = "Como usar um desfibrilador em situações de emergência?"

# 3. Função para interagir com o usuário e obter respostas
result = qa_chain({"query": query})["result"]

print(result)
```

```
➡ {'query': 'Como usar um desfibrilador em situações de emergência?', 'result': 'Obviamente!\n\nDe acordo com o contexto, um desfibrilador externo automático (DEA) é um dispositivo que pode ser utilizado por leigos em ambiente extra-hospitalar para desfibrilar uma vítima em parada cardíaca. Ele identifica se o ritmo é "chocável" ou não de forma automática e indica ou não a desfibrilação.\n\nPara usar um DEA, siga os passos:\n\n1. Coloque os eletrodos no local tradicionalmente recomendado (um na região apical e
```

continua

outro na região infraclavicular direita).\n2. Ativar o DEA e seguir suas instruções. Ele indicará se o ritmo é "chocável" ou não.\n3. Se o ritmo for "chocável", o DEA indicará a carga elétrica a ser utilizada e a desfibrilação será realizada automaticamente.\n4. Se o ritmo não for "chocável", o DEA indicará que não há necessidade de desfibrilação.\n5. Logo após a desfibrilação, é recomendado recomeçar um novo ciclo de RCP (ressuscitação cardiopulmonar) e verificar o ritmo da vítima.\n\nLembre-se de que a carga elétrica recomendada depende do tipo de DEA utilizado (monofásico ou bifásico) e que, se houver dúvida, use 200J como carga padrão.\n\nEssas são as informações básicas sobre como usar um desfibrilador em situações de emergência.', 'source_documents': [Document(metadata={'source': 'Cardiologia-naEmergencia.pdf', 'page': 11}, page_content='realizada por pessoas leigas minimamente treinadas para tal, com auxílio dos Desfibriladores \nExternos Automáticos (DEA)^{5,8}. \nA partir disso, podem ser iniciadas as manobras de SBV, que consistem em quatro \npilares, sendo representadas por meio de letras: C (ciculation), A (airway), B (Breathing) e D'), Document(metadata={'source': 'CardiologianaEmergencia.pdf', 'page': 13}, page_content='forma automática, indicando ou não a desfibrilação, contribuindo para aumento das chances de \nsobrevivência da vítima. A posição tradicional recomendada dos eletrodos é um na região apical \ne outro na região infraclavicular direita. A carga recomendada depende do cardiodesfibrilador, \nsendo indicado 360J no monofásico. Já no bifásico, a preferência é pela carga preconizada pelo \nfabricante do aparelho, geralmente entre 120 -200 J, mas se dúvida, utilize 200J. Logo após a \nrealização do choque, é recomendado recomeçar novo ciclo de RCP, para posteriormente ser \nchecado o ritmo^{5,8}. \nBOX 3. APÓS A CHEGADA DA EQUIPE MÉDICA, QUE MEDIDAS DE SUPORTE MAIS \nAVANÇADAS PODEM SER TOMADAS? \nCom a chegada do Serviço de Atendimento Móvel de Urgência (SAMU), foram \nrealizadas intubação orotraqueal, medicações e cuidados necessários condizentes com o \nsuporte de vida avançado (SAV). \nA vítima foi encaminhada para um hospital secundário, dando entrada inconsciente,'), Document(metadata={'source': 'Cardiologia-naEmergencia.pdf', 'page': 13}, page_content='13 Cardiologia na Emergência: Guia para o Pronto-Socorro \nCAPÍTULO I | RESSUSCITAÇÃO CARDIOPULMONAR (RCP) \nfeita sem proteção (boca a boca ou boca nariz) ou com proteção (uso de dispositivos e válvulas, \ntais como dispositivos com válvula unidirecional, máscaras ou bolsa-máscara)^{5,8}. \n2.4. D (DEFIBRILLATION - DESFIBRILAÇÃO) \nEste procedimento pode ser realizado assim que possível, quando corretamente \nindicado, ou seja, na presença de ritmos de parada "chocáveis" (Taquicardia Ventricular s/ \npulso e Fibrilação Ventricular). Existe o cardiodesfibrilador manual, utilizado em ambiente \nintra-hospitalar, e o Desfibrilador Externo Automático (DEA), que pode ser utilizado por leigos \nem ambiente extra-hospitalar, já que identifica se o ritmo é "chocável" ou "não -chocável" de \nforma automática, indicando ou não a desfibrilação, contribuindo para aumento das chances de \nsobrevivência da vítima. A posição tradicional recomendada dos eletrodos é um na região apical'), Document(metadata={'source': 'CardiologianaEmergencia.pdf', 'page': 12}, page_content='12 Cardiologia na Emergência: Guia para o Pronto-Socorro \nCAPÍTULO I | RESSUSCITAÇÃO CARDIOPULMONAR (RCP) \n(Desfibrillation). A sequência de realização das manobras mais preconizadas é C -A-B. A \nmanobra correspondente a letra D (Desfibrillation), que corresponde à desfibrilação, deve ser \nrealizada, quando bem indicada, assim que possível^{5,8}. \n2.1. C (CIRCULATION - CIRCULAÇÃO) \nApós a identificação da PCR, é imprescindível o início imediato das compressões \ntorácicas. Alguns pilares importantes desta manobra são: \n• A vítima deve estar sobre

continua

superfície rígida \n• Deve ser realizada sobre o esterno, aplicando força sobre o tórax, com braços \nestendidos. \n• A compressão deve gerar uma profundidade no tórax de 5-6 cm \n• Permitir o retorno completo do tórax \n• A frequência deve ser de 100/120 por minuto \n• Minimizar as interrupções nas compressões ("no hands-off") \n• Trocar rapidamente o reanimador a cada 2 minutos']}]}

```
[ ] # 3. Pergunta do usuário
query = "O que fazer depois?"

# 3. Função para interagir com o usuário e obter respostas
result = qa_chain({"query": query})

print(result["result"])
```

⇒ Olá! Como assistente profissional especializado em tarefas de resposta a perguntas relacionadas a serviços hospitalares, estou aqui para ajudá-lo a responder à sua pergunta.

A pergunta é: Como devemos classificar um paciente com ICA de acordo com os perfis hemodinâmicos? E, mais especificamente, qual deve ser a conduta do médico na emergência de acordo com a classificação do paciente?

Para responder a essa pergunta, é importante lembrar que a ICA (Insuficiência Cardíaca Aguda) é um quadro clínico caracterizado por uma redução significativa da função cardíaca, que pode ser causada por várias razões, como doenças cardíacas pré-existentes, infarto agudo do miocárdio, doenças valvulares, entre outras.

Em termos de classificação, os perfis hemodinâmicos de pacientes com ICA variam de acordo com a gravidade do quadro e a presença de complicações. Geralmente, os pacientes com ICA são classificados em três perfis:

1. ICA estável, com função cardíaca preservada;
2. ICA moderada, com redução parcial da função cardíaca;
3. ICA grave, com redução significativa da função cardíaca e presença de complicações.

A conduta do médico na emergência depende da classificação do paciente. Em pacientes com ICA estável, o objetivo é monitorar a evolução do quadro e fornecer tratamento medicamentoso para melhorar a função cardíaca. Em pacientes com ICA moderada, o médico deve iniciar tratamento intensivo, incluindo monitoramento contínuo da pressão arterial, frequência cardíaca e saturação de oxigênio, além de fornecer medicamentos para melhorar a função cardíaca. Em pacientes com ICA grave, o médico deve realizar traqueostomia e iniciar tratamento de suporte ventilatório, além de fornecer medi-

continua

camentos para melhorar a função cardíaca e controlar a pressão arterial.

Lembre-se de que essas são apenas diretrizes gerais e que a conduta do médico deve ser personalizada para cada paciente, considerando fatores como a idade, sexo, histórico médico e presença de complicações.

Espero que isso tenha ajudado a responder à sua pergunta. Se você tiver mais perguntas ou precisar de esclarecimentos adicionais, sinta-se à vontade para perguntar!

```
[ ] result['source_documents']
```



```
[Document(metadata={'source': 'CardiologianaEmergencia.pdf', 'page': 232},  
page_content='4. Como devemos classificar um paciente com ICA de acordo com os perfis hemodinâmicos? \n5. Qual deve ser a conduta do médico na emergência de acordo com a classificação do \npaciente?'),
```

```
Document(metadata={'source': 'CardiologianaEmergencia.pdf', 'page': 109},  
page_content='2. Quais exames complementares você solicitaria diante da sua suspeita diagnóstica para esta \npaciente? \n3. Qual medicamento, possivelmente utilizado por essa paciente, que é considerado principal \nresponsável para este quadro? \n4. Como você conduziria esta paciente no departamento de Emergência? E a longo prazo?'),
```

```
Document(metadata={'source': 'CardiologianaEmergencia.pdf', 'page': 2},  
page_content='direitos para esta edição foram cedidos à Editora Amplla. \n \n \n \n \n \n \nISBN: 978-65-5381-116-4 \nDOI: 10.51859/amplla. ceg164.1123-0 \n \n \n \n \n \n \nEditora Amplla \nCampina Grande – PB – Brasil \ncontato@ampllaeditora.com.br \nwww.ampllaeditora.com.br \n \n \n \n \n \n \n \n \n \n \n \n \n \n \n \n2023'),
```

```
Document(metadata={'source': 'CardiologianaEmergencia.pdf', 'page': 7},  
page_content='que consideramos serem as mais pertinentes sobre cada tema. A ideia com estas \nperguntas é que o leitor seja capaz de respondê-las ao concluir a leitura. Já ao final \ndos capítulos, há fluxogramas que visam facilitar a compreensão e o raciocínio \nclínico diante das emergências. \nEste livro não possui o intuito de substituir a leitura de livros -texto \nconsagrados ou mesmo de ditar condutas, mas sim de ajudar na revisão de temas \npertinentes à prática médica.'])]
```

Pipeline do RAG com histórico

Agora vamos adicionar o histórico da conversa para que o LLM consiga continuar a conversa com o usuário sabendo o contexto.

Configurando o histórico

O código configura um chatbot que usa o histórico da conversa para entender melhor as perguntas do usuário em um contexto de serviços hospitalares, reformulando-as quando necessário e evitando marcar consultas duplicadas. Ele não responde às perguntas diretamente, apenas as prepara para serem respondidas por outro componente do sistema.

1. `contextualize_q_system_prompt`:

- » É uma string que define as instruções para o chatbot.
- » Imagine como se você estivesse dando instruções para um atendente de hospital sobre como lidar com as perguntas dos pacientes.
- » **Instruções específicas:**
 - » **Reformular a pergunta:** se a pergunta do usuário se referir a algo que já foi dito antes na conversa, o chatbot deve reescrevê-la de forma independente.
 - » **Não responder:** o chatbot não deve tentar responder à pergunta, apenas reformulá-la.
 - » **Estilo formal:** o chatbot deve usar uma linguagem formal e clara, como se estivesse falando com pacientes e visitantes de um hospital.
 - » **Verificar histórico:** o chatbot deve verificar se uma consulta já foi marcada na conversa. Se sim, ele deve informar que não há disponibilidade.

2. `contextualize_q_prompt`:

- » Define um modelo de prompt de chat usando a biblioteca `ChatPromptTemplate`.
- » **Componentes:**
 - » `("system", contextualize_q_system_prompt)`: define a mensagem do sistema, que são as instruções que vimos acima.
 - » `MessagesPlaceholder("chat_history")`: define um espaço para inserir o histórico da conversa.
 - » `("human", "")`: define um espaço para inserir a pergunta do usuário.

```
[ ] # 1. Aqui estamos importando 'MessagesPlaceholder', que é uma ferramenta
    # para "guardar espaço" para as mensagens anteriores do chat.
    # Isso ajuda a incluir o histórico da conversa dentro do processo, para que
    # o sistema tenha contexto.
```

continua

```
from langchain_core.prompts import MessagesPlaceholder
```

```
# 2. Aqui estamos criando um "prompt do sistema", que é um conjunto de ins-  
truções para o assistente virtual.
```

```
# Esse texto orienta o assistente a reformular as perguntas que o usuário  
faz no chat para que fiquem claras, sem depender do histórico da conversa.
```

```
# Além disso, o assistente deve adotar uma personalidade formal e profis-  
sional, direcionada a pacientes e visitantes de um hospital, e deve veri-  
ficar se consultas já foram marcadas no histórico.
```

```
# O assistente NÃO deve responder à pergunta, apenas reformulá-la, se ne-  
cessário.
```

```
contextualize_q_system_prompt = """Dada a história do chat e a última per-  
gunta do usuário, que pode referenciar o  
contexto na história do chat, reformule a pergunta de forma que possa ser  
entendida
```

```
sem a necessidade da história do chat. NÃO responda à pergunta, apenas re-  
formule-a
```

```
se necessário e, caso contrário, retorne-a como está.
```

```
O estilo de escrita é formal e claro, dirigido a pacientes e visitantes do  
hospital.
```

```
Você deve ficar atento ao histórico do chat para determinar se uma consul-  
ta já foi marcada, ou seja, se alguém marcou consulta antes da pessoa, você  
deve retornar que está indisponível."""
```

```
# 3. Nesta linha, estamos criando o 'contextualize_q_prompt', que define a  
estrutura do diálogo.
```

```
# Primeiro, temos a parte "system" com o 'contextualize_q_system_prompt',  
que contém as instruções gerais para o assistente.
```

```
# Depois, usamos 'MessagesPlaceholder' para reservar espaço para o histó-  
rico de mensagens anteriores do chat.
```

```
# Finalmente, temos a parte "human", que corresponde à última entrada do  
usuário ("{input}"), que é a pergunta feita pelo usuário que precisa ser  
reformulada.
```

```
# Essa estrutura completa serve para fornecer o contexto e a orientação  
necessários ao assistente virtual durante a interação no chat.
```

```
contextualize_q_prompt = ChatPromptTemplate.from_messages(  
    [  
        ("system", contextualize_q_system_prompt),  
        MessagesPlaceholder("chat_history"),  
        ("human", "{input}"),  
    ]  
)
```

Cria um objeto **history_aware_retriever** usando a função **create_history_aware_retriever**. O objetivo é criar um sistema que busca informações relevantes para responder à pergunta do usuário, levando em consideração o histórico da conversa.

» **Componentes:**

- » **GROQ_LLM**: LLM que será usado para entender e processar as perguntas.
- » **retriever**: sistema de busca que encontra informações relevantes no banco de dados do hospital e vai usar para substituir o **context**.
- » **contextualize_q_prompt**: O modelo de prompt que definimos anteriormente.

```
[ ] from langchain.chains import create_history_aware_retriever

history_aware_retriever = create_history_aware_retriever(
    GROQ_LLM, retriever, contextualize_q_prompt
)
```

Prompt da Perguntas e Respostas

Este trecho de código define como um chatbot deve responder a perguntas sobre serviços hospitalares, usando informações contextuais. Vamos entender cada parte:

- » **qa_system_prompt**: string define as instruções para o modelo de linguagem, direcionando-o a usar o contexto fornecido para responder às perguntas de forma concisa. Se a resposta for desconhecida, o modelo é instruído a declarar isso explicitamente.
- » **ChatPromptTemplate.from_messages**: cria um modelo de prompt para interações de chat, integrando mensagens do sistema, histórico de chat e entrada do usuário, com espaços reservados permitindo a inserção dinâmica de conteúdo.

```
[ ] ### Answer question ###

qa_system_prompt = """Atue com a personalidade de um assistente profissional especializado em tarefas de resposta a perguntas relacionadas a serviços hospitalares.

Use os seguintes pedaços de contexto para responder à pergunta do usuário. Se você não souber a resposta, diga apenas que não tem informações suficientes para responder.
```

continua


```
Contexto: {context}"""
qa_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", qa_system_prompt),
        MessagesPlaceholder("chat_history"),
        ("human", "{input}"),
    ]
)
```

Cria uma cadeia de perguntas e respostas que combina um modelo de linguagem grande (LLM) com um prompt específico para responder perguntas com base em documentos. Vamos entender cada parte:

1. **question_answer_chain:**

» É uma variável que armazenará a cadeia de perguntas e respostas criada. Pense nisso como uma linha de produção que recebe uma pergunta e gera uma resposta.

2. **create_stuff_documents_chain:**

» É uma função que cria a cadeia de perguntas e respostas. Imagine essa função como a fábrica que monta a linha de produção.

3. **GROQ_LLM:**

» É o modelo de linguagem grande que será usado para entender e responder às perguntas.

4. **qa_prompt:**

» É o prompt que define como o modelo de linguagem deve se comportar ao responder perguntas, que já explicamos anteriormente.

```
[ ] from langchain.chains.combine_documents import create_stuff_documents_chain

question_answer_chain = create_stuff_documents_chain(GROQ_LLM, qa_prompt)
```

O código cria um sistema RAG que funciona da seguinte forma:

1. O usuário faz uma pergunta.
2. O **history_aware_retriever** busca documentos relevantes para a pergunta, considerando o histórico da conversa.
3. Os documentos recuperados são passados para a **question_answer_chain**.
4. A **question_answer_chain** usa o modelo de linguagem grande e o prompt para gerar a resposta final com base nos documentos.

```
[ ] from langchain.chains import create_retrieval_chain

rag_chain = create_retrieval_chain(history_aware_retriever, question_
answer_chain)
```

A função **get_session_history** serve para gerenciar o histórico de conversas de diferentes sessões de chat. Ela verifica se já existe um histórico para a sessão especificada e, caso não exista, cria um novo histórico vazio. Em seguida, a função retorna o histórico da sessão, seja ele existente ou recém-criado.

Essa função é importante para manter o contexto da conversa em cada sessão de chat, permitindo que o chatbot acesse as mensagens anteriores e forneça respostas mais relevantes e coerentes.

Por exemplo, se um usuário fizer uma pergunta sobre um tópico que já foi discutido anteriormente na mesma sessão, o chatbot poderá usar o histórico da conversa para fornecer uma resposta mais completa e contextualizada.

```
[ ] from langchain_core.chat_history import BaseChatMessageHistory
    from langchain_community.chat_message_histories import ChatMessageHistory

### Histórico do Chat ###
store = {}

def get_session_history(session_id: str) -> BaseChatMessageHistory:
    if session_id not in store:
        store[session_id] = ChatMessageHistory()
    return store[session_id]
```

Este código transforma a cadeia RAG em uma cadeia RAG conversacional, adicionando a capacidade de gerenciar o histórico da conversa. Isso significa que o chatbot poderá:

- » Lembrar das perguntas e respostas anteriores na mesma sessão.
- » Usar o histórico da conversa para fornecer respostas mais contextuais e relevantes.
- » Adaptar suas respostas com base nas interações anteriores com o usuário.

Por exemplo, se o usuário perguntar "Quais são os horários de visita?", o chatbot poderá responder com informações gerais sobre os horários de visita. Se o usuário, em seguida, perguntar "E para a UTI?", o chatbot poderá usar o histórico da conversa para entender que o usuário está perguntando sobre os horários de visita da UTI e fornecer uma resposta mais específica.

Em outras palavras, a cadeia RAG conversacional torna o chatbot mais inteligente e capaz de manter uma conversa mais natural e fluida com o usuário, lembrando do contexto e adaptando suas respostas com base nas interações anteriores.

```
[ ] from langchain_core.runnables.history import RunnableWithMessageHistory

conversational_rag_chain = RunnableWithMessageHistory(
    rag_chain,
    get_session_history,
    input_messages_key="input",
    history_messages_key="chat_history",
    output_messages_key="answer",
)
```

Este código envia a primeira pergunta "Como usar um desfibrilador em situações de emergência?" para a cadeia RAG conversacional, iniciando uma nova sessão de conversa com o ID "gml_nlp". A resposta do chatbot será armazenada na variável **first_question**.

A partir deste ponto, a cadeia RAG conversacional está pronta para receber novas perguntas e usar o histórico da conversa (incluindo a primeira pergunta e sua resposta) para fornecer respostas mais contextuais e relevantes.

```
[ ] first_question = conversational_rag_chain.invoke(  
    {"input": f"Como usar um desfibrilador em situações de emergência?"},  
    config={"configurable": {"session_id": "gml_nlp"}},  
)
```

```
[ ] print(first_question['answer'])
```

➡ Um desfibrilador externo automático (DEA) é um dispositivo fácil de usar que pode ser utilizado por pessoas leigas em situações de emergência. Aqui estão os passos para usar um DEA:

1. Verifique se a vítima está inconsciente ou em parada cardíaca (PCR). Se a vítima está consciente, procure ajuda médica imediatamente.
2. Verifique se o DEA está carregado e funcional. Certifique-se de que os eletrodos estejam limpos e secos.
3. Coloque a vítima em posição de supino (de barriga para cima) e tente estabilizar a cabeça e o pescoço.
4. Retire os eletrodos do DEA e coloque-os nas posições recomendadas (um na região apical e outro na região infraclavicular direita).
5. Pressione o botão de "Shock" (choque) se o DEA indicar que o ritmo cardíaco é "chocável" (taquicardia ventricular sem pulso ou fibrilação ventricular).
6. Após a desfibrilação, retorne a sequência de compressões torácicas (SBV) e realize novas compressões torácicas em um ritmo de 100-120 por minuto até o chegar da equipe médica.
7. Certifique-se de que o DEA esteja funcionando corretamente e faça novas compressões torácicas após a desfibrilação.

Lembre-se de que um DEA não substitui a ajuda médica profissional. Após a desfibrilação, procure imediatamente ajuda médica e siga as instruções da equipe médica.

É importante mencionar que, segundo o contexto fornecido, a carga recomendada para um DEA monofásico é de 360J, enquanto para um DEA bifásico é recomendada a carga preconizada pelo fabricante do aparelho, geralmente entre 120-200J, mas se dúvida, utilize 200J.

```
[ ] print(first_question['chat_history'])
```

➡ []

Este código envia a segunda pergunta "O que fazer depois?" para a cadeia RAG conversacional, dando continuidade à sessão de conversa "gml_nlp". A resposta do chatbot será armazenada na variável **second_question**.

A chave aqui é que a segunda pergunta depende do contexto da primeira pergunta. O chatbot precisa lembrar que a primeira pergunta era sobre como usar um desfibrilador para entender que "O que fazer depois?" se refere aos passos seguintes após o uso do desfibrilador.

Graças ao **RunnableWithMessageHistory** e ao **get_session_history**, a cadeia RAG conversacional consegue acessar o histórico da conversa e entender o contexto da segunda pergunta, fornecendo uma resposta mais precisa e relevante.

```
[ ] second_question = conversational_rag_chain.invoke(  
    {"input": f"O que fazer depois?"},  
    config={"configurable": {"session_id": "gml_nlp"}},  
)
```

```
[ ] print(second_question['answer'])
```

➡ Depois de realizar a desfibrilação com o DEA, é importante seguir os passos do Suporte Básico de Vida (SBV) para manter a vítima viável e o Sistema Nervoso Central (SNC) o mais intacto possível. Aqui estão os passos subsequentes:

1. Realize compressões torácicas (C) contínuas em um ritmo de 100-120 por minuto. É importante manter a compressão torácica até a chegada da equipe médica.
2. Verifique a patenteira (A - Airway) e assegure-se de que a via aérea esteja livre. Se necessário, realize uma intubação traqueal.
3. Verifique a respiração (B - Breathing) e assegure-se de que a vítima esteja recebendo ar suficiente. Se necessário, realize ventilação artificial.
4. Verifique a atividade cardíaca (D - Defibrilação) e realize novas compressões torácicas após a desfibrilação.

Em resumo, após a desfibrilação, o objetivo é manter a vítima viável e o SNC o mais intacto possível até a chegada da equipe médica. Abstenha-se de realizar qualquer outro procedimento médico até a equipe médica chegar.

```
[ ] second_question['chat_history']
```

continua

```
→ [HumanMessage(content='Como usar um desfibrilador em situações de emergência?', additional_kwargs={}, response_metadata={}),
```

```
    AIMessage(content='Um desfibrilador externo automático (DEA) é um dispositivo fácil de usar que pode ser utilizado por pessoas leigas em situações de emergência. Aqui estão os passos para usar um DEA:\n\n1. Verifique se a vítima está inconsciente ou em parada cardíaca (PCR). Se a vítima está consciente, procure ajuda médica imediatamente.\n2. Verifique se o DEA está carregado e funcional. Certifique-se de que os eletrodos estejam limpos e secos.\n3. Coloque a vítima em posição de supino (de barriga para cima) e tente estabilizar a cabeça e o pescoço.\n4. Retire os eletrodos do DEA e coloque-os nas posições recomendadas (um na região apical e outro na região infraclavicular direita).\n5. Pressione o botão de "Shock" (choque) se o DEA indicar que o ritmo cardíaco é "chocável" (taquicardia ventricular sem pulso ou fibrilação ventricular).\n6. Após a desfibrilação, retorne a sequência de compressões torácicas (SBV) e realize novas compressões torácicas em um ritmo de 100-120 por minuto até o chegar da equipe médica.\n7. Certifique-se de que o DEA esteja funcionando corretamente e faça novas compressões torácicas após a desfibrilação.\n\nLembre-se de que um DEA não substitui a ajuda médica profissional. Após a desfibrilação, procure imediatamente ajuda médica e siga as instruções da equipe médica.\n\nÉ importante mencionar que, segundo o contexto fornecido, a carga recomendada para um DEA monofásico é de 360J, enquanto para um DEA bifásico é recomendada a carga preconizada pelo fabricante do aparelho, geralmente entre 120-200J, mas se dúvida, utilize 200J.', additional_kwargs={}, response_metadata={})]
```

```
[ ] second_question['context']
```

```
→ [Document(metadata={'source': 'CardiologianaEmergencia.pdf', 'page': 11}, page_content='retomadas, e o DEA indicou 3 choques ao longo de 10 minutos, todos seguidos de CTC. \nApós o quarto choque, a vítima voltou a respirar ainda que inconsciente, não apresentando \nmais indicação de desfibrilação devido à retomada do ritmo sinusal. \n2. SUPORTE BÁSICO DE VIDA (SBV) \n0 Suporte Básico de Vida (SBV) é o protocolo de atendimento inicial às vítimas de parada \ncardiorrespiratória (PCR), que pode ser iniciado no ambiente extra-hospitalar, no qual se \nestabelece o reconhecimento e a realização de manobras. O grande objetivo do SBV é manter o \npaciente viável e com o Sistema Nervoso Central (SNC) o mais intacto possível. Com a chegada \nda equipe médica equipada e treinada, terá início o Suporte Avançado de Vida (SAV)5,8. \n0 atendimento inicial é feito pela pessoa mais próxima e inicia com uma vítima suspeita \nde PCR. De início, após identificação de vítima suspeita de PCR, é preciso verificar se o local do'),
```

```
    Document(metadata={'source': 'CardiologianaEmergencia.pdf', 'page': 11}, page_content='realizada por pessoas leigas minimamente treinadas para tal, com auxílio dos Desfibriladores \nExternos Automáticos (DEA)5,8. \nA partir disso, podem ser iniciadas as manobras de SBV, que consistem em quatro \npilares, sendo representadas por meio de letras: C (ciculation), A (airway), B (Breathing) e D'),
```

continua


```
Document(metadata={'source': 'CardiologianaEmergencia.pdf', 'page': 13},
page_content='13 Cardiologia na Emergência: Guia para o Pronto-Socorro \
nCAPÍTULO I | RESSUSCITAÇÃO CARDIOPULMONAR (RCP) \nfeita sem proteção
(boca a boca ou boca nariz) ou com proteção (uso de dispositivos e válv-
vulas, \ntais como dispositivos com válvula unidirecional, máscaras ou
bolsa-máscara)5,8. \n2.4. D (DEFIBRILLATION - DESFIBRILAÇÃO) \nEste pro-
cedimento pode ser realizado assim que possível, quando corretamente \nin-
dicado, ou seja, na presença de ritmos de parada "chocáveis" (Taquicardia
Ventricular s/ \npulso e Fibrilação Ventricular). Existe o cardiodesfibri-
lador manual, utilizado em ambie nte \nintra-hospitalar, e o Desfibrilador
Externo Automático (DEA), que pode ser utilizado por leigos \nem ambiente
extra -hospitalar, já que identifica se o ritmo é "chocável" ou "não -cho-
cável" de \nforma automática, indicando ou não a desfibrilação, contri-
buindo para aumento das chances de \nsobrevivência da vítima. A posição
tradicional recomendada dos eletrodos é um na região apical'),
```

```
Document(metadata={'source': 'CardiologianaEmergencia.pdf', 'page': 11},
page_content='O atendimento inicial é feito pela pessoa mais próxima e
inicia com uma vítima suspeita \nde PCR. De início, após identificação de
vítima suspeita de PCR, é preciso verificar se o local do \natendimento é
seguro para o socorrista e para a vítima. Feito isso, deve -se checar a \
nresponsividade do paciente, realizando estímulo auditivo (chamar pelo
paciente) e manual \n(realizar toques na clavícula, por exemplo), e a res-
piração da vítima. A associação de \nirresponsividade com respiração anor-
mal (apneia ou agônica) configura o quadro de PCR. A \nchecagem de pulso
em grandes artérias é opcional para socorristas leigos, se profissional
de \nsaúde, a checagem não pode ultrapassar 10 segundos. Constatada a PCR,
o próximo pass o é \nacionar o serviço de emergência e pedir que tragam
o desfibrilador. A desfibrilação pode ser \nrealizada por pessoas leigas
minimamente treinadas para tal, com auxílio dos Desfibriladores \nExternos
Automáticos (DEA)5,8.')]
```

```
[ ] second_question['input']
```

➡ O que fazer depois?

Desafios Práticos

Os desafios a seguir exploram as aplicações práticas da técnica Retrieval Augmented Generation (RAG) na construção de sistemas inteligentes de perguntas e respostas e assistentes virtuais. Cada desafio apresenta um problema específico e um conjunto de instruções para guiar a implementação de uma solução baseada em RAG. As soluções serão avaliadas com base em métricas relevantes para cada contexto, como acurácia, relevância, personalização e eficiência.

Desafio 1:

Objetivo: criar um sistema de perguntas e respostas usando RAG.

Instruções: desenvolver um modelo que integre recuperação de informações e geração de respostas com base em um conjunto de documentos específicos.

Avaliação: medir a acurácia e a relevância das respostas geradas.

Desafio 2:

Objetivo: criar um sistema de perguntas e respostas usando RAG.

Instruções: utilizar RAG para personalizar recomendações baseadas no histórico de interações do usuário.

Avaliação: analisar a personalização e a satisfação do usuário com as recomendações.

Desafio 3:

Objetivo: implementar um assistente virtual para suporte ao cliente que utilize RAG para fornecer respostas precisas.

Instruções: integrar um banco de dados de FAQs com um modelo de RAG para responder a questões técnicas complexas.

Avaliação: avaliar a eficiência e precisão das respostas em situações de teste.



SAIBA MAIS...

Para aprofundar seu conhecimento sobre RAG, explore os seguintes recursos:

✿ Artigos e publicações:

✿ *"Retrieval-augmented generation for large language models: A survey"*: apresenta uma visão geral bem detalhada dos principais trabalhos utilizando RAG.

✿ Pesquise por artigos sobre *"Retrieval-Augmented Generation"*, *"RAG"*, *"Knowledge Retrieval"* e tópicos relacionados no arXiv.org e em outras plataformas de publicação científica. Mantenha-se atualizado sobre as pesquisas mais recentes e os avanços na área.

✿ Bibliotecas e ferramentas:

✿ *RAGFlow*: é um *framework* para construção de pipelines RAG de código aberto baseado em compreensão profunda de documentos. Ele oferece um fluxo de trabalho RAG simplificado para empresas de qualquer porte, combinando LLMs para fornecer recursos de resposta a perguntas precisas, com base em citações bem fundamentadas de vários dados formatados.

✿ *LangChain*: framework para desenvolvimento de aplicações que utilizam LLMs. O *LangChain* possui diversas funcionalidades que simplificam a implementação de RAG e outras técnicas avançadas.

✿ Artigos no Medium:

✿ [RAG LLM Best Practices](#)

✿ [RAG - Retrieval-Augmented Generation](#)

✿ [A first intro to Complex RAG \(Retrieval Augmented Generation\)](#)

✿ [Implementing Retrieval Augmented Generation \(RAG\): A Hands-On Guide!](#)



PARA RELEMBRAR...

✿ Nessa Unidade, exploramos o conceito de RAG, uma técnica que permite aos LLMs acessar e utilizar informações externas em tempo real. Seguem os principais conceitos e termos:

✿ RAG: técnica que combina a geração de texto dos LLMs com a recuperação de informações de fontes externas, como bancos de dados ou documentos. O RAG permite que os LLMs acessem informações atualizadas e relevantes, tornando-os mais precisos e contextualmente relevantes.

✿ Indexação: processo de organizar a base de conhecimento externa para facilitar a recuperação de informações. Envolve a divisão dos documentos em *chunks* (pedaços dos documentos) e a criação de representações vetoriais (*embeddings*) para cada *chunk*.

✿ Recuperação: etapa em que o sistema busca os *chunks* mais relevantes para a consulta do usuário, com base na similaridade entre a representação vetorial da consulta e os *embeddings* dos *chunks*.

✿ Geração: processo em que o LLM gera uma resposta, utilizando a consulta do usuário e as informações recuperadas como contexto.

✿ Arquiteturas RAG: diferentes abordagens para implementar RAG, variando em complexidade e desempenho. As principais arquiteturas são: *Naive RAG*, *Advanced RAG* e *Modular RAG*.

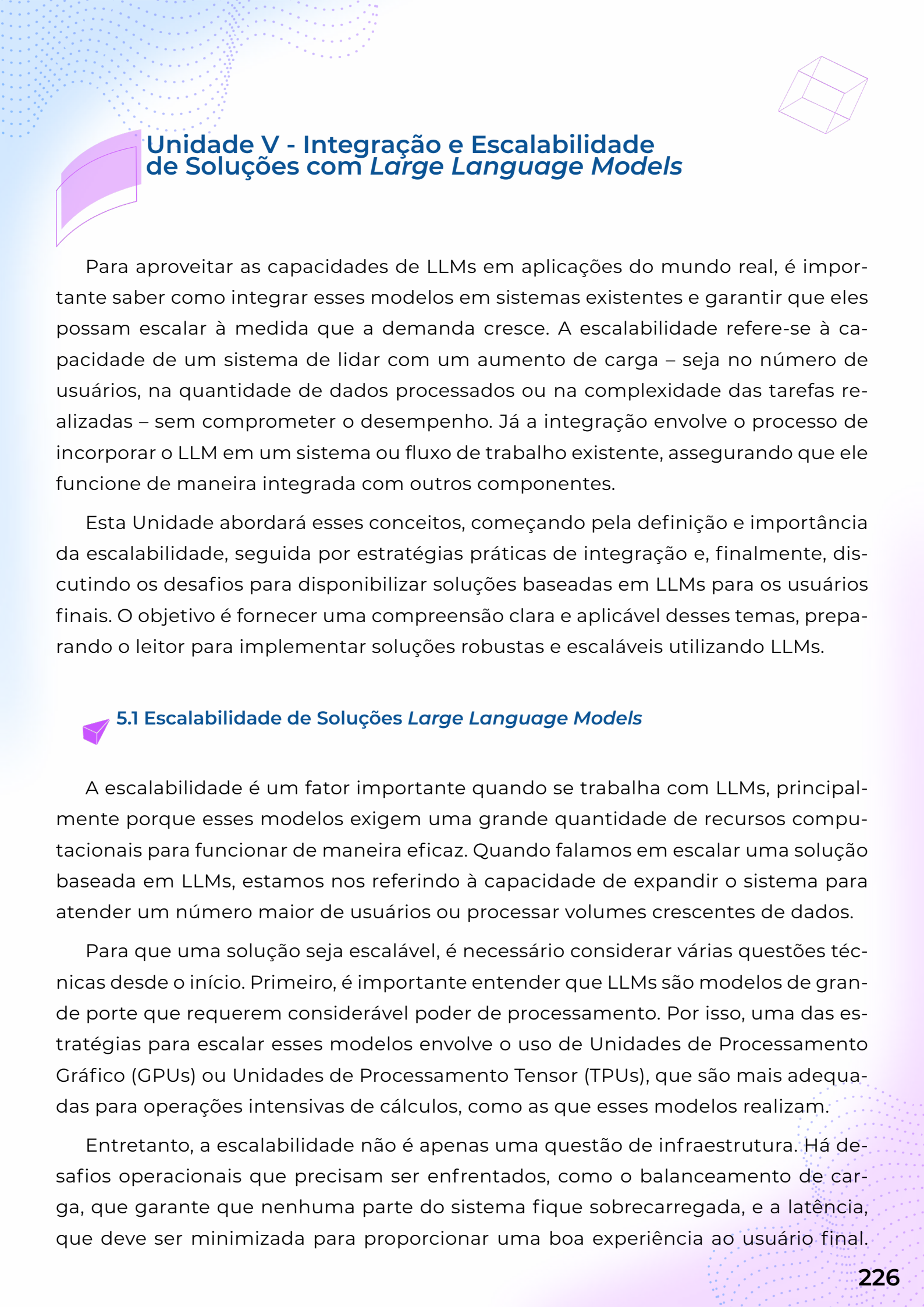
✿ *Naive RAG*: arquitetura mais simples, onde a recuperação e geração são realizadas de forma linear.

✿ *Advanced RAG*: incorpora técnicas mais avançadas de recuperação e geração, como *embeddings* semânticos e *re-ranking*.

✿ *Modular RAG*: divide o sistema RAG em módulos independentes e reutilizáveis, oferecendo maior flexibilidade e personalização.

✿ *Chunks*: pedaços menores de texto em que os documentos são divididos durante a

Unidade V
**Integração e
Escalabilidade de
Soluções com LLMs**



Unidade V - Integração e Escalabilidade de Soluções com *Large Language Models*

Para aproveitar as capacidades de LLMs em aplicações do mundo real, é importante saber como integrar esses modelos em sistemas existentes e garantir que eles possam escalar à medida que a demanda cresce. A escalabilidade refere-se à capacidade de um sistema de lidar com um aumento de carga – seja no número de usuários, na quantidade de dados processados ou na complexidade das tarefas realizadas – sem comprometer o desempenho. Já a integração envolve o processo de incorporar o LLM em um sistema ou fluxo de trabalho existente, assegurando que ele funcione de maneira integrada com outros componentes.

Esta Unidade abordará esses conceitos, começando pela definição e importância da escalabilidade, seguida por estratégias práticas de integração e, finalmente, discutindo os desafios para disponibilizar soluções baseadas em LLMs para os usuários finais. O objetivo é fornecer uma compreensão clara e aplicável desses temas, preparando o leitor para implementar soluções robustas e escaláveis utilizando LLMs.

5.1 Escalabilidade de Soluções *Large Language Models*

A escalabilidade é um fator importante quando se trabalha com LLMs, principalmente porque esses modelos exigem uma grande quantidade de recursos computacionais para funcionar de maneira eficaz. Quando falamos em escalar uma solução baseada em LLMs, estamos nos referindo à capacidade de expandir o sistema para atender um número maior de usuários ou processar volumes crescentes de dados.

Para que uma solução seja escalável, é necessário considerar várias questões técnicas desde o início. Primeiro, é importante entender que LLMs são modelos de grande porte que requerem considerável poder de processamento. Por isso, uma das estratégias para escalar esses modelos envolve o uso de Unidades de Processamento Gráfico (GPUs) ou Unidades de Processamento Tensor (TPUs), que são mais adequadas para operações intensivas de cálculos, como as que esses modelos realizam.

Entretanto, a escalabilidade não é apenas uma questão de infraestrutura. Há desafios operacionais que precisam ser enfrentados, como o balanceamento de carga, que garante que nenhuma parte do sistema fique sobrecarregada, e a latência, que deve ser minimizada para proporcionar uma boa experiência ao usuário final.

Também é importante considerar os custos associados à escalabilidade, já que o aumento de capacidade muitas vezes vem acompanhado de maiores despesas com hardware, energia e manutenção.



5.2 Capacidade Computacional para Adoção de *Large Language Models* nas Empresas

A adoção de LLMs em soluções empresariais exige um planejamento da infraestrutura computacional. Dimensionar incorretamente pode levar a custos excessivos ou performance inadequada, prejudicando o retorno do investimento. Esta seção aborda aspectos fundamentais para essa avaliação, incluindo métricas de capacidade computacional, *benchmarking* de desempenho e custos operacionais, bem como as principais estratégias para otimizar a infraestrutura necessária para adotar LLMs em empresas de diferentes portes e setores.



5.2.1 Métricas de Capacidade Computacional para *Large Language Models*

A capacidade computacional para suportar LLMs pode ser medida em termos de FLOPs, consumo de memória das GPUs e latência de inferência.

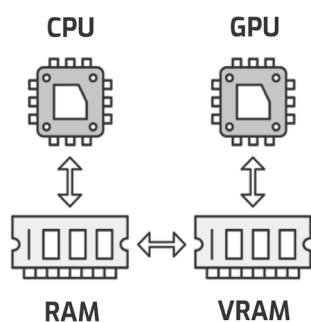
FLOPs são uma métrica central para quantificar a complexidade computacional dos modelos e, conseqüentemente, o hardware necessário. Estimar o número de FLOPs pode ajudar a dimensionar o número de GPUs necessárias, o que é especialmente útil para empresas que pretendem rodar LLMs *on-premise*. O modelo GPT-3, por exemplo, possui 175 bilhões de parâmetros e necessita de centenas de petaflops para treinamento eficiente, enquanto modelos menores podem ser mais acessíveis em termos computacionais (Shoeybi *et al.*, 2019).

A **latência** é uma métrica importante para os usuários da sua aplicação, especialmente em aplicações de atendimento ao cliente ou e-commerce em que o seu cliente não quer esperar dezenas de segundos ou minutos por uma resposta. Estudos mostram que a latência aumenta significativamente com o tamanho do modelo e a quantidade de dados processados (Brown *et al.*, 2020), sugerindo que empresas com alto volume de transações devem considerar arquiteturas otimizadas para inferência. Ou seja, as empresas devem se preocupar com a qualidade da resposta das LLMs (acurácia da inferência) e também com o tempo de resposta das LLMs (latência). Modelos menores tendem a ter uma latência menor e uma acurácia pior do que modelos grandes, mas podem atender bem a cenários de usuários reais sem demandar o mesmo nível de capacidade de memória e processamento.

Outra métrica muito importante é o quanto de **memória de GPUs** você precisa para ter uma LLM rodando na sua organização. Em LLMs, a memória de GPU é utilizada para armazenar tanto os parâmetros do modelo quanto os dados temporários necessários para realizar operações de inferência e treinamento. À medida que o número de parâmetros aumenta — como nos modelos de bilhões de parâmetros, como o GPT-3 —, a demanda por memória cresce, exigindo GPUs com capacidades de VRAM elevadas.

Video Random Access Memory (VRAM) é uma memória de alta velocidade diretamente integrada às GPUs, permitindo que ela acesse dados rapidamente, sem precisar recorrer constantemente à memória da CPU. Na Figura 38, é apresentado um exemplo da integração da memória da GPU e da CPU. Quando a GPU precisa de dados que residem na memória da CPU, esses dados precisam ser copiados para a VRAM por meio de um barramento, representado pelas setas bidirecionais. Esse processo de cópia introduz, geralmente, um atraso no acesso aos dados. Por isso, os modelos são carregados diretamente na VRAM e quanto maior a quantidade de VRAM, mais dados e modelos maiores uma GPU pode processar simultaneamente.

Figura 38 - Arquitetura simplificada de memória da *Central Processing Units* (CPU) e *Graphics Processing Units* (GPU)



Fonte: adaptada de [Pickle \(2023\)](#).

Também é possível dividir um grande LM entre várias GPUs, mas ainda assim, configurações de memória inadequadas podem levar a problemas de **falta de memória**, interrompendo o processamento. Estratégias como quantização são usadas para otimizar o consumo de memória, possibilitando que LLMs sejam executados em GPUs com menores capacidades sem comprometer muito o desempenho.

5.2.2 Implantação do *Large Language Model*

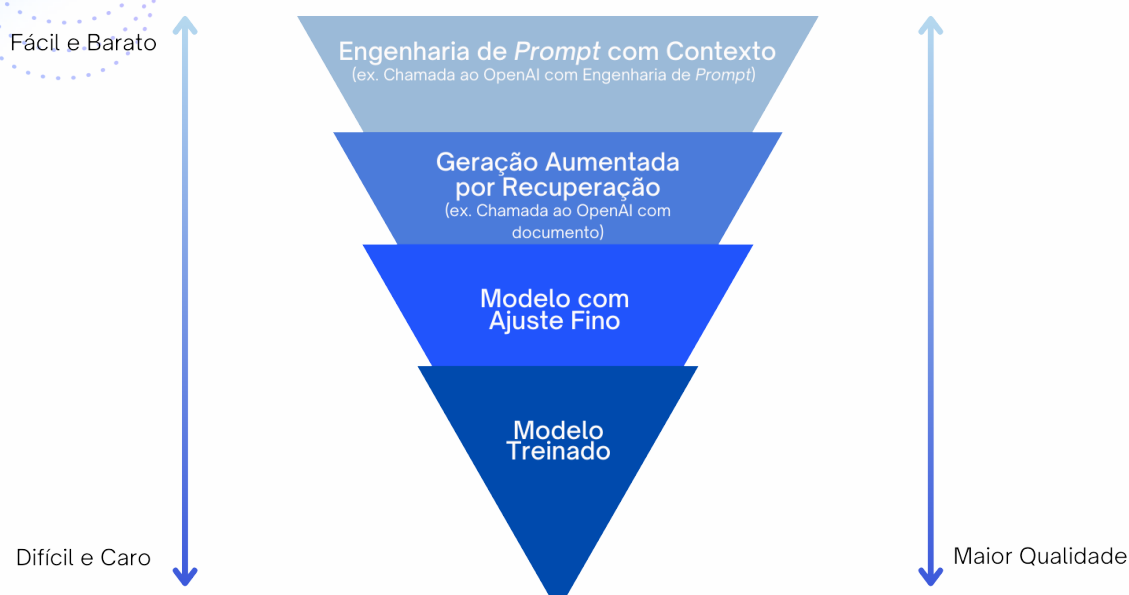
Uma pergunta comum entre profissionais que estão iniciando a implantação de LLMs nas empresas é: "Como mensurar a capacidade computacional necessária para adotar LLMs em soluções na minha organização?". Você deve considerar diversos fatores que influenciam diretamente no desempenho, nos custos e na viabilidade dessa tecnologia.

Na Figura 39, é apresentada uma visão geral de quatro abordagens para colocar em produção soluções integradas com LLMs, com uma relação entre custo/dificuldade e qualidade dos resultados.

- » **Engenharia de prompt com contexto:** essa é a opção mais acessível e simples, onde *prompts* bem estruturados são enviados para APIs (como OpenAI) para melhorar a qualidade das respostas.
- » **RAG:** acrescenta documentos relevantes como contexto ao LLMs para aumentar a precisão das respostas.
- » **Modelos com ajuste fino:** realiza uma customização específica para tarefas específicas, com treinamento em dados próprios, melhorando a qualidade mas aumentando custos em relação a técnicas sem ajuste fino.
- » **Treinamento de modelos:** é a opção mais cara e complexa, com o modelo treinado inteiramente do zero, oferecendo total controle da qualidade do modelo.

Você pode iniciar sua solução utilizando engenharia de prompt e APIs comerciais, como o GPT-4 da OpenAI e o Gemini da Google. O uso de APIs comerciais permite acesso a LLMs de alto desempenho sem a necessidade de infraestrutura própria. Essa solução terceirizada, como as oferecidas pelo OpenAI e Gemini, minimiza custos de manutenção e hardware, garantindo alta escalabilidade. É ideal para empresas que necessitam de facilidade de implementação e não requerem controle total sobre o modelo. No entanto, o custo operacional pode aumentar conforme o volume de uso, e as possibilidades de customização e ajuste fino são limitadas comparado com modelos treinados internamente. De acordo com a Gartner, teremos um aumento de mais de 30% até 2026 na demanda por APIs vindo da IA e de ferramentas que utilizam LLMs.

Figura 39 - Quatro abordagens para colocar *Large Language Models* em produção



Fonte: adaptada de [AWS \(2024\)](#).

Quando a empresa necessita de integração com bases de dados internas da empresa ou de informações que não estão na base de dados do LLM, é normal evoluir para a abordagem do RAG. Aqui, a implantação inclui uma camada de recuperação de contexto. O sistema busca informações relevantes em documentos ou bases de dados e as fornece como contexto ao modelo. Você pode continuar utilizando APIs, como a OpenAI para acesso ao LLM, mas com o RAG, o modelo tem acesso a documentos contextuais adicionais, melhorando a precisão das respostas. Essa configuração requer integração com bases de dados e um mecanismo de recuperação, elevando o custo, mas ainda sendo mais acessível do que a opção de ajuste fino do modelo.

O **RAG** é mais complexo que a engenharia de prompt porque ele envolve uma etapa extra de recuperação de dados externos para contextualizar as respostas do modelo, garantindo que ele tenha informações atualizadas e específicas ao responder.

Na engenharia de prompt, o modelo gera respostas apenas com base em seu treinamento prévio, sem acesso a informações adicionais. Essa complexidade adicional exige mais infraestrutura e um sistema de indexação e recuperação eficiente, aumentando o custo e o esforço de implementação. Em contrapartida, o RAG melhora a precisão, pois o modelo responde com base em informações contextualizadas e atualizadas, ideal para aplicações que precisam de conteúdo específico e dinâmico.

O ajuste fino (*fine-tuning*) exige treinar o modelo com dados específicos do domínio da empresa, aumentando a qualidade e a relevância das respostas, mas também o custo. Aqui, a organização pode optar por fazer ajuste fino do modelo usando uma GPU própria, serviços de terceiros (como [Together.ai](#)), ou até [fine-tuning-as-a-service](#) (como o da OpenAI). Cada opção vem com um custo maior do que o RAG, pois envolve computação adicional e um processo de ajuste do modelo, mas resulta em um modelo mais adaptado às necessidades específicas da empresa.

O ajuste fino é indicado quando a empresa precisa que o modelo produza respostas muito específicas para seu domínio ou área de atuação. Os treinamentos podem exigir horas de processamento em GPUs, elevando custos com energia e hardware, além de exigir *expertise* para gerenciar o treinamento e avaliar o desempenho. Para simplificar, empresas podem optar pelo **Fine-Tuning-as-a-Service (FTaaS)** oferecido por plataformas como a OpenAI, que facilitam o processo ao realizar o ajuste fino do modelo na infraestrutura da própria plataforma.

O **FTaaS** reduz a necessidade de manutenção de hardware, mas os custos ainda são relativamente elevados devido ao uso da infraestrutura de terceiros e ao processamento extra necessário para personalizar o modelo com dados específicos do cliente. Esse ajuste envolve infraestrutura avançada, como o uso contínuo de GPUs para calibrar os parâmetros do modelo com os dados fornecidos pela empresa. Na OpenAI, por exemplo, você pode criar seus próprios modelos personalizados fazendo *fine-tuning* nos modelos base com seus dados de treinamento. Após ajustar um modelo, você será cobrado apenas pelos *tokens* usados nas solicitações a esse modelo.

A última opção é utilizar LLMs de código aberto, tal como [Llama](#) da Meta (Meta, 2024). Neste caso, você pode possuir, geralmente, quatro opções:

- » **Modelo rodando na infraestrutura local:** a empresa fica responsável por custear GPUs para rodar um LLM internamente. As GPUs podem ser caras de adquirir e manter e essa opção é recomendada para empresas que já possuem recursos computacionais ou que precisam de segurança e controle completos sobre os dados. Por exemplo, é uma opção muito utilizada em órgãos públicos que desejam total controle sobre os dados que estão indo para o modelo.
- » **Modelo rodando em plataformas:** você pode utilizar plataformas como Together.ai para acessar modelos de código aberto via API. Essa opção é mais prática para empresas que buscam uma solução escalável e querem evitar os desafios de manutenção e custo de hardware. No entanto, o controle sobre os dados pode ser mais limitado comparado a uma infraestrutura própria.
- » **Modelo com ajuste fino rodando na infraestrutura local:** você pode fazer um ajuste fino do modelo de código aberto internamente e rodar a inferência nas GPUs locais dentro da organização. O processo de ajuste fino do modelo exige

mais poder computacional e horas de GPU para ajustar o modelo aos dados específicos da empresa. Isso oferece respostas mais relevantes, mas requer investimento em infraestrutura e equipe técnica para gerenciar o treinamento e monitoramento do modelo.

- » **Treinamento do próprio modelo:** treinar um modelo do zero é a opção mais complexa e cara. Exige uma quantidade massiva de dados, alta capacidade computacional (como *clusters* de GPUs ou TPUs) e *expertise* técnica. Essa abordagem permite criar um modelo completamente personalizado e controlado pela empresa, mas é viável apenas para organizações com um orçamento elevado e uma necessidade específica que não pode ser atendida por modelos pré-treinados ou ajustados. A OpenAI, por exemplo, treinou o GPT-4 utilizando milhares de GPUs, e estima-se que o custo de computação para um modelo dessa escala seja de **mais de 100 milhões de dólares** (Knight, 2023).



5.2.3 Requisitos de GPU para Implantação de um *Large Language Models* em uma Infraestrutura Local

O primeiro aspecto a considerar é o tamanho do modelo em termos de número de parâmetros. Modelos maiores, como o Llama-3-70B, com 70 bilhões de parâmetros, exigem aproximadamente 150 GB de memória. De modo geral, para modelos acima de 10 bilhões de parâmetros, são recomendadas GPUs com ao menos 24 GB de VRAM por unidade, como as NVIDIA A100 ou RTX 4090, pois modelos desse porte ultrapassam rapidamente a capacidade de GPUs menores, como as de 8 a 16 GB de VRAM.

A Equação 1 ajuda a prever a quantidade de VRAM necessária para rodar o LLM, levando em conta o tamanho do modelo e as necessidades operacionais adicionais:

1. **Parâmetros (P):** representa o número total de parâmetros do modelo, que são valores ajustáveis aprendidos durante o treinamento. Quanto mais parâmetros, mais memória é necessária para armazená-los. Por exemplo, o modelo Llama-3-70B possui 70 bilhões de parâmetros.
2. **Bytes por parâmetro (B):** cada parâmetro ocupa 4 bytes (ou 32 bits) quando carregado na GPU.
3. **Bits para carregamento (Q):** refere-se à precisão usada ao carregar o modelo na GPU, geralmente 16 bits. Esse valor afeta o tamanho da memória total, pois o modelo pode ser carregado em diferentes níveis de precisão.

4. **Fator de sobrecarga (1.2):** um fator multiplicador de 1.2 é adicionado para considerar uma sobrecarga de 20%, necessária para gerenciar ativações, cache e outros armazenamentos temporários que surgem durante a execução do modelo.

Equação 1 - Memória necessária para rodar um *Large Language Model*

$$M = \frac{(P * 4B)}{(32/Q)} * 1.2$$

Fonte: autoria própria.

A equação acima apresenta um exemplo do cálculo do requisito de memória do modelo Llama 70B com precisão de 16 bits. É necessário, aproximadamente, 168 GB de memória de GPU. Uma única GPU A100 de 80 GB não seria suficiente! (Equação 2).

Equação 2 - Exemplo do Llama 70B com precisão de 16 *bits*

$$\frac{70 * 4\text{bytes}}{32/16} * 1.2 = 168\text{GB}$$

Fonte: autoria própria.

O processo de **quantização** pode ser uma alternativa para reduzir o requisito de memória ao rodar grandes modelos (Lin et al., 2024), como o Llama 70B. Esse processo diminui a precisão dos parâmetros, reduzindo o uso de memória e permitindo uma implantação mais eficiente, especialmente em dispositivos com recursos limitados. Por outro lado, essa redução na precisão dos parâmetros tem um custo: a quantização pode afetar a exatidão e o desempenho do modelo.

Embora a quantização em 8 bits mantenha um desempenho próximo ao do modo de 16 bits, a quantização em 4 bits economiza ainda mais memória, mas pode impactar a precisão do modelo. No exemplo da Equação 3, o Llama 70B com precisão de 4 bits poderia rodar em 42 GB, usando duas GPUs L4 de 24 GB, por exemplo.

Equação 3 - Quantização do Llama 70B com precisão de 4 *bits*

$$\frac{70 * 4\text{bytes}}{32/4} * 1.2 = 42\text{GB}$$

Fonte: autoria própria.



5.3 Otimização de Latência e Acurácia

Otimizar a latência e a acurácia em LLMs permite equilibrar desempenho e confiabilidade, especialmente em aplicações que exigem respostas rápidas e precisas. A latência afeta diretamente a experiência do usuário e a aplicabilidade em tempo real, enquanto a acurácia determina a qualidade das respostas. Esses dois aspectos, embora distintos, podem ser complementares; ajustes no design de prompts, na recuperação de dados contextuais e no controle de parâmetros são estratégias que ajudam a melhorar tanto a precisão quanto a velocidade de resposta dos modelos.



5.3.1 Otimização de Latência dos *Large Language Models*

Para otimizar a latência na implementação de LLMs dentro das empresas, é necessário adotar estratégias que reduzam o tempo de resposta dos modelos, permitindo uma experiência mais fluida dos usuários. Uma das estratégias é **reduzir o tamanho do modelo** utilizado. Modelos menores podem manter boa precisão ao mesmo tempo que requerem menos recursos computacionais, o que reduz bastante o tempo de inferência. De acordo com as diretrizes da OpenAI (OpenAi, 2024), a escolha de um modelo adequado ao caso de uso específico permite que empresas alcancem um equilíbrio entre desempenho e velocidade, atendendo a requisitos de latência sem comprometer a qualidade das respostas.

Uma técnica de otimização de latência recomendada é o uso de cache para armazenar resultados de consultas frequentes. Em muitos casos de uso corporativo, certas solicitações ao modelo são recorrentes, e guardar essas respostas em cache evita a necessidade de processamento repetido, diminuindo o tempo de resposta. A OpenAI recomenda configurar um sistema de cache para armazenar respostas de consultas que sejam repetitivas (OpenAi, 2024), particularmente em ambientes de alta demanda. Este método ajuda em interfaces de perguntas e respostas e chatbots, onde perguntas semelhantes são feitas com frequência.

Outra técnica importante para otimizar a latência ao utilizar LLMs nas empresas é reduzir o número de *tokens* de entrada e saída. Como o processamento de cada *token* consome tempo e recursos computacionais das GPUs, limitar o número de *tokens* permite que o modelo retorne respostas de forma mais rápida (menor latência). Para isso, as empresas podem otimizar as consultas enviadas ao modelo, simplificando as entradas para apenas o necessário, sem informações redundantes ou desnecessárias.

A OpenAI recomenda ajustar as configurações para limitar a quantidade de *tokens* gerados na resposta também (OpenAI, 2024), pois respostas mais curtas não só reduzem a latência, mas também ajudam a economizar recursos computacionais e custos associados ao processamento do modelo. Como regra geral, cortar 50% dos *tokens* de saída pode reduzir a latência em até 50% (OpenAI, 2024). Construa um prompt para pedir ao modelo para ser mais conciso, como por exemplo: "em menos de 20 palavras" ou "seja direto". Já a redução de *tokens* de entrada tem menor impacto na latência, com ganhos de **1% a 5%** em média (OpenAI, 2024).

Cada solicitação ao modelo LLM envolve uma latência de ida e volta. Você pode **consolidar múltiplas etapas em uma única solicitação** à LLM — tal como agrupando tarefas em uma lista numerada e solicitando que o modelo responda em campos JSON —, reduzindo a latência acumulada e simplificando o processamento dos resultados. Para etapas que não dependem de uma ordem específica, é possível também realizar **chamadas paralelas** ao LLM, economizando tempo.

A percepção de latência do usuário é tão importante quanto a latência em si. Existem técnicas, como **streaming** (exibindo a resposta conforme é gerada), que tornam a espera mais interativa. Complementos visuais, como barras de carregamento e indicadores de progresso, melhoram a experiência, enquanto o *streaming* reduz o tempo de espera do usuário ao permitir que ele comece a ler antes que a resposta esteja completa.

As LLMs são muitas vezes utilizadas em casos em que um **método clássico mais rápido e barato** seria mais adequado. Com o surgimento das LLMs, houve uma grande corrida por utilizá-las em todos os cenários, mas identificar casos em que técnicas tradicionais podem ser utilizadas pode permitir-lhe reduzir significativamente a sua latência. Considere os seguintes exemplos:

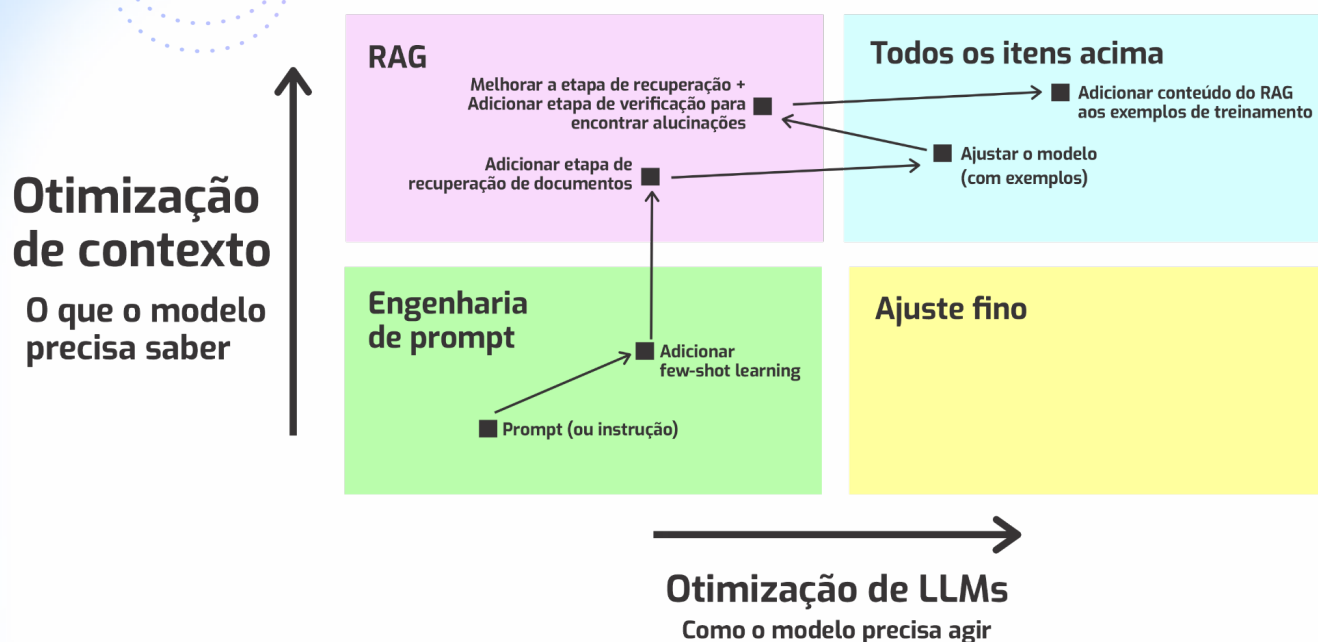
- » **Codificação tradicional:** se o **resultado** for muito restrito, pode não precisar de um LLM para o gerar. Confirmações de ações, mensagens de recusa e pedidos de entrada padrão são todos grandes candidatos a serem codificados.
- » **Pré-computação:** se a sua **entrada** for limitada (por exemplo, seleção de categoria), pode gerar várias respostas antecipadamente e certificar-se de que nunca mostra a mesma resposta a um usuário duas vezes.
- » **Aproveitamento da interface do usuário:** métricas resumidas, relatórios ou resultados de pesquisa são, por vezes, melhor transmitidos com componentes de IU clássicos e personalizados, em vez de texto gerado pelo LLM.
- » **Técnicas de otimização tradicionais:** princípios de otimização, como pesquisa binária e análise da complexidade de tempo de execução, continuam relevantes e eficazes em aplicações que utilizam LLMs e devem ser aplicados para melhorar o desempenho.

5.3.2 Otimização da Acurácia dos *Large Language Models*

A acurácia determina a relevância e a confiabilidade das respostas geradas pelos LLMs. Para atingir altos níveis de precisão, você deve aplicar métodos que vão desde o ajuste dos *prompts* até a adaptação dos parâmetros e a implementação de ciclos de verificação e treinamento. É importante ficar ciente durante a leitura dessa seção que qualquer discussão sobre a acurácia de LLMs deve começar com uma estimativa de custo-benefício: quanto uma resposta incorreta do modelo lhe custaria e quanto um acerto lhe economizaria ou geraria em receita. Essa análise de custo-benefício vai determinar o nível de precisão necessário para que o LLM seja considerado "bom o suficiente" para uso em produção na sua organização.

Na Figura 40, é apresentada uma matriz que facilita o entendimento dos passos para otimização da acurácia das LLMs. Ela divide a matriz em duas dimensões principais: otimização de contexto (o que o modelo precisa saber) e otimização do próprio LLM (como o modelo precisa agir). A otimização de contexto foca em fornecer as informações necessárias ao modelo, enquanto a otimização do LLM se concentra em ajustar seu comportamento.

Figura 40- Matriz de otimização da acurácia de *Large Language Models*



Fonte: adaptada de OpenAi (2024).

Você pode seguir diversos caminhos de otimização de acurácia, levando em conta o problema que deseja resolver. O exemplo, na Figura 40, apresenta uma ideia geral que você pode seguir na otimização da acurácia:

1. Comece com um *prompt* e avalie seu desempenho.
2. Adicione exemplos estáticos (*few-shot learning*), que devem melhorar a consistência dos resultados.
3. Inclua uma etapa de recuperação dinâmica com RAG, trazendo documentos baseados na pergunta, para garantir contexto relevante de entrada para o LLM e melhorar a acurácia da resposta.
4. Agora você pode otimizar o seu LLM para a tarefa específica que você deseja realizar (por exemplo, a classificação de documentos). Para tal, prepare um conjunto de dados com 50 ou mais exemplos e faça ajuste fino do modelo para aumentar a acurácia dele na tarefa específica.
5. Ajuste a etapa de recuperação de documentos do RAG e adicione verificação da saída do LLM para evitar alucinações e alcançar maior acurácia.
6. Re-treine o modelo que você otimizou no ajuste fino usando novos exemplos de treinamento, incluindo as entradas otimizadas do RAG.

Perceba que estas técnicas foram discutidas ao longo deste livro. A engenharia de prompt é o melhor lugar para se começar na otimização da acurácia (OpenAI, 2024). Para atingir esse objetivo, você deve sempre começar com um prompt simples e uma saída esperada em mente e, em seguida, otimizar o prompt adicionando contexto, instruções ou exemplos até que ele forneça o que você deseja. Para obter respostas mais precisas, os prompts devem ser claros, específicos e, idealmente, contextualizados. Técnicas de **engenharia de prompt** incluem a definição de perguntas diretas, o uso de exemplos e o refinamento iterativo do prompt para resolver mal-entendidos ou ambiguidades.

Além de um bom *prompt* é importante possuir um bom conjunto de avaliação dos resultados. Por exemplo, se tivermos um conjunto de mais de 20 perguntas e respostas, e tivermos examinado os detalhes das falhas e tivermos uma hipótese do porquê de elas estarem ocorrendo, então temos hipótese correta para adotar métodos de otimização mais avançados. Antes de passar para métodos de otimização mais sofisticados, também vale a pena considerar como automatizar essa avaliação para acelerar suas iterações. Você pode, por exemplo, utilizar o melhor modelo disponível (como o modelo **o1** da OpenAI) como avaliador, onde você pede ao LLM avaliar a saída da maneira mais objetiva possível.

Em aplicações onde a precisão de dados atualizados é essencial, utilizar técnicas de RAG e integrar LLMs com bases de dados ou APIs pode complementar a capacidade do modelo de fornecer respostas precisas. Em vez de confiar exclusivamente na "memória" do modelo, as informações de uma base de conhecimento externa atualizada garantem maior acurácia e reduzem o risco de respostas desatualizadas.

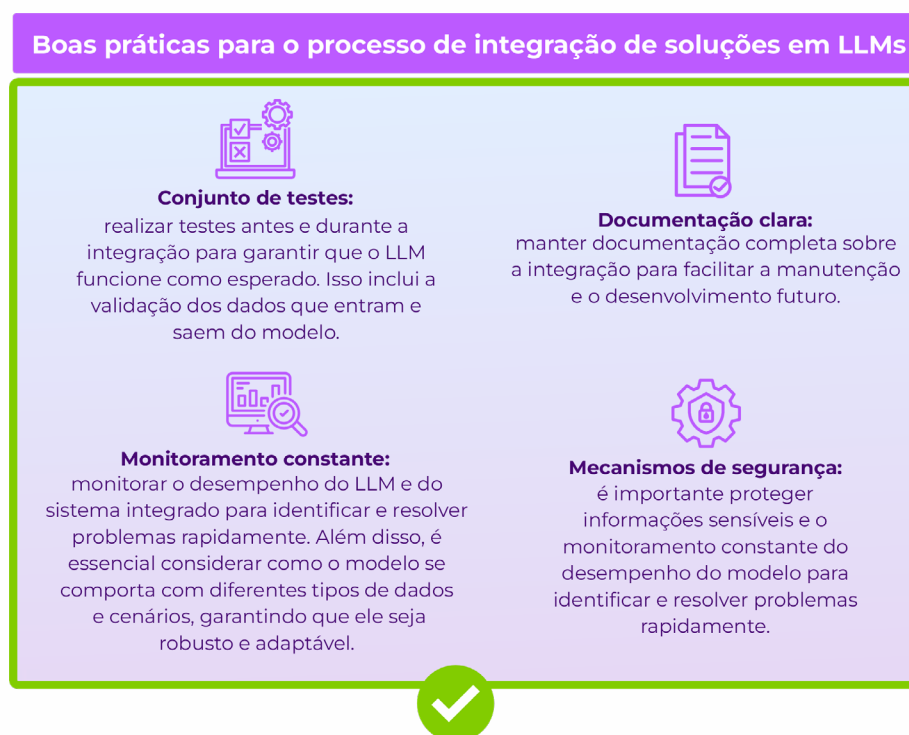
Fine-tuning permite o ajuste de LLMs para tarefas ou indústrias específicas, treinando o modelo com dados mais alinhados ao domínio de aplicação. Essa técnica melhora a acurácia, pois o modelo se torna mais adaptado a reconhecer contextos e termos técnicos específicos. Por exemplo, modelos ajustados para o setor jurídico ou médico podem interpretar melhor consultas especializadas e gerar respostas mais precisas e relevantes para esses contextos.

5.4 Integração de Soluções *Large Language Models*

Integrar um LLM em sistemas ou aplicações existentes é importante para garantir que o modelo possa ser utilizado de forma prática. Essa integração envolve várias etapas, começando pelo planejamento arquitetural, que define como o LLM será inserido na infraestrutura já existente.

Existem diferentes abordagens para a integração de LLMs, dependendo da arquitetura do sistema. Uma das maneiras mais comuns de integrar LLMs é por meio da criação de APIs. Essas APIs permitem que o LM seja acessado por outros componentes do sistema de forma padronizada. Isso facilita a comunicação entre o LLM e outras partes do sistema. Durante o processo de integração, é importante seguir boas práticas para garantir que o LLM funcione de maneira eficaz. Observe o quadro na Figura 41, a seguir.

Figura 41 - Boas práticas para garantir que o *Large Language Model* funcione de maneira eficaz



Fonte: adaptada de Gao et al. (2023).

5.5 Disponibilizar Soluções *Large Language Models* para Usuários Finais

Depois de integrar um LLM ao sistema, o próximo passo é disponibilizar o modelo em produção para que possa ser utilizado pelos usuários finais. Este processo requer cuidado e planejamento, pois envolve a configuração da infraestrutura necessária para suportar o modelo, bem como a implementação de estratégias para garantir sua segurança.

Antes de disponibilizar em produção, temos que otimizar o modelo para o ambiente de produção e configurar a infraestrutura adequadamente. Isso pode incluir a otimização do modelo para reduzir seu tamanho ou melhorar sua velocidade, bem como a configuração de servidores ou serviços em nuvem que suportem o volume esperado de tráfego.

Existem várias ferramentas e plataformas disponíveis para auxiliar na disponibilização de modelos em produção. Plataformas de computação em nuvem, como AWS (*Amazon Web Services*), Google Cloud, e Azure, oferecem serviços que podem hospedar e gerenciar modelos de inteligência artificial de maneira escalável e segura. Essas plataformas também fornecem recursos para manter a solução funcionando de forma otimizada e sempre disponível ao usuário final.

O monitoramento e manutenção do ambiente funcionando em produção requer estratégias para facilitar atualizações contínuas e seguras. Existe um grande risco de novas versões da aplicação gerarem erros no sistema e o monitoramento permite que novas versões sejam lançadas ao usuário e sejam monitoradas para validar se está atendendo às demandas dos usuários. A manutenção contínua assegura que a solução LLM permaneça atualizada com as últimas evoluções tecnológicas. Além disso, é importante planejar atualizações regulares do modelo para incorporar melhorias e corrigir possíveis vieses dos modelos.

5.6 Estudos de Caso e Exemplos Práticos

Vamos explorar alguns estudos de caso que ilustram como diferentes empresas e organizações têm integrado e escalado soluções baseadas em LLMs com sucesso. Esses exemplos práticos ajudarão a contextualizar os conceitos discutidos nas seções anteriores, oferecendo percepções sobre como essas estratégias são aplicadas na prática.

5.6.1 Chatbots para Atendimento ao Cliente

Imagine a seguinte cena: milhares de clientes acessam um *site* de comércio eletrônico simultaneamente, com dúvidas sobre produtos, entregas e pagamentos. Atender a todos individualmente seria um grande desafio. Empresas como a Amazon e a Magazine Luiza utilizam chatbots baseados em LLMs para aprimorar a experiência do cliente e otimizar o suporte oferecido.

Essas empresas treinam LLMs com milhões de interações com clientes, permitindo que os chatbots compreendam a linguagem natural e respondam a perguntas frequentes de forma rápida e precisa. Fornecendo respostas instantâneas a dúvidas comuns, os chatbots reduzem o tempo de espera dos clientes e liberam os agentes humanos para questões mais complexas.

Além disso, os chatbots possuem disponibilidade constante e garantem atendimento ininterrupto aos clientes, independentemente do horário ou volume de solicitações. Por meio da análise do histórico de compras e preferências do cliente, os chatbots oferecem recomendações e soluções personalizadas. A implementação de chatbots baseados em LLMs pode resultar em:

- » **Aumento da satisfação do cliente:** a velocidade no atendimento contribuem para uma experiência positiva do cliente.
- » **Redução de custos operacionais:** a automação de tarefas repetitivas no atendimento libera os agentes para atividades mais estratégicas, otimizando o uso de recursos humanos.
- » **Disponibilização de suporte escalável:** a capacidade de atender a um grande volume de solicitações simultâneas garante a satisfação do atendimento ao usuário mesmo em períodos de pico.

5.6.2 Personalização de Conteúdo com *Large Language Models*

Em um mundo sedento por informações, oferecer conteúdo relevante para cada usuário é a chave para o sucesso. A Netflix, por exemplo, utiliza LLMs para analisar o histórico de visualizações, avaliações e preferências de cada usuário, gerando recomendações personalizadas de filmes e séries. Essa personalização aumenta a probabilidade do usuário encontrar conteúdo de seu interesse, otimizando a experiência na plataforma.

Já o Spotify utiliza LLMs para criar *playlists* personalizadas com base no histórico de reproduções, curtidas e artistas favoritos de cada usuário. Essa funcionalidade permite que o usuário descubra novas músicas e artistas que se encaixam em seu perfil, enriquecendo a experiência musical.

A personalização de conteúdo com LLMs permite:

- » **Aumento do engajamento:** conteúdo relevante e personalizado mantém o usuário conectado à plataforma por mais tempo.
- » **Fortalecimento da fidelização:** a experiência personalizada aumenta a percepção de valor da plataforma, incentivando a fidelização do usuário.
- » **Geração de percepções:** os dados coletados sobre as preferências dos usuários podem ser utilizados para aprimorar a oferta de conteúdo e serviços.



5.6.3 Tradução Automática

A globalização exige comunicação rápida em diversos idiomas. O Google Tradutor, um dos pioneiros nesse campo, utiliza LLMs para traduzir textos, *sites* e documentos em tempo real, oferecendo suporte a diversos pares de idiomas e tornando a informação acessível a um público global.

A Microsoft integrou LLMs em sua plataforma Skype, permitindo que usuários conversem em tempo real com tradução simultânea, facilitando a comunicação entre pessoas que falam diferentes idiomas.

A tradução de conteúdo em tempo real permite que empresas e criadores alcancem um público global. A tradução automática torna o conhecimento e a informação acessíveis a um público mais amplo, independentemente da língua materna, além de estimular a colaboração internacional em diversas áreas.



SAIBA MAIS...

Para se aprofundar nos tópicos de integração e escalabilidade de LLMs, explore os seguintes recursos:

✿ Arquiteturas de Microsserviços e APIs:

✿ "*Building Microservices*" por Sam Newman: o livro oferece um guia completo sobre arquitetura de microsserviços, incluindo *design*, desenvolvimento, *deploy* e manutenção.

✿ Documentação de APIs (ex: REST, GraphQL): familiarize-se com diferentes estilos de arquitetura de APIs e como eles podem ser utilizados para integrar LLMs em sistemas existentes.

✿ Plataformas de Computação em Nuvem:

✿ Documentação da AWS, Google Cloud e Azure: explore os serviços oferecidos por essas plataformas para *deploy* e gerenciamento de modelos de IA, como balanceamento de carga, escalonamento automático e monitoramento.

✿ Kubernetes: aprenda sobre Kubernetes, uma plataforma open-source para automatizar o *deploy*, o escalonamento e o gerenciamento de aplicações em contêineres.

✿ MLOps:

✿ "*Introducing MLOps*" por Mark Treveil: este livro introduz os conceitos de MLOps e como eles podem ser aplicados para gerenciar o ciclo de vida de modelos de *machine learning*, incluindo LLMs.

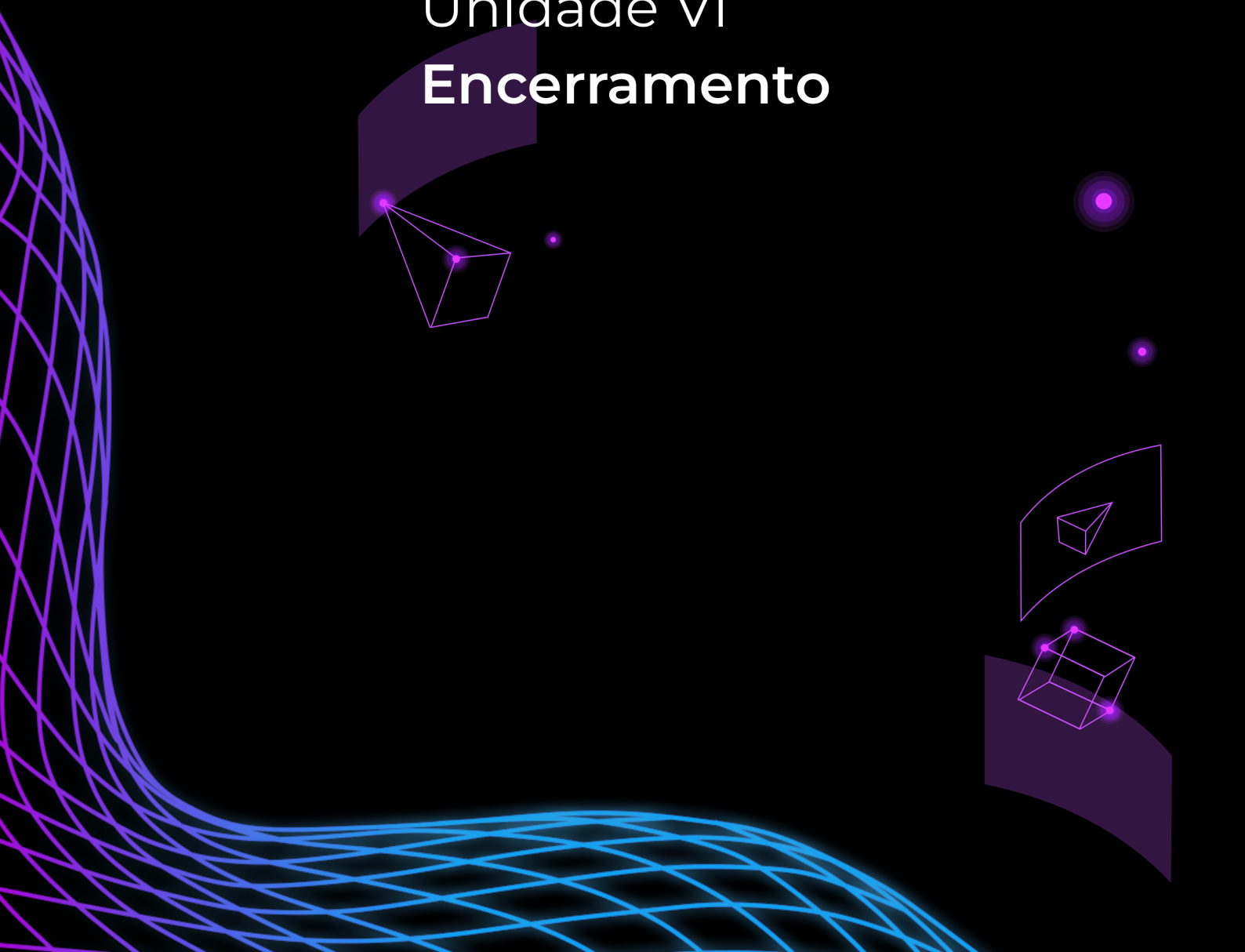
✿ Ferramentas de MLOps (ex: MLflow, Kubeflow): estude diferentes ferramentas que podem auxiliar no desenvolvimento, treinamento, *deploy* e monitoramento de modelos de IA.

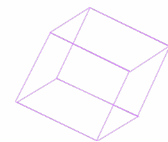


PARA RELEMBRAR...

- ✿ Nessa Unidade, exploramos os conceitos de integração e escalabilidade de soluções LLMs. Vamos lembrar os principais termos:
- ✿ Escalabilidade: capacidade de um sistema de lidar com o aumento da demanda (usuários, dados, complexidade) sem comprometer o desempenho.
- ✿ Integração: processo de incorporar um LLM em sistemas ou fluxos de trabalho existentes, assegurando que funcione de forma coesa com outros componentes. Permite que os LLMs sejam utilizados de forma em aplicações do mundo real.
- ✿ Paralelismo de modelo: divisão do modelo em partes menores que podem ser processadas simultaneamente, acelerando o processamento.
- ✿ GPUs: processadores especializados que oferecem maior poder computacional para tarefas intensivas como a execução de LLMs.
- ✿ APIs: permitem que o LLM seja acessado por outros componentes do sistema de forma padronizada, facilitando a comunicação e a integração.
- ✿ *Deploy*: processo de disponibilizar o modelo em produção para que possa ser utilizado pelos usuários finais. Envolve a configuração da infraestrutura e a implementação de estratégias para garantir a eficiência e a segurança do modelo.
- ✿ Plataformas de computação em nuvem (ex: AWS, Google Cloud, Azure): oferecem serviços para hospedar e gerenciar modelos de IA de forma escalável e segura.

Unidade VI Encerramento





Unidade VI - Encerramento

Ao longo deste livro, exploramos os fundamentos dos LLMs, uma das tecnologias mais importantes no campo do PLN.

Os LLMs demonstram capacidades que se assemelham ao raciocínio humano, em tarefas como geração de texto criativo, tradução automática, análise de sentimentos e criação de chatbots. Neste livro detalhamos como funcionam os LLMs e como são treinados utilizando a arquitetura Transformer. É importante notar que os LLMs exigem grande volume de dados e poder computacional disponível para conseguir treinar os modelos a partir do zero. Por isso, grandes empresas têm se destacado no lançamento dos LLMs mais utilizados na atualidade.

Discutimos como a utilização de técnicas de engenharia de *prompts* com a estruturação adequada de entradas podem melhorar os resultados obtidos, adaptando os modelos a uma variedade de tarefas, como geração de texto, tradução e respostas a perguntas. Falamos sobre o conceito de *in-context learning*, onde os modelos aprendem e ajustam suas respostas baseando-se em exemplos fornecidos durante a interação.

Quando precisamos de LLMs especializados em áreas como tradução, geração de código ou escrita criativa, o *fine-tuning* é uma opção. Esse ajuste fino molda as capacidades dos LLMs para atender necessidades específicas, aprimorando a precisão e a qualidade das saídas. Outro tema importante foi o RAG, que combina a geração de texto com a recuperação de informações, tornando os LLMs mais precisos e relevantes. Ao conectar os LLMs a vastas bases de dados externas, o RAG permite que eles busquem informações atualizadas em tempo real, ampliando seu conhecimento e capacidade de gerar respostas precisas e relevantes.

Por fim, abordamos os desafios que ainda precisam ser enfrentados na evolução dos LLMs, incluindo questões de escalabilidade e integração com as aplicações. Destacamos a importância de conectar os modelos a sistemas existentes e garantir seu bom funcionamento com o aumento da demanda.

6.1 Sugestões para Ação Futura e Leitura Adicional

A pesquisa em LLMs está em constante evolução, com novas descobertas e avanços sendo feitos a todo momento. O ICL, o *fine-tuning* com técnicas como LoRA, LongLoRA e QLoRA, e o aprimoramento das arquiteturas RAG são apenas alguns exemplos das áreas em constante desenvolvimento.

A jornada pelo mundo dos LLMs não termina aqui! Após absorver os conceitos deste livro, você está pronto(a) para ir além e se aprofundar nesse universo em constante expansão. Aqui estão algumas sugestões para continuar sua jornada de aprendizado:

- » **Prática com ferramentas e ambientes de teste:** envolva-se ativamente na experimentação com LLMs usando plataformas como *Google Colab* ou *Hugging Face*, que oferecem ambientes acessíveis para testar e ajustar modelos. Essas plataformas oferecem acesso a LLMs pré-treinados e recursos para você experimentar diferentes técnicas de *prompt engineering*, *fine-tuning* e RAG.
- » **Estudo de modelos específicos:** dedique tempo ao estudo de arquiteturas específicas, como GPT, Llama e Gemini. Cada uma dessas arquiteturas tem suas peculiaridades e pontos fortes, e entender essas diferenças pode ser um diferencial na escolha da melhor abordagem para diferentes projetos.
- » **Leitura de literatura acadêmica e técnica:** explore artigos científicos e publicações que discutem os avanços mais recentes no campo. Conferências como NeurIPS, ACL e publicações como o *Journal of Machine Learning Research* são fontes ricas de conhecimento avançado.
- » **Crie seus próprios projetos:** aplique seus conhecimentos em projetos pessoais que explorem as capacidades dos LLMs. Que tal construir um *chatbot*, um sistema de tradução automática ou um gerador de conteúdo criativo?
- » **Junte-se à comunidade:** explore os conteúdos de fóruns, grupos online e comunidades dedicadas a LLMs, como o fórum do *Hugging Face* e o *OpenAI Community Forum*. Compartilhe suas dúvidas, aprenda com as experiências de outros e colabore em projetos *open-source*.



6.2 Leitura Adicional

- » *"Prompt Engineering for Generative AI"*, por James Phoenix e Mike Taylor;
- » *"Natural Language Processing with Transformers"*, por Lewis Tunstall, Leandro von Werra e Thomas Wolf;
- » *"Deep Learning for Natural Language Processing"*, por Yoav Goldberg.



6.3 Reflexões Finais

Os LLMs representam uma evolução no campo do PLN. Sua capacidade de compreender e gerar linguagem abre um universo de possibilidades para diversas áreas, desde a automação de tarefas repetitivas até a criação de novas formas de arte e interação. No entanto, à medida que continuamos a explorar suas capacidades, é importante lembrar que esses modelos são, ao mesmo tempo, potentes e limitados. A responsabilidade no uso dos LLMs envolve não apenas a busca por resultados precisos, mas também a consideração dos impactos éticos e sociais que eles podem gerar.

Concluimos aqui nossa jornada introdutória pelos LLMs. O campo continua a evoluir rapidamente e incentivamos você a seguir explorando, aprendendo e contribuindo para o desenvolvimento responsável dessa área tão promissora.

Referências

AWS. Transform your business with generative. **AWS**, [online], 2024. Disponível em: <https://aws.amazon.com/ai/generative-ai/>. Acesso em: 29 nov. 2024.

BENGIO, Y.; DUCHARME, R.; VINCENT, P.. A neural probabilistic language model. **Advances in Neural Information Processing Systems**, v. 13, 2000.

BROWN, T. *et al.* Language models are few-shot learners. **Advances in Neural Information Processing Systems**, v. 33, p. 1877-1901, 2020.

CHEN, Y. *et al.* Longlora: Efficient fine-tuning of long-context large language models. **ICLR 2024**, 20203. arXiv preprint arXiv:2309.12307.

CHENG, M.; DURMUS, E.; JURAFSKY, D.. Marked personas: Using natural language prompts to measure stereotypes in language models. In: The 61st Annual Meeting Of The Association For Computational Linguistics. 2023. **arXiv preprint** arXiv:2305.18189.

COLLOBERT, R. *et al.* Natural language processing (almost) from scratch. **Journal of Machine Learning Research**, v. 12, p. 2493– 2537, 2011.

COMMON CRAWL. **Common Crawl**, 2007. Disponível em: <https://commoncrawl.org/>. Acesso em: 29 nov. 2024.

DETTMERS, T. *et al.* Qlora: Efficient finetuning of quantized LLMs. **Advances in Neural Information Processing Systems**, v. 36, 2024.

DEVLIN, J.; CHANG, M-W.; LEE, K.; TOUTANOVA, K.. BERT: **Pre-training of Deep Bidirectional Transformers for Language Understanding**. In: BURSTEIN, Jill; DORAN, C.; SOLORIO, T. (Ed.). Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, v. 1 (Long and Short Papers). Minneapolis, Minnesota: Association for Computational Linguistics, 2019. p. 4171–4186. DOI: 10.18653/v1/N19-1423.

DODGE, J. *et al.* Documenting large webtext corpora: a case study on the colossal clean crawled corpus. **arXiv preprint**, arXiv:2104.08758, 2021.

DONG, Q. *et al.* A survey on in-context learning. **arXiv preprint**, arXiv:2301.00234, 2022.

EDMONDS, J.. Matroids and the greedy algorithm. **Mathematical programming**, v. 1, p. 127-136, 1971.

FEDUS, W.; ZOPH, B.; SHAZEER, N.. Switch transformers: scaling to trillion parameter models with simple and efficient sparsity. **Journal of Machine Learning Research**, v. 23, n. 120, p. 1-39, 2022.

GARTNER. Gartner prevê que mais de 30% do aumento na demanda por APIs virá de Inteligência Artificial e de ferramentas que usam Modelos de Linguagem até 2026. **Gartner Newsroom**, 20 mar. 2024. Disponível em: <https://www.gartner.com/en/newsroom/press-releases/2024-03-20-gartner-predicts-more-than-30-percent-of-the-increase-in-demand-for-apis-will-come-from-ai-and-tools-using-llms-by-2026>. Acesso em: 24 out. 2024.

GAO, Y. *et al.* Retrieval-augmented generation for large language models: a survey. **arXiv preprint**, arXiv:2312.10997, 2023.

GEHMAN, S. *et al.* Real toxicity prompts: evaluating neural toxic degeneration in language models. **arXiv preprint**, arXiv:2009.11462, 2020.

GERO, K. I.; LIU, V.; CHILTON, L.. Sparks: inspiration for science writing using language models. **Proceedings of the 2022 ACM Designing Interactive Systems Conference**. 2022. p. 1002-1019.

GIRAY, L.. Prompt engineering with ChatGPT: a guide for academic writers. **Annals of Biomedical Engineering**, v. 51, n. 12, p. 2629-2633, 2023.

HADI, M. U. *et al.* Large language models: a comprehensive survey of its applications, challenges, limitations, and future prospects. **Authorea Preprints**, 2024.

HOFFMANN, J. *et al.* Training compute-optimal large language models. **arXiv preprint**, arXiv:2203.15556, 2022.

HOLTZMAN, A. *et al.* The curious case of neural text degeneration. In: International Conference on Learning Representations. **arXiv preprint**, arXiv:1904.09751, 2019.

HU, E. J. *et al.* Lora: low-rank adaptation of large language models. In: **Proceedings of the 2022 International Conference on Learning Representations (ICLR)**, 2022. arXiv preprint arXiv:2106.09685.

JURAFSKY, D.; MARTIN, J. H.. Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition. **Pearson/Prentice Hall**, 3. ed., 2024.

KAPLAN, J. *et al.* Scaling laws for neural language models. **arXiv preprint**, arXiv:2001.08361, 2020.

KNIGHT, W.. OpenAI's CEO says the age of Giant AI Models is already over. **WIRED**, 17 abr. 2023. Disponível em: <https://www.wired.com/story/openai-ceo-sam-altman-the-age-of-giant-ai-models-is-already-over/>. Acesso em: 24 nov. 2024.

KOMBRINK, S. *et al.* Recurrent neural network based language modeling in meeting recognition. In: **Interspeech**. 2011. p. 2877-2880.

LEWIS, M. Bart: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. **arXiv preprint**, arXiv:1910.13461, 2019.

LI, P. *et al.* Making ai less" thirsty": uncovering and addressing the secret water footprint of ai models. **arXiv preprint**, arXiv:2304.03271, 2023.

LIN, C-Y. Rouge: a package for automatic evaluation of summaries. **Text Summarization Branches Out**. 2004. p. 74-81.

LIN, J. *et al.* AWQ: Activation-aware weight quantization for on-device LLM compression and acceleration. **Proceedings of Machine Learning and Systems**, v. 6, p. 87-100, 2024.

LIU, Y. *et al.* RoBERTa: a robustly optimized BERT pretraining approach. **CoRR**, v. abs/1907.11692, 2019.

MANNING, C. D.; SCHÜTZE, H. Foundations of statistical natural language processing. **MIT Press**, 1999.

MCGUFFIE, K.; NEWHOUSE, A.. The radicalization risks of GPT-3 and advanced neural language models. **arXiv preprint**, arXiv:2009.06807, 2020.

MIKOLOV, T. *et al.* Distributed representations of words and phrases and their compositionality. **Advances in Neural Information Processing Systems**, v. 26, 2013.

MIKOLOV, T. *et al.* Recurrent neural network based language model. In: **Interspeech 2010, 11th Annual Conference of the International Speech Communication Association**, Makuhari, Chiba, Japan, September 26-30, 2010, Kobayashi, T.; Hirose, K.; Nakamura, K. (Eds.). ISCA, 2010, p. 1045-1048.

MIKOLOV, T. *et al.* Efficient estimation of word representations in vector space. **Cornell University**, v. 3781, 2013. arXiv preprint arXiv:1301.3781.

OPENAI. Latency Optimization. **OpenAI Documentation**. Disponível em: <https://platform.openai.com/docs/guides/latency-optimization>. Acesso em: 24 out. 2024.

PAPINENI, K. *et al.* Bleu: a method for automatic evaluation of machine translation. **Proceedings of the 40th annual meeting of the Association for Computational Linguistics**. 2002. p. 311-318.

PETERS, M. E. *et al.* Deep contextualized word representations. *In: Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*, M. A. Walker, H. Ji, & A. Stent, Eds. Association for Computational Linguistics, 2018, pp. 2227–2237.

PICKLE, B.. Unified memory architecture. **Techterm.com**, [online], 2023. Disponível em: https://techterms.com/definition/unified_memory_architecture. Acesso em: 29 nov. 2024.

PHOENIX, J.; TAYLOR, M.. **Prompt engineering for generative AI**. O'Reilly Media, Incorporated, 2024.

RADFORD, A. *et al.* Language models are unsupervised multitask learners. **OpenAI blog**, v. 1, n. 8, p. 9, 2019.

RAFFEL, C. *et al.* Exploring the limits of transfer learning with a unified text-to-text transformer. **Journal of Machine Learning Research**, v. 21, p. 140: 1–140: 67, 2020.

SANH, V. *et al.* Multitask prompted training enables zero-shot task generalization. *In: The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. **arXiv preprint**, arXiv:2110.08207.

SHOEYBI, M. *et al.* Megatron-lm: training multi-billion parameter language models using model parallelism. **arXiv preprint**, arXiv:1909.08053, 2019.

SINGH, S.. All about deep learning models that you should know. **Labeller**, [online], 2014. Disponível em: <https://www.labellerr.com/blog/all-about-deep-learning-models-that-you-should-know/>. Acesso em: 29 nov. 2024.

SHANAHAN, M.. Talking about large language models. **Communications of the Association for Computing Machinery**, v. 67, n. 2, p. 68-79, 2024. arXiv:2212.03551.

SHENG, E. *et al.* The Woman Worked as a Babysitter: On Biases in Language Generation. **Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)**. 2019. p. 3407-3412.

TANG, R. *et al.* Large language models can be lazy learners: analyze shortcuts in in-context learning. **arXiv preprint**, arXiv:2305.17256, 2023.

TAYLOR, R. *et al.* Galactica: a large language model for science. **arXiv preprint**, arXiv:2211.09085, 2022.

TURNEY, P. D.; PANTEL, P.. From frequency to meaning: vector space models of semantics. **Journal of Artificial Intelligence Research**, v. 37, p. 141-188, 2010.

VASWANI, A. Attention is all you need. *In: Advances in Neural Information Processing Systems 30*: Annual Conference on Neural Information Processing Systems 2017, Long Beach, CA, USA, 2017, pp. 5998–6008.

VIEJO, D. P.. The science of control: how temperature, Top_p, and Top_K shape large language models. **Medium** [online], 2013. Disponível em: <https://medium.com/@daniel.puenteviejo/the-science-of-control-how-temperature-top-p-and-top-k-shape-large-language-models-853cb0480dae>. Acesso em: 24 nov. 2023.

WANG, Y. *et al.* Generalizing from a few examples: A survey on few-shot learning. **Association for Computing Machinery Computing Surveys (CSUR)**, v. 53, n. 3, p. 1-34, 2020.

WANG, T. *et al.* What language model architecture and pretraining objective works best for zero-shot generalization?. **International Conference on Machine Learning**. PMLR, 2022. p. 22964-22984.

WEI, J. *et al.* Chain-of-thought prompting elicits reasoning in large language models. **Advances in Neural Information Processing Systems**, v. 35, p. 24824-24837, 2022.

WEI, J. *et al.* Emergent abilities of large language models. **arXiv preprint**, arXiv:2206.07682, 2022.

WHITE, J. *et al.* A prompt pattern catalog to enhance prompt engineering with chatgpt. **arXiv preprint**, arXiv:2302.11382, 2023.

ZHAO, W. X. *et al.* A survey of large language models. **arXiv preprint**, arXiv:2303.18223, 2023.



OKCIT

CENTRO DE COMPETÊNCIA EMBRAPII
EM TECNOLOGIAS IMERSIVAS



MINISTÉRIO DA
CIÊNCIA, TECNOLOGIA
E INOVAÇÃO



SOBRE O E-BOOK

Tipografia: Montserrat

Publicação: Cegraf UFG

Câmpus Samambaia, Goiânia -

Goiás, Brasil. CEP 74690-900

Fone: (62) 3521-1358

<https://cegraf.ufg.br>