

Contruindo Chatbots: Do AIML e PLN até os LLMs

Teoria e Prática com Python

Preparação e revisão
Diagramação

Giseldo Neo e Alana Neo
Giseldo Neo

©2025
versão 0.4

E-mail: giseldo@gmail.com

Todos os direitos reservados.
Nenhuma parte desta publicação
poderá ser armazenada ou reproduzida
por qualquer meio sem a autorização
por escrito dos autores.

Agradecimentos

Expresso minha gratidão ao Professor Olival de Gusmão Freitas Júnior (UFAL) pela cuidadosa revisão deste livro, cujo olhar crítico contribuiu para elevar a qualidade do trabalho.

Agradeço especialmente ao Dr. Joaquim José Cintra Maia Honório pelo auxílio no capítulo dos LLMs e *Fine-Tuning*; essas contribuições foram importantes e melhoraram muito o texto a partir do rascunho original.

Agradeço ao Professor Evandro de Barros Costa (UFAL) pela apresentação do ELIZA; ao Professor José Antônio Beltrão Moura (UFMG) pela inspiração e dedicação ao ensino; e agradeço ao Professor Otávio Monteiro Pereira (UFAL) pelo apoio relacionado à ética dos chatbots.

Agradeço ao Instituto Federal de Alagoas e a seus servidores que permitiram que esta obra fosse realizada.

Estendo meus agradecimentos a todos que, de forma direta ou indireta, colaboraram com ideias, discussões e sugestões que enriqueceram o conteúdo apresentado.

Giseldo Neo

Agradeço à minha família, que sempre esteve ao meu lado em cada etapa desta jornada. Aos meus pais, pelo amor, paciência e incentivo incondicional. Aos parentes, pelo apoio e pela compreensão das ausências nos momentos de dedicação intensa. Agradeço também aos amigos e colegas que contribuíram com ideias, sugestões e palavras de motivação nos dias desafiadores. Este livro é fruto não apenas do meu esforço, mas também da força coletiva daqueles que acreditaram e apoiaram para que esta obra se tornasse realidade.

Alana Neo

Prefácio

Vivemos um momento histórico em que a inteligência artificial (IA) deixou de ser apenas tema de ficção científica para se tornar parte do nosso cotidiano. Os chatbots, os avanços em Processamento de Linguagem Natural (PLN) e os poderosos Modelos de Linguagem Grande (LLMs) - em inglês, Large Language Models - são hoje protagonistas dessa transformação, moldando a forma como nos comunicamos, aprendemos, trabalhamos e interagimos com a tecnologia.

O que antes era limitado a sistemas rígidos, que apenas repetiam respostas pré-programadas, evoluiu para assistentes virtuais capazes de compreender contexto, interpretar nuances da linguagem e até gerar conteúdo inédito. Essa trajetória revela não apenas conquistas tecnológicas, porém também desafios éticos, sociais e econômicos.

Este livro é um convite acessível e profundo para quem deseja compreender como nascem, evoluem e funcionam os chatbots — desde os primórdios com ELIZA até os modernos modelos de linguagem como o ChatGPT.

Este livro nasce com o propósito de explicar, refletir e inspirar. Explicar como o PLN e os LLMs funcionam e por que se tornaram tão relevantes. Refletir sobre os impactos dessa revolução: o futuro do trabalho, a responsabilidade no uso da IA, a relação entre humano e máquina. Inspirar pesquisadores, profissionais, estudantes e curiosos a explorar esse campo em rápida expansão, contribuindo para um uso responsável e criativo dessas ferramentas tecnológicas. Assim, o leitor encontrará não apenas conceitos técnicos, mas também análises críticas, casos de uso e visões sobre os caminhos possíveis que a era da IA conversacional nos reserva.

Este livro é um convite para explorar esse universo fascinante. Você encontrará explicações sobre como essas tecnologias funcionam, exemplos e reflexões sobre seus impactos na sociedade. Mais do que compreender as máquinas que conversam, trata-se de entender o futuro da linguagem e da interação entre humanos e sistemas inteligentes.

Seja você estudante, pesquisador ou profissional do mercado, este livro vai inspirar novas ideias e mostrar como a inteligência artificial conversacional já está transformando o presente — e continuará a transformar o futuro.

Boa leitura!

Informações

O código-fonte do livro está disponível no endereço: <https://github.com/giseldo/Livro-A-jornada-dos-Chatbots---Verde---C-digos>.

Uma versão atualizada do livro pode ser encontrada no site <https://giseldo.github.io/>.

Resumo

Se você já se perguntou como funcionam os chatbots que conversam como humanos, este livro é a sua porta de entrada para um dos campos mais fascinantes da inteligência artificial. Este livro é mais do que um guia técnico — é uma jornada envolvente pela evolução, construção e futuro dos agentes conversacionais, unindo clareza didática com profundidade técnica. A obra começa explorando o conceito de chatbot, diferenciando suas principais categorias — conversacionais e orientados a tarefas — e contextualizando o leitor com um panorama histórico que vai do lendário ELIZA, criado nos anos 1960, até os impressionantes modelos atuais como ChatGPT. Essa viagem no tempo é pontuada por explicações acessíveis e exemplos práticos em Python que mostram, passo a passo, como esses sistemas funcionam por trás das cortinas. O livro se destaca por traduzir conceitos complexos em linguagem clara. Você vai entender o que é o pattern matching usado por chatbots clássicos, como funciona o AIML, e por que expressões regulares ainda são ferramentas utilizadas no desenvolvimento de bots. Em capítulos dedicados ao Processamento de Linguagem Natural (PLN), técnicas como tokenização, lematização e análise sintática ganham vida por meio de exemplos aplicáveis, preparando o leitor para dar seus próprios passos no mundo da IA conversacional. Mas ele não se limita ao passado. Ele mergulha fundo nas tecnologias que revolucionaram o campo, como *Transformers*, Word2Vec e os Modelos de Linguagem Grande (LLMs), incluindo BERT, GPT e LLaMA. O leitor descobre como essas arquiteturas funcionam, como treiná-las e como aplicá-las usando frameworks modernos como Hugging Face e LangChain. Além de guiar a construção técnica de um chatbot — do código ao deploy — o livro também traz reflexões sobre ética, privacidade e o impacto social desses agentes. Tópicos como personalização, explicabilidade e segurança ampliam o olhar do leitor para além da implementação. Combinando teoria, prática e visão de futuro, ele é um convite para programadores, educadores, empreendedores e curiosos que desejam dominar a arte de construir diálogos entre humanos e máquinas. É uma obra que ensina, inspira e instiga. Se você quer entender como os chatbots realmente pensam — ou pelo menos como fingem tão bem — este livro é para você.

Sumário

1	Chatbots	13
1.1	Introdução	14
1.2	Agentes	16
1.3	Fluxo Conversacional	18
1.4	Histórico	21
1.5	Abordagens	22
1.6	Chatbot não pensa	24
2	Eliza: O primeiro chatbot	27
2.1	Introdução	28
2.2	Processamento	29
2.3	Regras de Transformação	31
2.4	Implementação e Variações	32
2.5	Mecanismo de Pesos	34
2.6	Geração de Texto	35
2.7	ELIZA em Python	36
2.8	Criando um ELIZA Simples com Interface e Gradio	40
2.9	Artificial Intelligence Markup Language	45
2.9.1	Tags do AIML	50
2.9.2	Exemplo em Python	54
3	Processamento de Linguagem Natural (PLN)	57
3.1	Inteligência Artificial	58
3.2	Aprendizado de Máquina	60
3.3	Processamento de Linguagem Natural	62
3.4	Instalação do Python	63
3.5	Principais Técnicas de PLN	66
3.5.1	Tokenização	66
3.5.2	Lematização	68
3.5.3	Stemização	69
3.5.4	Stopwords	70
3.6	Outras técnicas de PLN	71
3.7	Expressões Regulares	72

3.8	Entendimento de Linguagem Natural	77
3.8.1	Intents	79
3.8.2	Utterances	80
3.8.3	Entities	80
3.8.4	Desafios	80
3.9	Vetorização e Representação de Texto	81
3.9.1	One-Hot Encoding	82
3.9.2	Bag of Words (BoW)	83
3.9.3	TF-IDF (Term Frequency-Inverse Document Frequency)	84
3.9.4	Comparação de Técnicas de Vetorização	84
3.9.5	Implementação em Projetos Reais	85
3.9.6	Embeddings de Palavras	86
3.10	Resumo	91
4	Modelos de Linguagem Grande (LLM)	93
4.1	Introdução	94
4.2	Arquitetura Geral do <i>Transformer</i>	97
4.2.1	BERT (<i>Bidirectional Encoder Representations from Transformers</i>)	98
4.2.2	GPT (<i>Generative Pre-trained Transformer</i>)	101
4.2.3	distilbert-base-uncased	104
4.3	<i>Fine-Tuning</i> de Modelos Pré-Treinados	105
4.4	Few Shot e Zero Shot Learning	109
4.5	Retrieval-Augmented Generation	110
4.6	LLaMA: Modelos Pequenos e Eficientes	115
4.6.1	Arquitetura do LLaMA	115
4.6.2	Exemplo de uso do LLaMA	116
4.6.3	Aplicações de LLaMA	117
4.6.4	Comparação com Outros Modelos	118
4.7	LLM na prática	118
4.7.1	Hugging Face Pipeline	118
4.7.2	Usando um LLM Local com Ollama	119
4.7.3	Tokenizador no LLM	127
4.7.4	LangChain	127
4.7.5	Mangaba.AI	130
4.7.6	Fluxos em LLM (Engenharia de Prompts)	131
4.8	Integração de Técnicas	140
4.9	API e Playground	144
4.10	Resumo	148
5	Conclusão	151
5.1	Considerações Éticas	152
5.2	Privacidade e Proteção de Dados	154
5.3	Viés Algorítmico	155

5.4	Segurança de Chatbots	156
5.5	Manutenção e Atualização	158
5.6	Perspectivas Futuras	160
5.7	Oportunidades de Inovação	161
5.8	Considerações Finais	167

Capítulo 1

Chatbots

Podemos ver apenas uma curta distância à frente, mas pode-se ver muito que precisa ser feito.

Alan Turing

Objetivo
Apresentar os conceitos fundamentais de chatbots, sua classificação, histórico e principais abordagens, preparando o leitor para compreender a evolução dessas tecnologias.

1.1 Introdução

Para uma parcela da população com maior afinidade com a inteligência artificial, o chatbot tornou-se uma ferramenta essencial para executar tarefas, redigir e-mails e traduzir textos. O que um dia causou espanto e admiração — como ocorreu com a descoberta do fogo, o computador pessoal, a Internet, as ferramentas de busca e as redes sociais — hoje se converteu em dependência. Nesse contexto, um chatbot é um programa de computador que simula uma conversa humana, via texto ou áudio, oferecendo respostas diretas a perguntas e apoiando diversas atividades, desde conversas gerais até ações específicas, como abrir uma conta bancária ou agendar um voo para o show do seu cantor preferido.

Para entender como chegamos a esse ponto, vale recordar a trajetória do próprio conceito. Embora o programa ELIZA, criado por Weizenbaum [1966], seja frequentemente citado como um dos primeiros exemplos de software conversacional, o termo “chatbot” ainda não era empregado à época. Sua origem remete a “chatterbot” — sinônimo de “chatbot” —, popularizado por Michael Mauldin em 1994 ao descrever seu programa JULIA [Mauldin, 1994]. Anos depois, outras publicações acadêmicas, como os anais da *Virtual Worlds and Simulation Conference* de 1998 [Jacobstein et al., 1998], contribuíram para consolidar os termos entre os pesquisadores.

O chatbot ELIZA representou um experimento marcante na interação entre humano e computador [Weizenbaum, 1966]. Seu roteiro (ou script) mais famoso, o DOCTOR, imitava rudimentarmente um psicoterapeuta, utilizando correspondência de padrões simples. Por exemplo, quando um usuário inseria a frase “*Estou triste*” no ELIZA, o programa respondia “*Por que você está triste hoje?*”, reformulando a entrada do usuário como uma pergunta. O funcionamento básico do sistema baseava-se em um conjunto restrito de regras e substituições, o que lhe permitia apenas uma compreensão superficial e limitada da linguagem humana.

O roteiro DOCTOR do ELIZA adequou-se bem a um tipo de diálogo mais simples, pois suas respostas dependiam de pouco conhecimento sobre o ambiente externo. As regras no roteiro permitiam que o programa respondesse ao usuário com outras perguntas ou simplesmente refletisse a afirmação original. Uma

descrição detalhada do funcionamento do ELIZA, com exemplos em Python, será apresentada no Capítulo 2.

Outro chatbot famoso é o ChatGPT da OpenAI. Ele é um modelo de linguagem capaz de gerar texto muito semelhante ao criado por humanos. Ele utiliza redes neurais, com aprendizagem profunda, para gerar sentenças e parágrafos com base nas entradas e informações fornecidas. Entre suas capacidades, ele pode traduzir e resumir textos, responder a perguntas e explicar conceitos. Contudo, o ChatGPT não possui consciência nem a capacidade de compreender contexto ou emoções.

O chatGPT é um exemplo de Modelo de Linguagem Grande (em inglês Large Language Model - LLM), baseado na arquitetura *Transformers*, introduzida em 2017 [Vaswani et al., 2017]. Modelos deste tipo são treinados com terabytes de texto, utilizando mecanismos de autoatenção que avaliam a relevância de cada palavra em uma frase. Ao contrário das regras manuais do ELIZA, os LLMs extraem padrões linguísticos a partir da vasta quantidade de dados com que a rede neural foi treinada.

Esses dois chatbots, ELIZA e ChatGPT, são bons representantes do tipo de chatbot conversacional. Apesar de terem surgido com décadas de diferença — ELIZA em 1966 e ChatGPT em 2022 — e de diferirem bastante na forma como geram suas respostas, ambos compartilham semelhanças em seu objetivo: conversar sobre determinado assunto ou responder perguntas, mantendo o usuário em um diálogo fluido quando necessário. Chatbots com essas características podem ser agrupados, de acordo com o objetivo, como chatbots conversacionais e são utilizados para interagir sobre assuntos gerais.

Outro tipo de chatbot classificado em relação ao objetivo é o chatbot orientado a tarefas. Os chatbots orientados a tarefas executam ações específicas, como abrir uma conta bancária ou pedir uma pizza. Geralmente, as empresas disponibilizam chatbots orientados a tarefas para seus usuários, com regras de negócio embutidas na conversação e com fluxos bem definidos. Normalmente, não se espera pedir uma pizza e, no mesmo chatbot, discutir os estudos sobre Ética do filósofo Immanuel Kant (embora talvez haja quem queira).

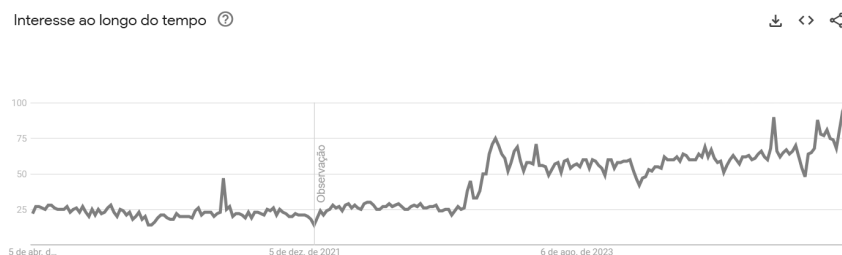
Essas duas classificações, “conversacional” e “orientado a tarefas”, ainda não são suficientes para uma completa classificação. Existem outras classificações que serão discutidas em seções posteriores. Além disso, uma abordagem

híbrida, unindo funções de chatbots do tipo “conversacional” e “orientado a tarefas” vem sendo utilizada para atender às necessidades dos usuários.

A popularidade dos chatbots tem crescido significativamente em diversos domínios de aplicação [Marcondes et al., 2020, Klopfenstein et al., 2017, Sharma et al., 2020]. Essa tendência é corroborada pelo aumento do interesse de busca pelo termo “chatbots”, conforme análise de dados do Google Trends no período entre 2020 e 2025 (Figura 1.1). Nesta figura, os valores representam o interesse relativo de busca ao longo do tempo, onde 100 indica o pico de popularidade no período analisado e 0 (ou a ausência de dados) indica interesse mínimo ou dados insuficientes.

Figura 1.1: Evolução do interesse de busca pelo termo “chatbot” (Google Trends, 2020-2025).

Fonte: Google Trends acesso em 05/04/2025



1.2 Agentes

As definições de chatbot e agentes são usadas indiscriminadamente, o que pode causar confusão. Vamos a uma definição mais precisa. Um chatbot é um programa computacional projetado para interagir com usuários por meio de linguagem natural. Por outro lado, o conceito de agente possui uma definição mais ampla. Um agente trata-se de uma entidade computacional que percebe seu ambiente por meio de sensores e atua sobre esse ambiente por meio de atuadores.

Nesse contexto, um chatbot (Figura 1.3) pode ser considerado uma instanciação específica de um agente - veja na Figura 1.2 a arquitetura conceitual de alto nível para um agente - cujo propósito primário é a interação conversacional em linguagem natural.

Figura 1.2: Arquitetura conceitual de um agente.
Fonte: Adaptado de [Russel and Norving, 2013]

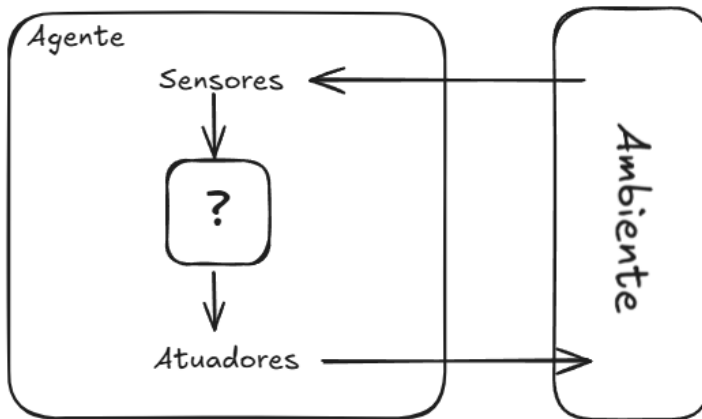
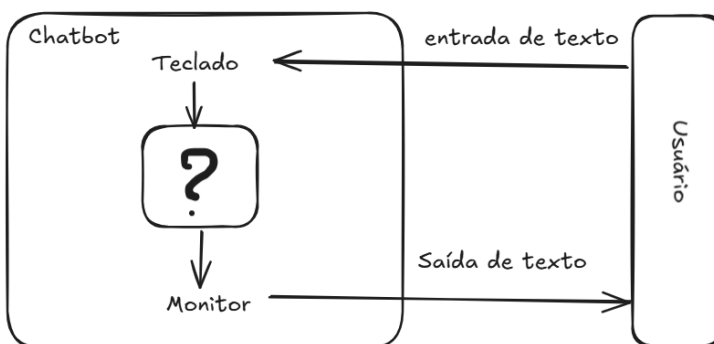


Figura 1.3: Representação esquemática de um chatbot.



Com o advento de modelos de linguagem, como os baseados na arquitetura *Generative Pretrained Transformer* (GPT), a exemplo do ChatGPT, observou-se uma recontextualização do termo “agente” no domínio dos sistemas conversacionais. Nessa abordagem mais recente, um sistema focado predominantemente na geração de texto conversacional tende a ser denominado “chatbot”. Em contraste, o termo “agente” é frequentemente reservado para sistemas que, além da capacidade conversacional, integram e utilizam ferramentas externas (por exemplo, acesso à Internet, execução de código e interação com APIs) para realizar tarefas complexas e interagir proativamente com o ambiente digital. Um sistema capaz de realizar uma compra online, processar um pagamento e confirmar um endereço de entrega por meio do navegador do usuário seria, portanto, classificado como um agente, diferentemente de chatbots mais simples como ELIZA, ou mesmo versões mais simples do chatGPT (GPT-2), cujo foco era estritamente o diálogo.

1.3 Fluxo Conversacional

Um chatbot responde a uma entrada do usuário. Porém, essa interação textual mediada por chatbots não se constitui em uma mera justaposição aleatória de turnos de conversação ou pares isolados de estímulo-resposta. Pelo contrário, espera-se que a conversação exiba coerência e mantenha relações lógicas e semânticas entre os turnos consecutivos. O estudo da estrutura e organização da conversa humana é abordado por disciplinas como a Análise da Conversação.

No contexto da análise da conversação em língua portuguesa, os trabalhos de Marcuschi [Marcuschi, 1986] são relevantes ao investigar a organização dessa conversação. Marcuschi analisou a estrutura conversacional em termos de unidades coesas, como o “tópico conversacional”, que agrupa turnos relacionados a um mesmo assunto ou propósito interacional.

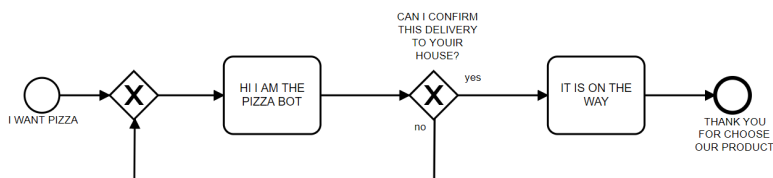
Conceitos oriundos da Análise da Conversação, como a gestão de tópicos, têm sido aplicados no desenvolvimento de chatbots para aprimorar sua capacidade de manter diálogos coerentes e contextualmente relevantes com usuários humanos [Neves and Barros, 2005].

Na prática de desenvolvimento de sistemas conversacionais, a estrutura lógica e sequencial da interação é frequentemente modelada e referida como

“fluxo de conversação” ou “fluxo de diálogo”. Contudo, é importante ressaltar que a implementação explícita de modelos sofisticados de gerenciamento de diálogo, inspirados na Análise da Conversação, não é uma característica universal de todos os chatbots, variando conforme a complexidade e o propósito do sistema.

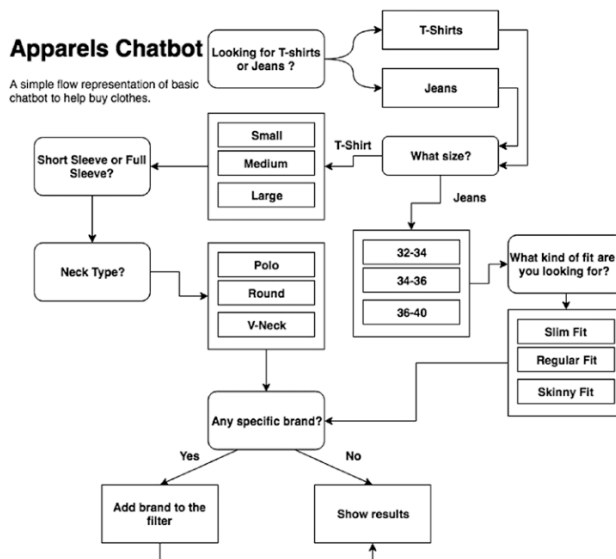
Um exemplo esquemático de um fluxo conversacional é apresentado na Figura 1.4. Nesta figura, o fluxo de conversação inicia quando o usuário entra com o texto: I WANT PIZZA, o chatbot responde com uma pergunta: HI I AM THE PIZZA BOT. CAN I CONFIRM THIS DELIVERY TO YOUR HOUSE? O usuário então pode responder: YES, e o chatbot finaliza a conversa com: IT'S ON THE WAY. THANK YOU FOR CHOOSE OUR PRODUCT. Caso o usuário responda: NO, o chatbot responde com a pergunta original: HI I AM THE PIZZA BOT. CAN I CONFIRM THIS DELIVERY TO YOUR HOUSE? O fluxo de conversação continua até que o usuário responda com um “YES” para a pergunta inicial. Essa estrutura de perguntas e respostas é comum em chatbots orientados a tarefas, onde o objetivo é guiar o usuário por um processo específico, tal como fazer um pedido de pizza.

Figura 1.4: Exemplo esquemático de um fluxo conversacional em um chatbot.



Um outro tipo de fluxo para um chatbot que vende roupas online está representado na Figura 1.5.

Figura 1.5: Representação de uma árvore de decisão para vender roupas online.
Retirado de [Raj, 2019].

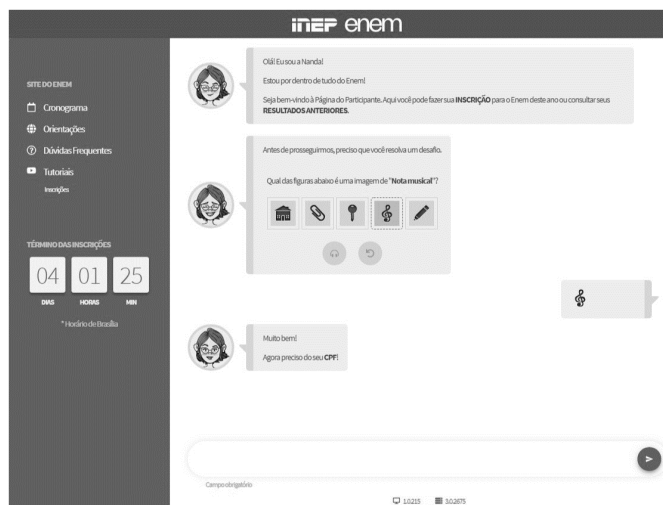


1.4 Histórico

Um marco significativo na evolução dos chatbots depois do ELIZA foi o ALICE, que introduziu a Artificial Intelligence Markup Language (AIML), uma linguagem de marcação baseada em XML [Wallace, 2000]. A AIML estabeleceu um paradigma para a construção de agentes conversacionais ao empregar algoritmos de correspondência de padrões. Essa abordagem utiliza modelos pré-definidos para mapear as entradas do usuário a respostas correspondentes, permitindo a definição modular de blocos de conhecimento [Wallace, 2000].

No contexto brasileiro, um dos primeiros chatbots documentados capaz de interagir em português, inspirado no modelo ELIZA, foi o Cybele [PRIMO and COELHO, 2001]. Posteriormente, foi desenvolvido o Elecktra, também em língua portuguesa, com aplicação voltada para a educação a distância [Leonhardt et al., 2003]. Em 2019, o processo de inscrição para o Exame Nacional do Ensino Médio (ENEM) foi disponibilizado por meio de uma interface conversacional baseada em chatbot (Figura 1.6).

Figura 1.6: Interface de chatbot para inscrição no ENEM 2019.



O desenvolvimento de chatbots tem atraído investimentos de grandes corporações. Notavelmente, a IBM desenvolveu um sistema de resposta a perguntas em domínio aberto utilizando sua plataforma Watson [Ferrucci, 2012]. Esse tipo de tarefa representa um desafio computacional e de inteligência artificial (IA)

considerável. Em 2011, o sistema baseado em Watson demonstrou sua capacidade ao competir e vencer competidores humanos no programa de perguntas e respostas JEOPARDY! [Ferrucci, 2012].

Diversos outros chatbots foram desenvolvidos para atender a demandas específicas em variados domínios. Exemplos incluem: BUTI, um companheiro virtual com computação afetiva para auxiliar na manutenção da saúde cardiovascular [Junior, 2008]; EduBot, um agente conversacional projetado para a criação e desenvolvimento de ontologias com lógica de descrição [Lima, 2017]; PMKLE, um ambiente inteligente de aprendizado focado na educação em gerenciamento de projetos [Torreao, 2005]; RENAN, um sistema de diálogo inteligente fundamentado em lógica de descrição [de Azevedo, 2015]; e MORFEu, voltado para a mediação de atividades cooperativas em ambientes inteligentes na Web [Bada, 2012].

Entre os chatbots baseados em LLMs de destaque atualmente estão o Qwen, desenvolvido pela Alibaba, que se destaca por sua eficiência e suporte multilíngue; o DeepSeek, de código aberto voltado para pesquisa e aplicações empresariais com foco em precisão e escalabilidade; o Maritaca, modelo brasileiro otimizado para o português; o Gemini, da Google, que integra capacidades multimodais e forte desempenho em tarefas diversas; o Mixtral, da Mistral AI, que utiliza arquitetura de mistura de especialistas para maior eficiência; o Llama, da Meta, reconhecido por ser código aberto e ampla adoção na comunidade; o Claude, da Anthropic, projetado com ênfase em segurança e alinhamento ético, que vem ganhando adeptos para tarefas e codificação; e o Nemotron, da NVIDIA, que oferece modelos de linguagem otimizados para execução em GPUs e aplicações empresariais de alto desempenho.

1.5 Abordagens

Desde o pioneirismo do ELIZA, múltiplas abordagens e técnicas foram exploradas para o desenvolvimento de chatbots. Entre as mais relevantes, destacam-se: AIML com correspondência de padrões (pattern matching), análise sintática (Parsing), modelos de cadeia de Markov (Markov Chain Models), uso de ontologias, redes neurais recorrentes (RNNs), redes de memória de longo prazo (LSTMs), modelos neurais sequência-a-sequência (Sequence-to-Sequence),

aprendizado adversarial para geração de diálogo, além de abordagens baseadas em recuperação (Retrieval-Based) e generativas (Generative-Based) [Borah et al., 2018, Ramesh et al., 2017, Shaikh et al., 2016, Abdul-Kader and Woods, 2015, Li et al., 2018], entre outras.

A seguir, uma lista resumida das tecnologias e marcos da criação dos chatbots:

- ELIZA: o primeiro chatbot, que utilizava correspondência de padrões simples para simular um psicoterapeuta. Foi um marco na história dos chatbots e influenciou o desenvolvimento de sistemas conversacionais subsequentes [Weizenbaum, 1966].
- AIML: Artificial Intelligence Markup Language, uma linguagem de marcação baseada em XML que deu origem ao ALICE [Wallace, 2000]. Essa linguagem de marcação permite a definição de regras de correspondência de padrões (pattern matching) para mapear entradas do usuário a respostas predefinidas. Ele é amplamente utilizado na construção de chatbots, permitindo a criação de diálogos complexos e interativos.
- TransformersBERT: arquitetura de rede neural baseada em atenção, que revolucionou o processamento de linguagem natural (NLP) [Vaswani et al., 2017]. Modelos como BERT e GPT são exemplos de arquiteturas baseadas em *Transformers* que têm sido amplamente utilizadas em chatbots modernos.
- GPT: modelos de linguagem generativa, como o GPT-2, que utilizam redes neurais profundas para gerar texto coerente e relevante em resposta a entradas do usuário. Esses modelos são treinados em grandes quantidades de dados e podem ser adaptados para tarefas específicas, como atendimento ao cliente ou suporte técnico.

Além disso, diversos frameworks têm sido desenvolvidos para facilitar a criação desses agentes complexos, como CrewAI [CrewAI, 2025] e Mangaba.AI [Mangaba AI, 2025] bibliotecas associadas a plataformas como Hugging Face (e.g., *Transformers Agents*), que fornecem abstrações e ferramentas em Python para orquestrar múltiplos componentes e o uso de ferramentas externas.

1.6 Chatbot não pensa

Apesar do progresso recente de chatbots, o mecanismo fundamental da inteligência em nível humano, frequentemente refletido na comunicação, ainda não está totalmente esclarecido [Shum et al., 2018]. Para avançar na solução desses desafios, serão necessários progressos em diversas áreas da IA cognitiva, tais como: modelagem empática de conversas, modelagem de conhecimento e memória, inteligência de máquina interpretável e controlável, e calibração de recompensas emocionais [Shum et al., 2018].

Uma das dificuldades na construção de chatbots do tipo orientado a tarefas - a exemplo do Artificial Intelligence Markup Language (AIML) usado no ALICE - reside em gerenciar a complexidade das estruturas condicionais (“se-então”) que definem o fluxo do diálogo [Raj, 2019]. Quanto maior o número de decisões a serem tomadas, mais complexas tendem a ser essas estruturas condicionais. Contudo, elas são essenciais para codificar fluxos de conversação complexos. Se a tarefa que o chatbot visa simular é inerentemente complexa e envolve múltiplas condições, o código precisará refletir essa complexidade. Para facilitar a visualização desses fluxos, uma solução eficaz é a utilização de fluxogramas. Embora simples de criar e entender, os fluxogramas constituem uma ferramenta visual de representação para este problema. Uma explicação detalhada do AIML será apresentada no Capítulo 2.

Os chatbots baseados em AIML apresentam desvantagens específicas. Por exemplo, o conhecimento é representado como instâncias de arquivos AIML. Se esse conhecimento for criado com base em dados coletados da Internet, ele não será atualizado automaticamente, exigindo atualizações periódicas manuais [Madhumitha et al., 2015]. No entanto, já existem abordagens para mitigar essa limitação, permitindo carregar conteúdo AIML a partir de fontes como arquivos XML [Macedo and Fusco, 2014], um corpus textual [De Gasperis et al., 2013] ou dados do Twitter [Yamaguchi et al., 2018]. Além de abordagens no-code que geram o AIML a partir de fluxogramas [Neo et al., 2023b].

Outra desvantagem do AIML, a exemplo do Eliza, reside na relativa complexidade de seus padrões de correspondência (patterns). Além disso, a manutenção do sistema pode ser árdua, pois, embora a inserção de conteúdo (categorias) seja conceitualmente simples, grandes volumes de informação frequentemente precisam ser adicionados manualmente [Madhumitha et al., 2015].

Especificamente no caso do AIML, a construção e a visualização de fluxos de diálogo complexos enfrentam dificuldades adicionais. Devido ao seu formato baseado em texto, muitas vezes é difícil perceber claramente como as diferentes categorias (unidades de conhecimento e resposta) se interligam para formar a estrutura da conversação.

O interesse pelos chatbots continua crescendo. No entanto, eles podem ser complicados para se construir e os usuários nem sempre têm experiência suficiente para configurá-los. Alguns usuários não têm necessariamente habilidades de programação ou de TI avançadas. Para que eles possam criar e personalizar os chatbots, é importante que a autoria seja fácil de usar e intuitiva. Ela não deve exigir conhecimento de linguagens de computador que sejam difíceis de entender para o público em geral. Para resolver esses problemas, várias ferramentas de autoria já foram propostas e podem ser utilizadas por usuários sem o uso de código, ferramentas no-code [da Silva Neo et al., 2023].

Capítulo 2

Eliza: O primeiro chatbot

Somos autómatos em três
quartas partes das nossas ações.

Wilhelm Leibniz

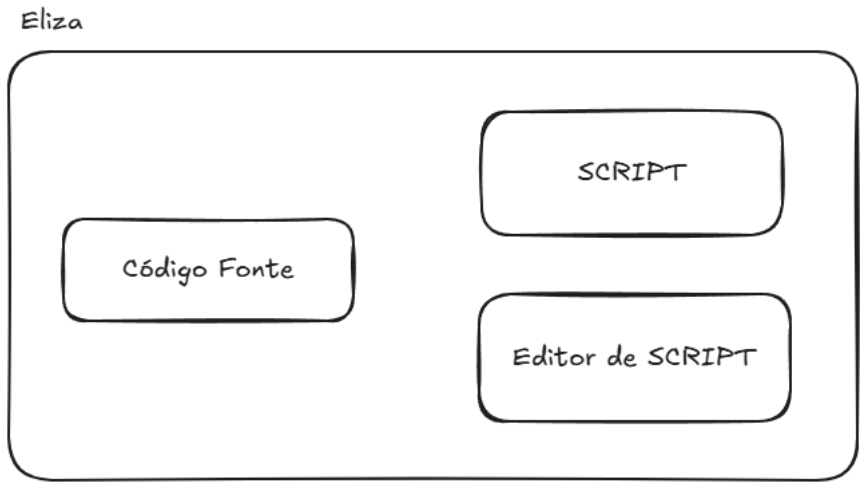
Objetivo
Explorar o funcionamento do ELIZA, suas regras de transformação, implementação e legado histórico, evidenciando as bases dos sistemas conversacionais. Além de explicar a linguagem AIML, mostrando como ela possibilitou a construção de chatbots mais robustos e modulares.

2.1 Introdução

O chatbot ELIZA foi um marco histórico e um dos primeiros programas de processamento de linguagem natural e foi apresentado em 1966 por Joseph Weizenbaum no MIT [Weizenbaum, 1966].

O conjunto de padrões e respostas predefinidas constitui o que Weizenbaum chamou de “roteiro” (ou script) de conversa. O mecanismo do ELIZA separa o motor de processamento dos dados, sendo o algoritmo de busca de palavras-chave e aplicação de regras, do script em si. Isso significa que ELIZA podia, em teoria, simular diferentes personalidades ou tópicos apenas carregando um script diferente, sem alterar o código do programa (veja na Figura 2.1). Também foi codificado um editor de texto para as alterações do próprio script.

Figura 2.1: ELIZA: separação entre o código fonte, o script e o editor de texto.



Um destes roteiros, o script DOCTOR, deixou o ELIZA famoso. Com este script carregado, o ELIZA simulava precariamente um psicoterapeuta (do estilo Rogeriano), refletindo as palavras do usuário de volta em forma de pergunta.

Na lista abaixo é apresentado um exemplo de diálogo (retirado do artigo original) entre um usuário humano e ELIZA, mostrando as respostas em letras maiúsculas. Nesse exemplo, o usuário faz afirmações pessoais (“*Men are all alike.*”, “*Well, my boyfriend made me come here.*”) e ELIZA responde com perguntas ou reformulações (“*IN WHAT WAY*”, “*YOUR BOYFRIEND MADE YOU COME*

HERE?") reutilizando termos do próprio usuário. Esse estilo reflexivo encoraja o usuário a continuar falando sem fornecer respostas diretas.

Código-fonte 2.1: Exemplo de diálogo do ELIZA.

```
1 Men are all alike.
2 IN WHAT WAY
3 They're always bugging us about something or other.
4 CAN YOU THINK OF A SPECIFIC EXAMPLE
5 Well, my boyfriend made me come here.
6 YOUR BOYFRIEND MADE YOU COME HERE
7 He says I'm depressed much of the time.
8 I AM SORRY TO HEAR YOU ARE DEPRESSED
9 It's true. I am unhappy.
10 DO YOU THINK COMING HERE WILL HELP YOU NOT TO BE UNHAPPY
11 I need some help, that much seems certain.
12 WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP
13 Perhaps I could learn to get along with my mother.
14 TELL ME MORE ABOUT YOUR FAMILY
15 My mother takes care of me.
16 WHO ELSE IN YOUR FAMILY TAKES CARE OF YOU
17 My father.
18 YOUR FATHER
19 You are like my father in some ways.
20 WHAT RESEMBLANCE DO YOU SEE
21 You are not very aggressive but I think you don't want me to
22 notice that.
23 WHAT MAKES YOU THINK I AM NOT VERY AGGRESSIVE
24 You don't argue with me.
25 WHY DO YOU THINK I DON'T ARGUE WITH YOU
26 You are afraid of me.
27 DOES IT PLEASE YOU TO BELIEVE I AM AFRAID OF YOU
28 My father is afraid of everybody.
29 WHAT ELSE COMES TO MIND WHEN YOU THINK OF YOUR FATHER
30 Bullies.
31 DOES THAT HAVE ANYTHING TO DO WITH THE FACT THAT YOUR BOYFRIEND MADE YOU
    COME HERE
```

2.2 Processamento

O funcionamento do ELIZA baseia-se em correspondência de padrões por palavras-chave [Mittmann, 2021]. Técnica também chamada “casamento de

padrão”, em inglês *pattern matching*, seu paradigma também se encaixa na técnica baseada em regras (ou *rule-based*). A cada turno do diálogo, o texto de entrada do usuário é analisado em busca de palavras-chave predefinidas. O ELIZA faz uma varredura da sentença da esquerda para a direita, identificando se alguma palavra correspondia a uma palavra-chave do script. Caso encontre uma palavra-chave no script, ELIZA empilha essa palavra até o final da frase. Depois, ele seleciona a palavra-chave mais “importante” até encontrar uma pontuação. Para isso, ele mantém uma lista de pesos associados a cada palavra-chave.

Por exemplo, o script DOCTOR definia palavras-chave como “*ALIKE*” ou “*SAME*” com alta prioridade; assim, na frase “*Men are all alike.*” o programa detectava a palavra “*ALIKE*” e disparava uma resposta associada a ela (no caso: “*In what way?*”). Se múltiplas palavras-chave aparecessem, ELIZA escolhia aquela de maior peso para formular a resposta.

Primeiro o texto de entrada digitado pelo usuário era separado em palavras, em uma técnica que hoje chamamos de tokenização de palavras, mas que ainda não existia na época. A palavra-chave era identificada, comparando-a sequencialmente até o fim das palavras existentes, ou até ser encontrada uma pontuação. Caso fosse encontrada uma pontuação (ponto final ou vírgula), o texto após a pontuação era ignorado se já tivesse sido identificada uma palavra-chave. Assim, cada processamento da resposta utiliza uma única frase do usuário. Se várias palavras-chave fossem encontradas antes da pontuação, a de maior peso era selecionada.

Por exemplo, o usuário entra com o texto: “*I am sick. but, today is raining.*”. Se houvesse uma palavra-chave no script ranqueando a palavra “*SICK*” com alta prioridade, a entrada processada seria somente “*I am sick*”, o restante depois da pontuação (neste caso, o ponto) seria ignorado pelo programa.

Se nenhuma palavra-chave fosse encontrada na entrada, ELIZA recorria a frases genéricas programadas, chamadas de respostas vazias ou sem conteúdo. Nesses casos, o chatbot emitia mensagens do tipo “*I see.*” ou “*Please, go on.*”. Esse mecanismo evitava silêncio quando o usuário dizia algo fora do escopo do script.

Além disso, a implementação original incluía uma estrutura de memória: algumas declarações recentes do usuário eram armazenadas e, se uma entrada subsequente não contivesse novas *keywords*, ELIZA poderia recuperar um tópico anterior e introduzi-lo na conversa. Por exemplo, se o usuário mencionasse

família (em inglês *family*) em um momento e depois fizesse uma afirmação vaga, o programa poderia responder retomando o assunto da família (“*DOES THAT HAVE ANYTHING TO DO WITH YOUR FAMILY?*”). Essa estratégia dava uma pseudo-continuidade ao diálogo, simulando que o sistema “lembrava” de informações fornecidas anteriormente.

2.3 Regras de Transformação

Encontrada a palavra-chave, ELIZA aplicava uma regra de transformação associada a ela para gerar a resposta. As regras são definidas em pares: um padrão de análise (*decomposition rule*) e um modelo de reconstrução de frase (*reassembly rule*).

Primeiro, a frase do usuário é decomposta conforme um padrão que identifica a contexto mínimo em torno da palavra-chave. Essa decomposição frequentemente envolve separar a frase em partes e reconhecer pronomes ou estruturas gramaticais relevantes. Por exemplo, considere a entrada “*You are very helpful.*”. Uma regra de decomposição pode identificar a estrutura “*You are X*” — onde “*X*” representa o restante da frase — e extrair o complemento “*very helpful*” como um componente separado.

Em seguida, a regra de reassembly correspondente é aplicada, remontando uma sentença de resposta em que “*X*” é inserido em um template pré-definido. No exemplo dado, o template de resposta poderia ser “*What makes you think I am X?*”; ao inserir $X = \text{“very helpful”}$, gera-se “*What makes you think I am very helpful?*”. Observe que há uma inversão de pessoa: o pronome “*you*” do usuário foi trocado por “*I*” na resposta do bot.

De fato, uma parte importante das transformações do ELIZA envolve substituir pronomes (eu/você, meu/seu) para que a resposta faça sentido como uma frase do ponto de vista do computador falando com o usuário. Esse algoritmo de substituição é relativamente simples (por exemplo, “meu” \square “seu”, “eu” \square “você”, etc.), mas é essencial para dar a impressão de entendimento gramatical.

Veja no código-fonte abaixo uma parte do conteúdo do arquivo DOCTOR adaptado. A primeira linha REMEMBER 5 estabelece que a prioridade da palavra REMEMBER é 5, caso o usuário entre com uma frase com a palavra REMEMBER, o ELIZA irá responder com uma das perguntas definidas nos parênteses, dentro do

nível da palavra REMEMBER. A segunda lista dentro do REMEMBER são as regras de transformação de frases (chamadas de *decomposition rule* e *reassembly rule*) associadas à palavra-chave REMEMBER. O mesmo se repete para a palavra IF que tem um peso diferente de REMEMBER.

Código-fonte 2.2: Trecho do roteiro DOCTOR do chatbot ELIZA. Adaptado de Weizenbaum [1966].

```
1 (REMEMBER 5
2 (
3   (0 YOU REMEMBER 0)
4   (DO YOU OFTEN THINK OF 4)
5   (DOES THINKING OF ~ BRING ANYTHING ELSE TO MIND)
6   (WHAT ELSE DO YOU REMEMBER)
7   (WHY DO YOU REMEMBER 4 JUST NOW)
8   (WHAT IN THE PRESENT SITUATION REMINDS YOU OF 4)
9   (WHAT IS THE CONNECTION BETWEEN ME AND 4)
10 )
11 (
12   (0 DO I REMEMBER 0)
13   (DID YOU THINK I WOULD FORGET 5)
14   (WHY DO YOU THINK I SHOULD RECALL 5 NOW)
15   (WHAT ABOUT 5)
16   (=WHAT)
17   (YOU MENTIONED 5)
18 )
19 )
20 (IF 3
21 (
22   (0 IF 0)
23   (DO YOU THINK ITS LIKELY THAT 3)
24   (DO YOU WISH THAT 3)
25   (WHAT DO YOU THINK ABOUT 3)
26   (REALLY, 2 3)
27 )
28 )
```

2.4 Implementação e Variações

A implementação original de ELIZA foi feita em uma linguagem chamada MAD-SLIP (um dialeto de Lisp) rodando em um mainframe IBM 7094 no sistema

CTSS do MIT. O código-fonte do programa principal continha o mecanismo de correspondência, enquanto as regras de conversação (script DOCTOR) eram fornecidas separadamente em formato de listas associativas, similar a uma lista em Lisp. Infelizmente, Weizenbaum não publicou o código completo no artigo de 1966 (o que era comum na época), mas décadas depois o código em MAD-SLIP foi recuperado nos arquivos do MIT, comprovando os detalhes de implementação [Lane et al., 2025]. De qualquer forma, a arquitetura descrita no artigo influenciou inúmeras reimplementações acadêmicas e didáticas nos anos seguintes.

Diversos entusiastas e pesquisadores reescreveram ELIZA em outras linguagens de programação, dada a simplicidade relativa de seu algoritmo. Ao longo dos anos, surgiram versões em Lisp, PL/I, BASIC, Pascal, Prolog, Java, Python, OZ, JavaScript, entre muitas outras. Cada versão normalmente incluía o mesmo conjunto de regras do script terapeuta ou pequenas variações.

As ideias de ELIZA também inspiraram chatbots mais avançados. Poucos anos depois, em 1972, surgiu PARRY, escrito pelo psiquiatra Kenneth Colby, que simulava um paciente paranoico. PARRY tinha um modelo interno de estado emocional e atitudes, mas na camada de linguagem ainda usava muitas respostas baseadas em regras, chegando a “conversar” com o próprio ELIZA em experimentos da época.

Em 1995, Richard Wallace desenvolveu o chatbot ALICE (Artificial Linguistic Internet Computer Entity), que levava o paradigma de ELIZA a uma escala muito maior. ALICE utilizava um formato XML chamado AIML (Artificial Intelligence Markup Language) para definir milhares de categorias de padrões e respostas. Com mais de 16.000 templates mapeando entradas para saídas [Wallace, 2000], ALICE conseguia manter diálogos bem mais naturais e abrangentes que o ELIZA original, embora o princípio básico de correspondência de padrões permanecesse. Esse avanço rendeu a ALICE três vitórias no Prêmio Loebner (competição de chatbots) no início dos anos 2000 [Wallace, 2000].

Outras variações e sucessores notáveis incluem Jabberwacky (1988) – que já aprendia novas frases – e uma profusão de assistentes virtuais e bots de domínio específico nas décadas seguintes [Wallace, 2000]. Em suma, o legado de ELIZA perdurou por meio de inúmeros chatbots baseados em regras, até a transição para abordagens estatísticas e de aprendizado de máquina no final do século XX.

2.5 Mecanismo de Pesos

A técnica de ELIZA, baseada em palavras-chave com respostas predefinidas, contrasta fortemente com os métodos de Modelos de Linguagem Grande (LLMs) atuais, como o *Generative Pre-trained Transformer* (GPT), que utilizam redes neurais de milhões (ou trilhões) de parâmetros e mecanismos de atenção. Mais detalhes sobre LLM no Capítulo 4.

No ELIZA, a “importância” de uma palavra era determinada manualmente pelo programador através de pesos ou rankings atribuídos a certas palavras-chave no script. Ou seja, o programa não aprendia quais termos focar – ele seguia uma lista fixa de gatilhos. Por exemplo, termos como “sempre” ou “igual” tinham prioridade alta no script DOCTOR para garantir respostas apropriadas.

Em contraste, modelos modernos como o GPT não possuem uma lista fixa de palavras importantes; em vez disso, eles utilizam o mecanismo de *self-attention* para calcular dinamicamente pesos entre todas as palavras da entrada conforme o contexto [Vaswani et al., 2017].

Na arquitetura *Transformer*, cada palavra (na prática não é uma palavra e sim um *token*) de entrada gera consultas e chaves que interagem com todas as outras, permitindo ao modelo atribuir pesos maiores às palavras mais relevantes daquela frase ou parágrafo [Vaswani et al., 2017]. Em outras palavras, o modelo aprende sozinho quais termos ou sequências devem receber mais atenção para produzir a próxima palavra na resposta. Esse mecanismo de atenção captura dependências de longo alcance e nuances contextuais que um sistema de palavras-chave fixas como o ELIZA não consegue representar.

Além disso, o “vocabulário” efetivo de um LLM é imenso – um modelo GPT pode ser treinado com trilhões de palavras e ter ajustado seus parâmetros para modelar estatisticamente a linguagem humana [Vaswani et al., 2017]. Como resultado, pode-se dizer metaforicamente que os LLMs têm uma lista de “palavras-chave” milhões de vezes maior (na prática, distribuída em vetores contínuos) e um método bem mais sofisticado de calcular respostas do que o ELIZA.

Enquanto ELIZA dependia de coincidências exatas de termos para disparar regras, modelos como GPT avaliam similaridades semânticas e contexto histórico graças às representações densas (embeddings) aprendidas durante o treinamento de rede neural.

2.6 Geração de Texto

Devido à sua abordagem baseada em regras locais, o ELIZA tinha capacidade de contextualização muito limitada. Cada input do usuário era tratado quase isoladamente: o programa não construiu uma representação acumulada da conversa, além de artifícios simples como repetir algo mencionado (a estrutura de memória) ou usar pronomes para manter a ilusão de continuidade. Se o usuário mudasse de tópico abruptamente, o ELIZA não “perceberia” – ele apenas buscaria a próxima palavra-chave disponível ou recorreria a frases genéricas.

Em contraste, modelos de linguagem modernos levam em conta um longo histórico de diálogo. Chatbots que usam GPT podem manter um contexto de centenas ou milhares de tokens (palavras ou fragmentos) em sua janela de atenção, o que significa que eles conseguem referenciar informações mencionadas vários parágrafos atrás e integrá-las na resposta corrente. O mecanismo de self-attention, em particular, permite que o modelo incorpore relações contextuais complexas: cada palavra gerada pode considerar influências de palavras distantes no texto de entrada [Vaswani et al., 2017].

Por exemplo, ao conversar com um LLM, se você mencionar no início da conversa que tem um irmão chamado Alex e depois perguntar “ele pode me ajudar com o problema?”, o modelo entenderá que “ele” se refere ao Alex mencionado anteriormente (desde que dentro da janela de contexto). Já o ELIZA original não teria como fazer essa ligação, a menos que houvesse uma regra explícita para “ele” e algum armazenamento específico do nome – algo impraticável de antecipar via regras fixas para todos os casos.

Outra diferença está na geração de linguagem. O ELIZA não gera texto original no sentido pleno: suas respostas são em grande parte frases prontas (ou templates fixos) embaralhadas com partes da fala do usuário. Assim, seu vocabulário e estilo são limitados pelo script escrito manualmente. Modelos GPT, por sua vez, geram respostas novas combinando probabilisticamente o conhecimento adquirido de um extenso corpus. Eles não se restringem a repetir trechos da entrada, podendo elaborar explicações, fazer analogias, criar perguntas coerentes com os exemplos linguísticos em sua base de treinamento. Enquanto ELIZA tendia a responder com perguntas genéricas ou devolvendo as palavras do usuário, os LLMs podem produzir respostas informativas e detalhadas sobre o assunto (pois “aprenderam” uma ampla gama de tópicos durante o treinamento).

Por exemplo, se perguntarmos algo factual ou complexo, o ELIZA falharia por não ter nenhuma regra a respeito, provavelmente dando uma resposta vazia. Já um modelo como GPT tentará formular uma resposta baseada em padrões linguísticos aprendidos e em conhecimento implícito dos dados, muitas vezes fornecendo detalhes relevantes.

Em termos de fluência e variedade, os modelos modernos superam o ELIZA amplamente. O ELIZA frequentemente se repetia ou caía em loops verbais quando confrontado com inputs fora do roteiro – um limite claro de sistemas por regras estáticas. Os LLMs produzem linguagem muito mais natural e adaptável, a ponto de muitas vezes enganarem os usuários sobre estarem conversando com uma máquina (um efeito buscado desde o Teste de Turing). Ironicamente, ELIZA nos anos 60 já provocou um precursor desse fenômeno – o chamado *Efeito ELIZA*, em que pessoas atribuem compreensão ou sentimentos a respostas de computador que, na verdade, são superficiais. Hoje, em chatbots GPT, esse efeito se intensifica pela qualidade das respostas, mas a distinção fundamental permanece: ELIZA seguia scripts sem compreender, enquanto LLMs inferem padrões e significados de forma estatística, sem entendimento consciente, mas atingindo resultados que simulam compreensão de maneira muito mais convincente.

Em resumo, os avanços de arquitetura (especialmente o mecanismo de atenção) ampliaram drasticamente a capacidade de contextualização e geração dos chatbots modernos, marcando uma evolução significativa desde o mecanismo simples, porém pioneiro, de ELIZA.

2.7 ELIZA em Python

A seguir o código-fonte de um programa que retorna o que o usuário digitou na linguagem de programação Python, inspirado no paradigma ELIZA.

Código-fonte 2.3: Código-fonte de um programa inspirado no ELIZA.

```
1 # método que processa a entrada do usuário
2 def response(user_input):
3     return "Você disse: " + user_input
4
5 # Exemplo de uso
6 user_input = "Eu estou feliz"
7 print("Você: {}".format(user_input))
```

```

8 print("Eliza: {}".format(response(user_input)))
9 user_input = "Eu estou alegre"
10 print("Você: {}".format(user_input))
11 print("Eliza: {}".format(response(user_input)))

```

```

1 Você: Eu estou feliz
2 Eliza: Você disse: Eu estou feliz
3 Você: Eu estou alegre
4 Eliza: Você disse: Eu estou alegre

```

Este código acima simplesmente repete o que o usuário digita. Ele define a função *response*, que retorna a string “Você disse: ” concatenada ao texto recebido.

Logo abaixo uma implementação um pouco mais robusta de um chatbot inspirado no paradigma ELIZA. Esta implementação demonstra a utilização de expressões regulares - mais sobre expressões regulares no Capítulo 3 - para a identificação de padrões textuais (palavras-chave) na entrada fornecida pelo usuário e a subsequente geração de respostas, fundamentada em regras de transformação predefinidas manualmente.

Código-fonte 2.4: Chatbot Eliza em Python.

```

1 import re
2 import random
3
4 regras = [
5     (re.compile(r'\b(hello|hi|hey)\b', re.IGNORECASE),
6      ["Hello. How do you do. Please tell me your problem."]), # regra 1
7     (re.compile(r'\b(I am|I\'?m) (.+)\', re.IGNORECASE), # regra 2
8      ["How long have you been {1}?",
9       "Why do you think you are {1}?"]),
10    (re.compile(r'\bI need (.+)\', re.IGNORECASE), # regra 3
11     ["Why do you need {1}?",
12      "Would it really help you to get {1}?"]),
13    (re.compile(r'\bI can\'?t (.+)\', re.IGNORECASE), # regra 4
14     ["What makes you think you can't {1}?",
15      "Have you tried {1}?"]),
16    (re.compile(r'\bmy (mother|father|mom|dad)\b', re.IGNORECASE), #
17     regra 5
18     ["Tell me more about your family.",
19      "How do you feel about your parents?"]),
19    (re.compile(r'\b(sorry)\b', re.IGNORECASE), # regra 6

```

```

20     ["Please don't apologize."]),
21     (re.compile(r'\b(maybe|perhaps)\b', re.IGNORECASE), # regra 7
22     ["You don't seem certain."]),
23     (re.compile(r'\bbebecause\b', re.IGNORECASE), # regra 8
24     ["Is that the real reason?"]),
25     (re.compile(r'\b(are you|do you) (.+)\?$', re.IGNORECASE), # regra 9
26     ["Why do you ask that?"]),
27     (re.compile(r'\bcomputer\b', re.IGNORECASE), # regra 10
28     ["Do computers worry you?"]),
29 ]
30
31 respostas_padrao = [
32     "I see.",
33     "Please tell me more.",
34     "Can you elaborate on that?"
35 ]
36
37 def response(entrada_usuario):
38     for padrao, respostas in regras:
39         match = padrao.search( entrada_usuario)
40         if match:
41             resposta = random.choice(respostas)
42             if match.groups():
43                 resposta = resposta.format( *match.groups())
44             return resposta
45     return random.choice(respostas_padrao)
46
47 print("User: Hello.")
48 print("Bot: " + response("Hello."))
49 print("User: I am feeling sad.")
50 print("Bot: " + response("I am feeling sad."))
51 print("Maybe I was not good enough.")
52 print("Bot: " + response("Maybe I was not good enough."))
53 print("My mother tried to help.")
54 print("Bot: " + response("My mother tried to help."))

```

```

1 User: Hello.
2 Bot: Hello. How do you do. Please tell me your problem.
3 User: I am feeling sad.
4 Bot: How long have you been feeling sad.?
5 Maybe I was not good enough.
6 Bot: You don't seem certain.
7 My mother tried to help.
8 Bot: How do you feel about your parents?

```

Na implementação anterior, são definidos múltiplos padrões de expressões regulares que correspondem a palavras-chave ou estruturas frasais de interesse (e.g., saudações, construções como “I am” ou “I need”, referências a termos familiares). A função `response`, ao receber uma string de entrada, itera sequencialmente sobre essas regras.

Quando uma correspondência (*match*) é encontrada, uma resposta associada à regra é selecionada. É relevante notar que diversas respostas predefinidas contêm marcadores (*placeholders*), como `{1}`. Estes marcadores são dinamicamente substituídos pelo conteúdo dos grupos capturados pela expressão regular (mais detalhes sobre expressão regular no Capítulo 3 na entrada original do usuário). Por exemplo, se a entrada for “*I am sad*” e esta corresponder a um padrão como `r` `I am (.*)'``, o grupo capturado `((.*))` conterá a string “*sad*”. Se a resposta associada for “*Why do you think you are {1}?*”, o sistema a formatará como “*Why do you think you are sad?*”, substituindo `{1}` pelo conteúdo do primeiro grupo capturado. Na ausência de correspondência com qualquer regra específica, o sistema seleciona aleatoriamente uma resposta genérica a partir de um conjunto predefinido (denominado, por exemplo, `respostas_padrao`), tal como “*I see.*”.

Código-fonte 2.5: Exemplo de Interação.

```
1 Usuário: Hello
2 Bot: Hello. How do you do. Please tell me your problem.
3 Usuário: I am feeling sad
4 Bot: Why do you think you are feeling sad?
5 Usuário: Because I lost my job
6 Bot: Is that the real reason?
7 Usuário: Maybe I was not good enough.
8 Bot: You don't seem certain.
9 Usuário: My mother tried to help.
10 Bot: Tell me more about your family.
```

Observa-se na interação que o chatbot inicialmente respondeu à saudação (“*Hello...*”) com base na regra 1. Em seguida, a entrada “*I am sad*” ativou a regra 2, resultando em uma resposta que incorpora o termo “*sad*”. A interação prossegue com o sistema respondendo a entradas iniciadas por “*Because...*” (regra 8) e “*Maybe...*” (regra 7), entre outras. Cada resposta gerada foi determinada pela

regra correspondente ativada e pelo conteúdo específico da sentença do usuário, mimetizando o comportamento baseado em padrões do sistema ELIZA original [Abdul-Kader and Woods, 2015].

2.8 Criando um ELIZA Simples com Interface e Gradio

Nesta seção, vamos explorar como criar um chatbot usando a biblioteca Gradio e Python.

O Gradio é uma biblioteca de código aberto que permite criar interfaces de usuário para modelos de machine learning de forma rápida e fácil. Com Gradio, você pode criar aplicativos web interativos para visualizar e testar seus modelos, sem precisar de conhecimentos em frontend ou backend.

Antes de começar, certifique-se de ter o Python e o pip instalados em sua máquina. Em seguida, instale a biblioteca Gradio usando o comando:

Código-fonte 2.6: Instalação do Gradio.

```
1 pip install gradio
```

À seguir está um exemplo básico de como criar um chatbot com Gradio e Python:

Código-fonte 2.7: Exemplo de código para criar um chatbot simples.

```
1 import gradio as gr
2
3 def chatbot(message, history):
4     resposta = "Olá! Eu sou um chatbot. Como posso ajudar você?"
5     return resposta
6
7 demo = gr.ChatInterface(
8     fn=chatbot,
9     title="Chatbot Simples"
10 )
11
12 demo.launch()
```


Este código define uma função chatbot que processa o texto de entrada e retorna uma resposta. Em seguida, cria uma interface do chatbot usando a biblioteca Gradio, com um campo de texto para entrada e outro para saída.

Execute o chatbot: Salve o arquivo e, no terminal, navegue até o diretório onde ele está salvo. Digite:

Código-fonte 2.8: Instalação do Gradio.

```
1 python chatbot.py
```

Você verá um link local (algo como <http://127.0.0.1:7860>). Clique nele ou copie e cole no navegador. Uma interface simples aparecerá com um campo de texto. Experimente digitar “olá” ou “tchau” e veja as respostas. Veja na Figura 2.2 um print da tela do chatbot já em execução.

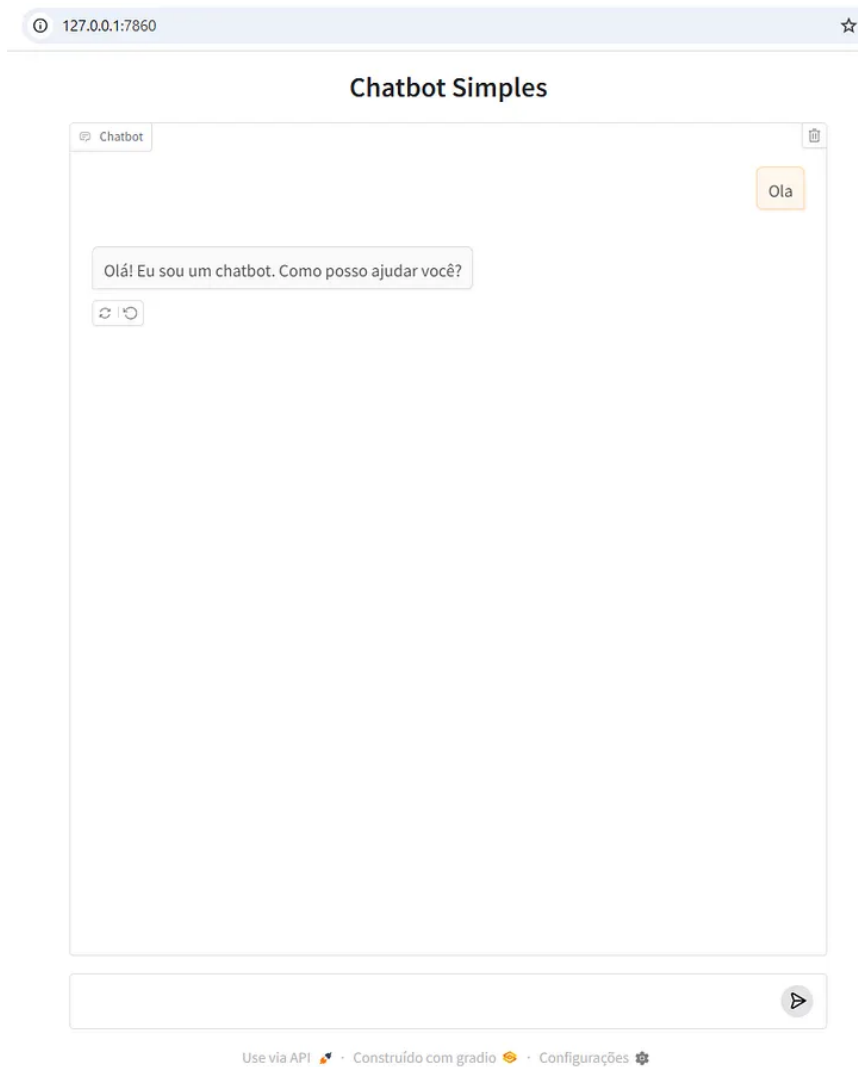
Adicionando Lógica ao Chatbot: Agora que você tem uma interface básica, é hora de adicionar lógica ao chatbot. Você pode fazer isso adicionando condições e processamento de texto à função chatbot. Por exemplo:

Código-fonte 2.9: Chatbot simples com Gradio.

```
1 # pip install gradio
2 import gradio as gr
3
4 def chatbot(message, history):
5     if "Olá" in message:
6         resposta = "Olá! Eu sou um chatbot. Como posso ajudar você?"
7     elif "Quem é você?" in message:
8         resposta = "Eu sou um chatbot criado para ajudar você com suas
9             perguntas."
10    else:
11        resposta = "Desculpe, não entendi sua pergunta. Pode tentar
12            novamente?"
13    return resposta
14
15 demo = gr.ChatInterface(
16     fn=chatbot,
17     title="Chatbot Simples"
18 )
19
20 demo.launch()
```

Na Figura 2.3 é possível visualizar o print da tela de um chatbot que responde de forma aleatória. Logo abaixo, o código-fonte que deu origem ao chatbot.

Figura 2.2: Print do Chatbot.

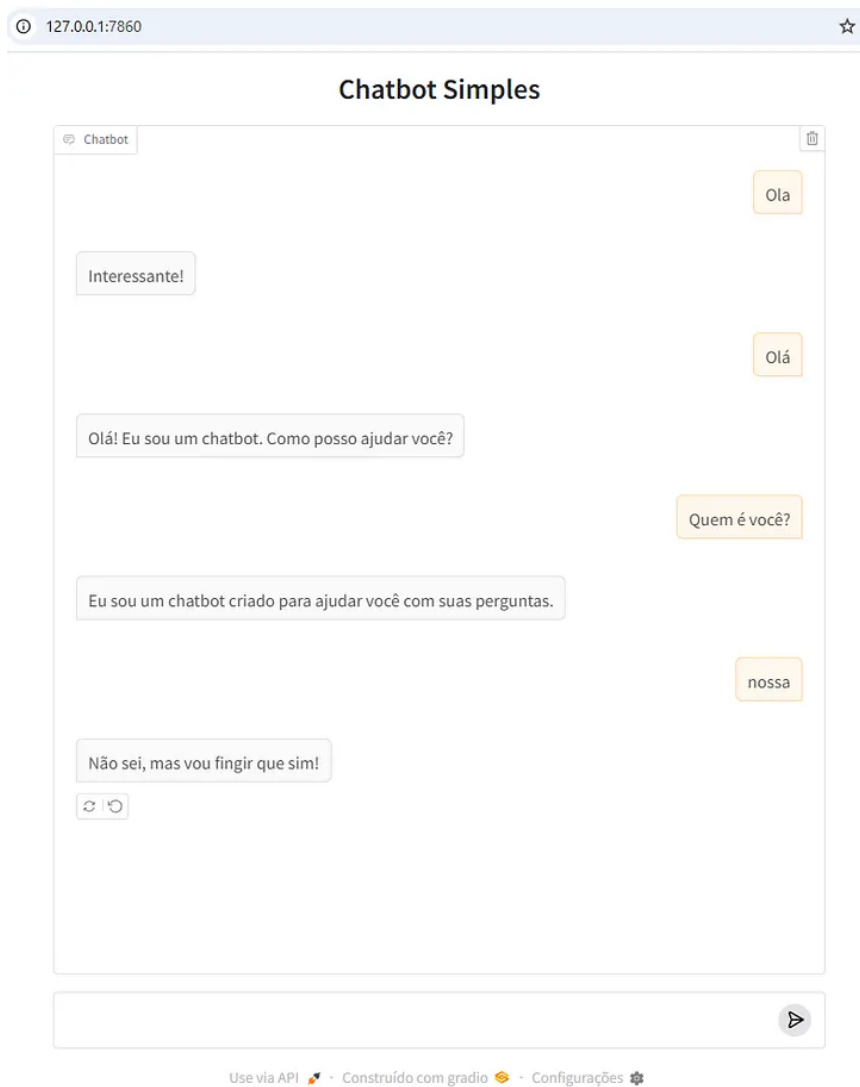


Código-fonte 2.10: Chatbot simples com Gradio com respostas aleatórias.

```
1 # pip install gradio
2 import gradio as gr
3 import random
4
5 def chatbot(message, history):
6     respostas = ["Interessante!", "Hmm, me conte mais!", "Não sei, mas
7         vou fingir que sim!"]
8     if "Olá" in message:
9         resposta = "Olá! Eu sou um chatbot. Como posso ajudar você?"
10    elif "Quem é você?" in message:
11        resposta = "Eu sou um chatbot criado para ajudar você com suas
12            perguntas."
13    else:
14        resposta = random.choice(respostas)
15    return resposta
16
17 demo = gr.ChatInterface(
18     fn=chatbot,
19     title="Chatbot Simples"
20 )
21
22 demo.launch()
```

Criar um chatbot com Gradio e Python é uma tarefa relativamente simples e rápida. Com essa biblioteca, você pode criar interfaces de usuário interativas para seus modelos de aprendizagem de máquina e criar chatbots para automação de tarefas ou suporte ao cliente. Este é apenas um exemplo básico, e você pode adicionar mais lógica e funcionalidades ao seu chatbot para torná-lo mais útil.

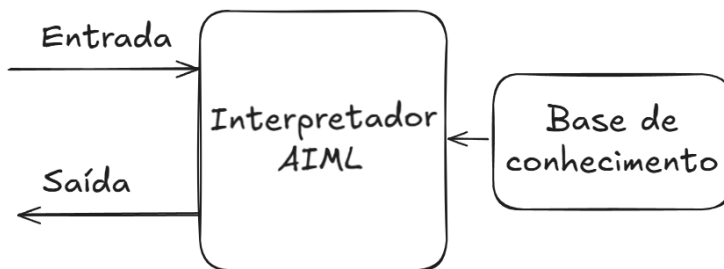
Figura 2.3: Print do chatbot com respostas aleatórias e um pouco de lógica Conclusão.



2.9 Artificial Intelligence Markup Language

Décadas depois do Eliza, as técnicas de programação avançaram, surgiu a web e as linguagens de marcação como o HTML ganharam tração. É neste cenário que é criado o Artificial Intelligence Markup Language (AIML), uma especificação baseada em XML, proposta por Wallace [2009], destinada à programação de chatbots. A concepção da linguagem prioriza o minimalismo, característica que simplifica o processo de criação de bases de conhecimento por indivíduos sem experiência prévia em programação [Wallace, 2009]. A arquitetura fundamental de um interpretador AIML genérico é ilustrada na Figura 2.4.

Figura 2.4: Interpretador AIML arquitetura.
Adaptado de [da Silva and Costa, 2007]

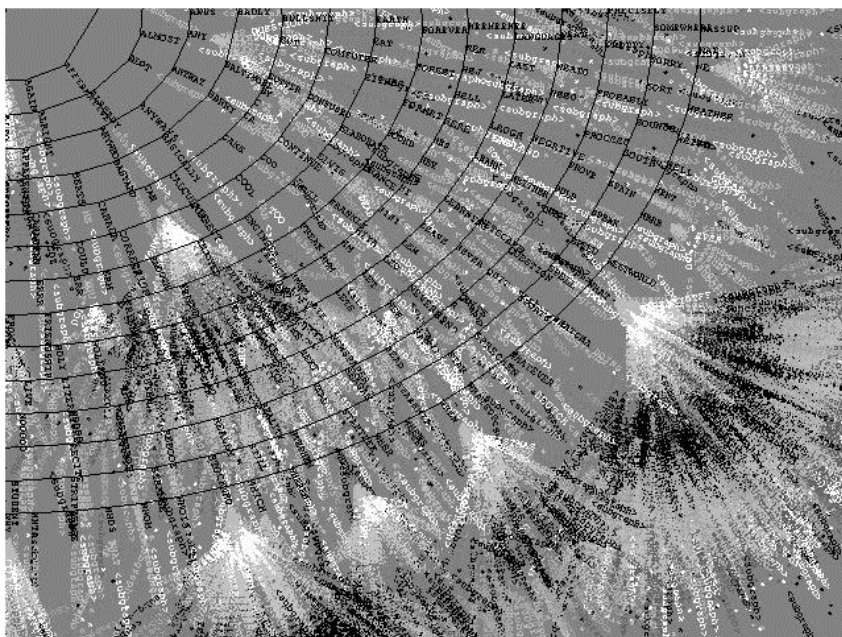


A técnica central empregada pelo AIML é a correspondência de padrões (*pattern matching*). Este método é amplamente utilizado no desenvolvimento de chatbots, particularmente em sistemas orientados a perguntas e respostas [Abdul-Kader and Woods, 2015]. Uma das metas de projeto do AIML é possibilitar a fusão de bases de conhecimento de múltiplos chatbots especializados em domínios distintos. Teoricamente, um interpretador poderia agregar essas bases, eliminando automaticamente categorias redundantes para formar um *chatbot* mais abrangente [Wallace, 2000].

AIML é frequentemente associado aos chatbots de terceira geração [Mária et al., 2010] e estima-se sua adoção em mais de 50.000 implementações em diversos idiomas. Extensões da linguagem foram propostas, como o iAIML, que introduziu novas *tags* e incorporou o conceito de intenção com base nos princípios da Teoria da Análise da Conversação [Neves and Barros, 2005]. Adicionalmente, ferramentas baseadas na Web foram desenvolvidas para apoiar

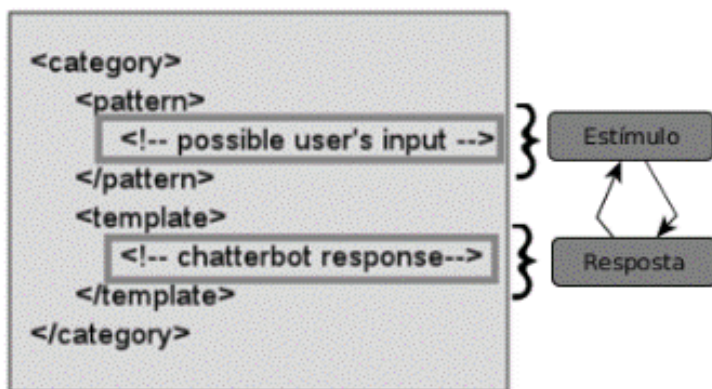
a construção de bases de conhecimento AIML [Krassmann et al., 2017]. Um exemplo proeminente é o *chatbot* ALICE, cuja implementação em AIML compreendia aproximadamente 16.000 categorias, cada uma potencialmente contendo múltiplas *tags* XML aninhadas [Wallace, 2000]. Uma representação visual desta estrutura de conhecimento é apresentada na Figura 2.5.

Figura 2.5: Representação visual da base de conhecimento do chatbot ALICE.
Retirado de [Wallace, 2003]



Wallace [2000] estabeleceu analogias entre o funcionamento de interpretadores AIML e a teoria do Raciocínio Baseado em Casos (RBC). Nessa perspectiva, as categorias AIML funcionam como “casos”, onde o algoritmo identifica o padrão que melhor se alinha à entrada do usuário. Cada categoria estabelece um vínculo direto entre um padrão de estímulo e um modelo de resposta. Consequentemente, chatbots AIML inserem-se na tradição da robótica minimalista, reativa ou de estímulo-resposta [Wallace, 2000], conforme esquematizado na Figura 2.6. Vale notar que a própria técnica de RBC já foi integrada a interpretadores AIML como um mecanismo para consultar fontes de dados externas e expandir a base de conhecimento do agente [Kraus and Fernandes, 2008].

Figura 2.6: Teoria estímulo-resposta aplicada no AIML.
Retirado de [Lima, 2017].



Os chatbots que utilizam AIML são classificados como sistemas “baseados em recuperação” (retrieval-based). Tais modelos operam a partir de um repositório de respostas predefinidas, selecionando a mais apropriada com base na entrada do usuário e no contexto conversacional, guiando assim o fluxo da interação. Esta abordagem é frequentemente empregada na construção de chatbots destinados a operar em domínios de conhecimento restritos [Borah et al., 2018].

O código-fonte à seguir, demonstra a estrutura elementar de um arquivo AIML. A tag `<category>` encapsula a unidade básica de conhecimento. Internamente, a tag `<pattern>` define o padrão de entrada a ser reconhecido (no exemplo, o caractere curinga `*`, que corresponde a qualquer entrada), enquanto a tag `<template>` contém a resposta associada. No exemplo ilustrado, o *chatbot* responderia “Hello!” a qualquer interação. Uma visão abstrata da árvore de conhecimento resultante pode ser observada logo abaixo. O AIML padrão suporta transições baseadas primariamente em correspondência de padrões, uma limitação inerente, embora extensões específicas de interpretadores possam permitir a integração de outras técnicas de processamento.

Código-fonte 2.11: Exemplo de uma base de conhecimento em AIML. Adaptado de [Wallace, 2000].

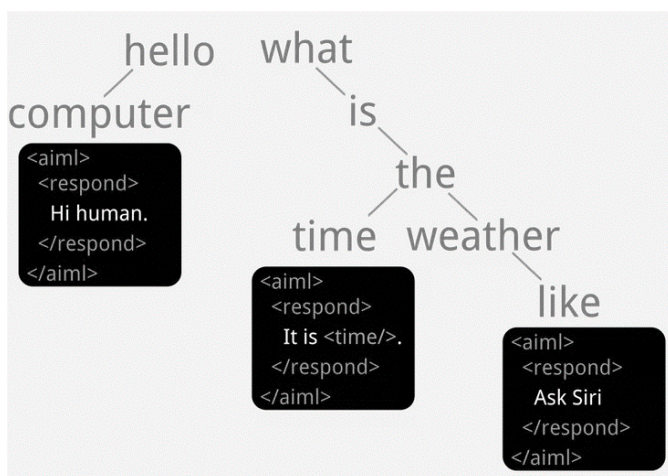
1 `<aiml>`

```

2 <category>
3   <pattern>*/</pattern>
4   <template>Hello!</template>
5 </category>
6 </aiml>

```

Figura 2.7: Representação visual abstrata de uma base de conhecimento AIML.
Retirado de <https://www.pandorabots.com/docs/aiml-fundamentals/>



O profissional responsável pela criação, manutenção e curadoria da base de conhecimento de um *chatbot* AIML é denominado *botmaster* [Wallace, 2000]. Suas atribuições englobam a edição da base (frequentemente via ferramentas auxiliares), a análise de logs de diálogo para identificar padrões de interação e a subsequente criação ou refino de respostas. Este papel pode ser exercido por indivíduos com diferentes perfis, incluindo *webmasters*, desenvolvedores, redatores, engenheiros ou outros interessados na construção de chatbots [Wallace, 2000].

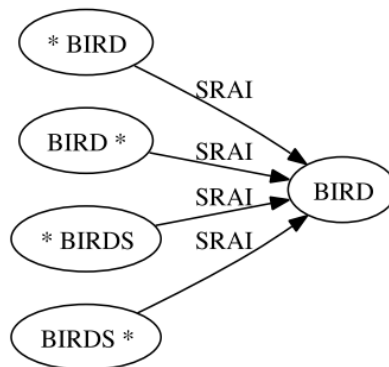
Algumas implementações de interpretadores AIML podem incorporar capacidades rudimentares de compreensão semântica através do *Resource Description Framework* (RDF)¹. O RDF é um padrão W3C para representação de informações na Web, usualmente por meio de triplas (sujeito-predicado-objeto) que descrevem relações entre entidades. No contexto AIML, RDF pode ser utilizado para armazenar e consultar fatos. Contudo, mesmo com tais adições, as capacidades

¹<https://github.com/keiffster/program-y/wiki/RDF>

linguísticas permanecem aquém da complexidade e do potencial gerativo da linguagem humana, conforme descrito por Chomsky and Lightfoot [2002].

Embora Höhn [2019] argumente que o AIML padrão carece de um conceito explícito de “intenção” (*intent*), similar ao encontrado em plataformas de *Natural Language Understanding* (NLU), é possível emular o reconhecimento de intenções. Isso é tipicamente alcançado definindo categorias que representam “formas canônicas” ou “padrões atômicos” para uma intenção específica². Variações de entrada (e.g., “oi”, “olá”) podem ser mapeadas para uma categoria canônica (e.g., “saudação”) usando a tag <srail> (*Symbolic Reduction Artificial Intelligence*), que redireciona o fluxo de processamento (ver Figura 2.8). Dessa forma, um *chatbot* AIML pode gerenciar intenções distintas dentro de seu domínio, como realizar um pedido ou verificar o status de entrega.

Figura 2.8: Uso da tag <srail>.
Retirado de [De Gasperis et al., 2013]



Os chatbots baseados em AIML têm obtido sucesso significativo em competições como o Prêmio Loebner. Notavelmente, o chatbot Mitsuku³, desenvolvido por Steve Worswick, conquistou múltiplos títulos⁴, seguindo vitórias anteriores do ALICE [Wallace, 2000].

²<https://medium.com/pandorabots-blog/new-feature-visualize-your-aiml-26e33a590>

³<https://www.pandorabots.com/mitsuku/>

⁴<https://aisb.org.uk/category/loebner-prize/>

Adicionalmente, Mitsuku foi classificado em primeiro lugar numa análise comparativa envolvendo oito chatbots [Sharma et al., 2020]. Nesse estudo, que avaliou atributos conversacionais com base em um conjunto padronizado de perguntas, o Google Assistant obteve a segunda posição, seguido pela Siri em terceiro. O *chatbot* ALICE alcançou a quarta posição, enquanto o ELIZA ficou na última colocação entre os sistemas comparados [Sharma et al., 2020].

2.9.1 Tags do AIML

Esta seção descreve as principais tags do AIML, versão 1.0.

`<aiml>`

No contexto de AIML (Artificial Intelligence Markup Language), a tag `<aiml>` é usada para definir o início de um documento AIML que contém os padrões e respostas que um motor de chatbot deve usar. Ela envolve todo o documento, indicando que o conteúdo entre as tags `<aiml>` é escrito em AIML.

Código-fonte 2.12: Tag raiz que engloba todo o conteúdo AIML.

```
1 <aiml version="1.0">
2   <!-- Categorias aqui -->
3 </aiml>
```

`<category>`

Descrição: Unidade básica de conhecimento, contendo um padrão e uma resposta.

Código-fonte 2.13: Unidade básica de conhecimento, contendo um padrão e uma resposta.

```
1 <category>
2   <pattern>OLÁ</pattern>
3   <template>Oi! Como posso ajudar você hoje?</template>
4 </category>
```

<pattern>

Descrição: Define o padrão de entrada do usuário, com curingas como * e

—.

Código-fonte 2.14: Define o padrão de entrada do usuário, com curingas.

```
1 <category>
2   <pattern>EU GOSTO DE *</pattern>
3   <template>Que bom que você gosta de <star/>!</template>
4 </category>
```

<template>

Descrição: Define a resposta do bot ao padrão correspondente.

Código-fonte 2.15: Define a resposta do bot ao padrão correspondente.

```
1 <category>
2   <pattern>QUAL É O SEU NOME</pattern>
3   <template>Meu nome é neo chatbot.</template>
4 </category>
```

<star/>

Descrição: Captura o conteúdo do curinga * ou _.

Código-fonte 2.16: Captura o conteúdo do curinga.

```
1 <category>
2   <pattern>MEU NOME É *</pattern>
3   <template>Olá, <star/>!</template>
4 </category>
```

<that>

Descrição: Considera a última resposta do bot para decidir a próxima.

Código-fonte 2.17: Considera a última resposta do bot para decidir a próxima.

```
1 <category>
2   <pattern>SIM</pattern>
```

```
3 <that>Você gosta de programar?</that>
4 <template>Ótimo! Qual linguagem você prefere?</template>
5 </category>
```

<topic>

Descrição: Define um contexto ou tópico para categorias.

Código-fonte 2.18: Define um contexto ou tópico para categorias.

```
1 <category>
2 <pattern>VAMOS FALAR SOBRE ESPORTE</pattern>
3 <template>Ok! <topic name="esporte" /></template>
4 </category>
```

<random> e

Descrição: Escolhe aleatoriamente uma resposta de uma lista.

Código-fonte 2.19: Escolhe aleatoriamente uma resposta de uma lista.

```
1 <category>
2 <pattern>COMO ESTÁ O TEMPO</pattern>
3 <template>
4 <random>
5 <li>Está ensolarado!</li>
6 <li>Está chovendo.</li>
7 </random>
8 </template>
9 </category>
```

<condition>

Descrição: Adiciona lógica condicional baseada em variáveis.

Código-fonte 2.20: Adiciona lógica condicional baseada em variáveis.

```
1 <category>
2 <pattern>COMO EU ESTOU</pattern>
3 <template>
4 <condition name="humor">
5 <li value="feliz">Você está bem!</li>
```

```
6      <li>Não sei ainda!</li>
7      </condition>
8      </template>
9  </category>
```

`<set> e <get>`

Descrição: Define e recupera variáveis.

Código-fonte 2.21: Define e recupera variáveis.

```
1 <category>
2   <pattern>MEU NOME É *</pattern>
3   <template>
4     <set name="nome"><star/></set>Olá, <get name="nome"/>!
5   </template>
6 </category>
```

`<srai>`

Descrição: Redireciona a entrada para outro padrão.

Código-fonte 2.22: Redireciona a entrada para outro padrão.

```
1 <category>
2   <pattern>OI</pattern>
3   <template><srai>OLÁ</srai></template>
4 </category>
```

`<think>`

Descrição: Executa ações sem exibir o conteúdo.

Código-fonte 2.23: Executa ações sem exibir o conteúdo.

```
1 <category>
2   <pattern>EU SOU TRISTE</pattern>
3   <template>
4     <think><set name="humor">triste</set></think>Sinto muito!
5   </template>
6 </category>
```

<person>, <person2>, <gender>

Descrição: Transforma pronomes ou ajusta gênero.

Código-fonte 2.24: Transforma pronomes ou ajusta gênero.

```
1 <category>
2   <pattern>EU TE AMO</pattern>
3   <template><person><star/></person> ama você também!</template>
4 </category>
```

<formal>, <uppercase>, <lowercase>

Descrição: Formata texto (capitaliza, maiúsculas, minúsculas).

Código-fonte 2.25: Formata texto (capitaliza, maiúsculas, minúsculas).

```
1 <category>
2   <pattern>MEU NOME É joão</pattern>
3   <template>Olá, <formal><star/></formal>!</template>
4 </category>
```

<sentence>

Descrição: Formata como frase (primeira letra maiúscula, ponto final).

Código-fonte 2.26: Formata como frase (primeira letra maiúscula, ponto final).

```
1 <category>
2   <pattern>oi</pattern>
3   <template><sentence><star/></sentence></template>
4 </category>
```

2.9.2 Exemplo em Python

A seguir um exemplo do uso de um interpretador AIML em Python. O arquivo “cerebro.aiml” deve existir anteriormente. Use uma versão compatível com a biblioteca aiml que é somente compatível com versões antigas do Python, do Python 3.6 para trás, ou seja, não funciona no Python 3.12.

Código-fonte 2.27: Um chatbot de terminal que responde a algumas palavras-chave pré-definidas.

```

1 # pip install aiml
2 import aiml
3 # Criar kernel (núcleo do bot)
4 kernel = aiml.Kernel()
5 cerebro_aiml_text = """
6 <aiml version="1.0.1" encoding="UTF-8">
7   <category>
8     <pattern>OI</pattern>
9     <template>Olá! Como posso ajudar você?</template>
10  </category>
11  <category>
12    <pattern>OBRIGADO</pattern>
13    <template>De nada!</template>
14  </category>
15 </aiml>
16 """
17
18 # Salvar o conteúdo AIML em um arquivo
19 with open("cerebro.aiml", "w", encoding="utf-8") as f:
20     f.write(cerebro_aiml_text)
21 # Carregar o arquivo AIML
22 kernel.learn("cerebro.aiml")
23
24 # Loop de conversa
25 while True:
26     user_input = input("Você: ")
27     if user_input.lower() in ["sair", "exit", "quit"]:
28         break
29     response = kernel.respond(user_input)
30     print("Bot:", response)

```

```

1 Você: oi
2 Bot: Olá! Como posso ajudar você?

```

O arquivo std-startup.xml é um ponto de partida e geralmente carrega outros arquivos .aiml.

Código-fonte 2.28: Estrutura básica de um arquivo cerebro.aiml.

```

1 <aiml version="1.0.1" encoding="UTF-8">
2   <category>
3     <pattern>OI</pattern>

```

```
4      <template>Olá! Como posso te ajudar?</template>
5  </category>
6  <category>
7      <pattern>QUAL SEU NOME</pattern>
8      <template>Eu sou um chatbot em AIML.</template>
9  </category>
10 </aiml>
```


Capítulo 3

Processamento de Linguagem Natural (PLN)

Os ignorantes afirmam, os sábios
duvidam, os sensatos refletem.

Aristóteles

Objetivo

Introduzir técnicas essenciais de PLN, como tokenização, lematização, POS tagging e NER, para capacitar o leitor a processar e analisar linguagem natural em projetos de chatbot; demonstrar o uso de expressões regulares; apresentar métodos de representação de texto, como Bag-of-Words, TF-IDF e embeddings; além disso, descrever como configurar o Python para executar os códigos.

3.1 Inteligência Artificial

A inteligência artificial (IA) é a força tecnológica mais transformadora do século XXI [Ribeiro, 2025]. Mas o que é IA? As definições de IA dependem do contexto e isso pode trazer confusão no entendimento e delimitação do tema. Menos abrangente, porém mais confuso ainda, é o termo “inteligência artificial”. Portanto, dado as diversas definições de inteligência artificial (IA), ou *artificial intelligence* em inglês, delimitaremos um pouco o escopo da inteligência em questão.

A IA aparece em nossa cultura de diversas formas, tais como, o HAL 9000 do filme “2001: uma Odisseia no Espaço”, clássico de Stanley Kubrick, ou como a IA do filme “Ela”, com o ator Joaquin Phoenix, onde um humano se apaixona por um sistema operacional.

Espero que você leitor seja um membro da espécie Homo-Sapiens. O termo “Homo-Sapiens” vem do latim e significa homem sábio [Wikipedia, 2024c]. A importância da sapiência (sinônimo de inteligência) é tamanha que define a nossa espécie. Porém, neste contexto, consideramos que um gato ou cachorro também é dotado de inteligência. Uma abelha é praticamente uma cientista [Wikipedia, 2024a]. Portanto, seremos mais contidos e reservados quanto ao significado do termo inteligência.

O que confunde é que inteligência e artificial são palavras que têm significado implícito para pessoas que não são da área de computação. Naturalmente, médicos, advogados, engenheiros (só para citar alguns) querem verificar como a “inteligência artificial” pode ser inserida na sua rotina diária. Meu dentista já quis saber como a IA iria afetar seus procedimentos odontológicos. Porém, ele nunca me perguntou em como a “Transformada de Fourier” poderia melhorar o seu dia-a-dia, mesmo sabendo que a transformada já é utilizada em vários domínios do conhecimento e com entusiasmo [Wikipedia, 2024b].

A inteligência artificial da computação está mais relacionada com a capacidade de realizar coisas que seres inteligentes (tais como um gato, um bebê, uma abelha ou um humano) realizam, como, por exemplo, puxar a mão (ou pata) instantaneamente ao tocar em uma superfície quente, realizar uma prova objetiva de anatomia ou elaborar um recurso para a anulação de uma questão de concurso. Se um programa realiza uma ação geralmente realizada por uma entidade dotada de inteligência, ele pode ser encarado como um programa que

simula uma inteligência artificial. Convenhamos que praticamente qualquer coisa cabe neste conceito.

Sobre este tema, o livro de Russell e Norvig (um dos livros mais lidos em todas as universidades do mundo) tem uma boa definição sobre o tema: “O campo da inteligência artificial [...] tenta não apenas compreender, mas também construir entidades inteligentes” (tradução nossa) [Russel and Norving, 2013]. Em outras palavras, a inteligência artificial da ciência da computação tem o audacioso objetivo de construir agentes dotados de inteligência.

A origem do termo “inteligência artificial”, na ciência da computação, é geralmente atribuída a John McCarthy, professor de Matemática da Universidade Dartmouth College [blipblog, 2024] (Figura 3.1). Ele organizou uma conferência com duração de oito semanas com outros colegas em 1956, alguns anos após a Segunda Guerra, e desde então o termo vem sendo utilizado para designar parte de conteúdos estudados em ciência da computação.

Um pouco antes, o artigo seminal de Alan Turing, com quem John McCarthy trabalhou em conjunto, já apresentava reflexões sobre a inteligência que uma máquina poderia possuir [Turing, 1950]. No entanto, a inteligência artificial aparece na literatura há milhares de anos; um exemplo é o Gigante Talos de Creta, um autômato proveniente da mitologia grega [Pickover, 2021].

(a) Jhon MacCarthy.



(b) Alan Turing.

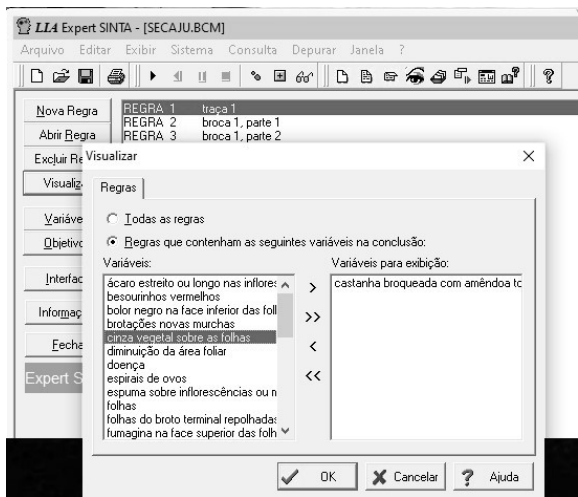


Figura 3.1: Jhon Maccarthy e Alan Turing. Imagens da internet.

Foi na década de 1970 que o uso da inteligência artificial começou a ser mais difundido. Uma das primeiras abordagens com relativo sucesso foi os Sistemas Especialistas (SE). Eles dependiam dos especialistas do domínio para transformar o conhecimento tácito (baseado em sua experiência) em explícito

(formalizado, documentado), que era então codificado na forma de regras em lógica formal. O processo de aquisição desse conhecimento acabou sendo um grande obstáculo na adoção em massa dessa abordagem. Veja um exemplo de software que implementa um motor de inferência baseado na teoria dos SE na Figura 3.2.

Figura 3.2: Interface de um Sistema Especialista ExpertSinta.



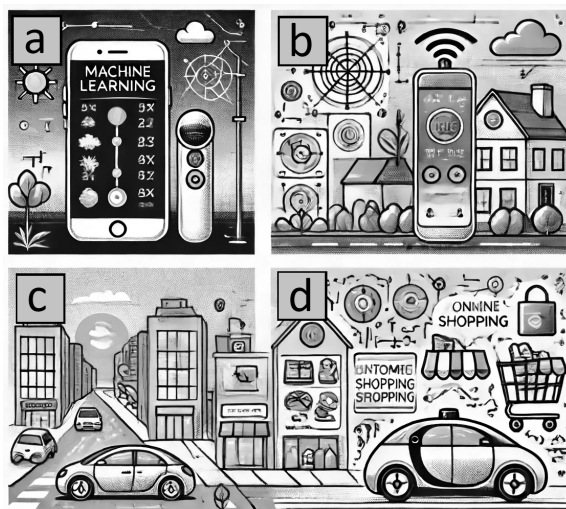
A superação de algumas limitações (tais como o aumento da capacidade de processamento e armazenamento dos computadores, a geração de grandes volumes de dados, novidades científicas e tecnológicas, chips supercondutores e a eficiência energética) permitiu o avanço de outras técnicas. Uma das técnicas que têm ganho notoriedade, por causa desses avanços, é o Aprendizado de Máquina.

3.2 Aprendizado de Máquina

O Aprendizado de Máquina (AM) é uma subárea da IA motivada pelo desenvolvimento de softwares mais independentes da intervenção humana para extração do conhecimento, o que era uma dificuldade nos Sistemas Especialistas. Geralmente, aplicações de AM utilizam indução para buscar modelos capazes de representar o conhecimento existente nos dados.

Na Figura 3.3, é possível identificar alguns usos de AM integrado em algumas atividades cotidianas. São elas: (a) um smartphone com um assistente de voz fornecendo atualizações meteorológicas; (b) um sistema de casa inteligente ajustando o termostato com base nas preferências do usuário; (c) um carro autônomo dirigindo em uma rua movimentada da cidade; (d) uma plataforma de compras online recomendando produtos a um usuário com base em suas compras anteriores. Essa figura foi criada inclusive com uma inteligência artificial chamada DALL-E 3, disponível no ChatGPT. ChatGPT é um chatbot que ganhou notoriedade, sendo um dos aplicativos que mais ganhou usuários rapidamente no mundo.

Figura 3.3: Exemplos AM.



As tarefas de aprendizado de máquina podem ser divididas entre tarefas **preditivas** e **descritivas**. As tarefas de aprendizado preditivas visam inferir o atributo alvo de uma nova entrada a partir da exposição prévia aos dados durante o treinamento do modelo. As tarefas descritivas buscam extrair padrões e correlações; além disso, não existe esta distinção entre atributos alvo e preditivos.

Ambas as tarefas podem ser categorizadas sob o conceito de aprendizado indutivo, sendo a capacidade de generalizar a partir de exemplos específicos, isto é, do conjunto de dados de treinamento. Em se tratando de tarefas preditivas, os

Figura 3.4: Classificação de AM.



algoritmos poderão implementar tarefas de **classificação**, nas quais o atributo alvo é **qualitativo discreto** (ou categórico), ou de **regressão**, em que o atributo alvo é **quantitativo contínuo** (ou numérico). Já as tarefas descritivas podem ser: *agrupamento*, que busca por similaridades, *associação*, que busca por padrões frequentes, e *sumarização*, que resulta em um resumo do conjunto de dados. No entanto, outras técnicas de aprendizagem de máquina supervisionadas e não supervisionadas estão fora do escopo deste livro.

3.3 Processamento de Linguagem Natural

O Processamento de Linguagem Natural (PLN) é um campo ligado à inteligência artificial, dedicando-se a equipar computadores com a capacidade de analisar e compreender a linguagem humana. Ele emprega técnicas computacionais com o propósito de aprender, compreender e produzir conteúdo em linguagem humana [Zhao et al., 2020]. Os sistemas de PLN podem suportar diferentes níveis ou combinações de níveis de análise linguística [Zhao et al., 2020]. Os níveis de análise linguística referem-se à análise fonética, morfológica, léxica, sintática, semântica, de discurso e análise pragmática da linguagem; existe uma suposição de que os seres humanos normalmente utilizam todos esses níveis para produzir ou compreender a linguagem [Zhao et al., 2020].

As abordagens de PLN podem ser classificadas em dois grandes grupos: o PLN simbólico e o PLN estatístico [Zhao et al., 2020]. Embora ambos os tipos de PLN tenham sido investigados ao mesmo tempo, foi o PLN simbólico que dominou o campo por algum tempo. Porém, abordagens estatísticas ganharam força principalmente após a divulgação do ChatGPT [OpenAI, 2023].

Na construção de chatbots, sua função principal reside em processar a entrada bruta do usuário, realizando a limpeza e a preparação dos dados textuais para que o sistema possa interpretar a mensagem e tomar as ações subsequentes apropriadas. Geralmente, o processo envolve a decomposição da linguagem em unidades menores, a compreensão do seu significado intrínseco e a determinação da resposta ou ação mais adequada.

3.4 Instalação do Python

Nesta seção, abordaremos como preparar o ambiente necessário para trabalhar com vetorização de texto em Python. Isso inclui a instalação do Python e das bibliotecas necessárias, além de uma breve introdução ao uso do Jupyter Notebook.

NOTA SOBRE REPRODUÇÃO DOS CÓDIGOS

Todos os exemplos de código deste livro foram testados em Python na versão 3.12 na IDE Visual Studio Code em um PC com GPU. Alguns poucos códigos quando especificados foram executados no google Colab. No quadro abaixo do código-fonte é apresentado a saída do console.

Para começar a trabalhar com vetorização de texto, é essencial ter o Python instalado. Python é uma linguagem de programação amplamente utilizada para análise de dados e aprendizado de máquina devido à sua simplicidade e à vasta gama de bibliotecas disponíveis.

Para instalar o Python, siga as instruções abaixo:

- No Windows, baixe o instalador do site oficial do Python (<https://www.python.org/>) e siga as instruções do instalador.
- No macOS, você pode usar o Homebrew para instalar o Python executando o comando `brew install python`.

- No Linux, o Python geralmente já está instalado, mas você pode atualizá-lo usando o gerenciador de pacotes da sua distribuição.

Uma vez que o Python esteja instalado, precisamos instalar algumas bibliotecas que são fundamentais para a vetorização de texto. Entre as principais estão “NumPy”, “Pandas”, “Scikit-learn”, “NLTK” e “SpaCy” e “Gensim”.

O “SpaCy” é uma biblioteca de PLN de código aberto em Python, conhecida por sua velocidade e eficiência. O spaCy oferece APIs intuitivas e modelos pré-treinados para diversas tarefas de PLN, incluindo tokenização, POS tagging, lematização, NER e análise de dependências. Sua arquitetura é focada em desempenho para aplicações em produção.

O “NLTK” (Natural Language Toolkit) é uma biblioteca Python fundamental para PLN, oferecendo uma ampla gama de ferramentas e recursos para tarefas como tokenização, stemming, POS tagging, análise sintática e NER. O NLTK é frequentemente utilizado para fins educacionais e de pesquisa.

O “Gensim” é uma biblioteca Python especializada em modelagem de tópicos, análise de similaridade semântica e vetores de palavras. Ele é particularmente útil para identificar estruturas semânticas em grandes coleções de texto.

O “pip” é o gerenciador de pacotes padrão do Python. Você pode instalar as bibliotecas necessárias usando o seguinte comando:

Código-fonte 3.1: Comando do terminal para instalar algumas das bibliotecas necessárias do Python via pip.

```
1 pip install numpy pandas scikit-learn nltk spacy gensim
```

Após instalar as bibliotecas, é importante verificar se elas foram instaladas corretamente:

Código-fonte 3.2: Exibe as versões das bibliotecas instaladas.

```
1 import numpy as np
2 import pandas as pd
3 import sklearn
4 import nltk
5 import spacy
6 import gensim
7
8 print("NumPy version:", np.__version__)
```



```
9 print("Pandas version:", pd.__version__)
10 print("Scikit-learn version:", sklearn.__version__)
11 print("NLTK version:", nltk.__version__)
12 print("SpaCy version:", spacy.__version__)
13 print("Gensim:", gensim.__version__)
```

```
1 NumPy version: 1.26.4
2 Pandas version: 2.2.2
3 Scikit-learn version: 1.5.1
4 NLTK version: 3.9.1
5 SpaCy version: 3.7.6
```

Este código importará as bibliotecas e exibirá suas versões, garantindo que todas estejam corretamente instaladas.

O Jupyter Notebook é uma ferramenta poderosa para o desenvolvimento de scripts em Python, permitindo a combinação de código, texto, visualizações e resultados em um único documento.

Você pode instalar o Jupyter Notebook usando o pip:

Código-fonte 3.3: Comando do instalador de dependências do python para instalar a biblioteca para usar o notebook.

```
1 pip install jupyterlab
```

Para iniciar o Jupyter Notebook, execute o seguinte comando no terminal:

Código-fonte 3.4: Comando no terminal depois de tudo instalado para iniciar o notebook.

```
1 jupyter notebook
```

Isso abrirá o Jupyter Notebook no seu navegador padrão, permitindo que você comece a escrever e executar código Python de maneira interativa.

Um exemplo simples de uso do Jupyter Notebook seria a criação de uma célula de código para calcular a soma de dois números:

Código-fonte 3.5: Código Python para somar dois números.

```
1 a = 10
2 b = 20
3 print("A soma de a e b é:", a + b)
```

```
1 A soma de a e b é: 30
```

Este exemplo demonstra a simplicidade e interatividade que o Jupyter Notebook oferece, permitindo que você execute código Python célula por célula e veja os resultados imediatamente. Em resumo, configuramos o ambiente necessário para trabalhar com Python e as principais bibliotecas instaladas, além da configuração do Jupyter Notebook.

3.5 Principais Técnicas de PLN

3.5.1 Tokenização

Tokenização é o processo de dividir um texto em unidades menores chamadas “tokens” que podem ser palavras, frases ou até mesmo sentenças [Jurafsky and Martin, 2023]. A tokenização é o primeiro passo em muitos algoritmos de PLN, pois permite que os dados textuais sejam manipulados de forma programática.

Tokenizar não é só separar por espaços, mas também lidar com pontuações, contrações e outros aspectos que podem afetar a análise. Um exemplo simples seria a frase “Eu estou feliz.”, que seria tokenizada em [“Eu”, “estou”, “feliz”, “.”]. Não necessariamente uma palavra equivale a um token. Em alguns casos, como em palavras compostas ou expressões idiomáticas, um único token pode representar uma ideia ou conceito mais amplo. Por exemplo, “São Paulo” poderia ser considerado um único token em vez de dois (“São” e “Paulo”).

Existem diferentes abordagens para tokenização, incluindo tokenização baseada em regras, onde padrões específicos são definidos para identificar tokens (geralmente utilizando expressões regulares), e tokenização baseada em aprendizado de máquina, onde algoritmos aprendem a segmentar o texto com base em exemplos anteriores.

A tokenização pode ser feita de várias maneiras, dependendo do idioma e do objetivo da análise. Em inglês, por exemplo, a tokenização pode ser mais simples devido à estrutura gramatical, enquanto em idiomas como o chinês, onde não há espaços entre as palavras, a tokenização pode ser mais complexa.

A seguir, um exemplo do uso da biblioteca nltk e a tokenização de determinado texto, também chamado de corpus.

Código-fonte 3.6: Este código demonstra como separar um texto em suas palavras e sentenças usando NLTK.

```

1 # pip install nltk
2 import nltk
3 from nltk.tokenize import word_tokenize, sent_tokenize
4
5 # Exemplo de texto
6 texto = "Chatbots estão se tornando cada vez mais populares. Eles podem
7         realizar muitas tarefas automaticamente."
8
9 # Tokenização em palavras
10 tokens_palavras = word_tokenize(texto)
11 print("Tokens de palavras:", tokens_palavras)
12
13 # Tokenização em sentenças
14 tokens_sentencas = sent_tokenize(texto)
15 print("Tokens de sentenças:", tokens_sentencas)

```

```

1 Tokens de palavras: ['Chatbots', 'estão', 'se', 'tornando', 'cada',
2                     'vez', 'mais', 'populares', '.', 'Eles', 'podem', 'realizar',
3                     'muitas', 'tarefas', 'automaticamente', '.']
4
5 Tokens de sentenças: ['Chatbots estão se tornando cada vez mais
6                       populares.', 'Eles podem realizar muitas tarefas automaticamente.']

```

SpaCy é uma biblioteca para PLN que oferece uma interface fácil de usar e é otimizada para processamento rápido. É possível também utilizar a biblioteca “spacy”, conforme os códigos a seguir:

Código-fonte 3.7: Utiliza a biblioteca SpaCy para processamento de linguagem natural em português.

```

1 # pip install spacy
2 # import os
3 # os.system("python -m spacy download pt_core_news_sm")
4 import spacy
5
6 # Carregar o modelo de linguagem em português
7 nlp = spacy.load("pt_core_news_sm")
8
9 # Exemplo de texto
10 texto = "Chatbots estão se tornando cada vez mais populares."
11
12 # Processando o texto
13 doc = nlp(texto)

```

```
14 |
15 | # Tokenização
16 | tokens = [token.text for token in doc]
17 | print("Tokens:", tokens)

1 | Tokens: ['Chatbots', 'estão', 'se', 'tornando', 'cada', 'vez', 'mais',
  |         'populares', '.']
```

3.5.2 Lematização

Lematização é o processo de reduzir palavras flexionadas ao seu lema, ou forma base [Singh et al., 2019]. Diferente da stemização, a lematização leva em consideração o contexto e a gramática da palavra para obter a forma correta.

Código-fonte 3.8: Utiliza a biblioteca NLTK para realizar lematização de palavras em português. Ele define uma lista de palavras e aplica o lematizador a cada uma delas.

```
1 | # pip install nltk
2 | # import nltk
3 | # nltk.download('wordnet')
4 |
5 | import nltk
6 | from nltk.stem import WordNetLemmatizer
7 |
8 | # Inicializando o lematizador
9 | lemmatizer = WordNetLemmatizer()
10 |
11 | # Exemplo de palavras
12 | palavras = ["correndo", "correu", "corredores"]
13 |
14 | # Lematização das palavras
15 | lematizadas = [lemmatizer.lemmatize(palavra, pos='v') for palavra in
16 |               palavras]
17 | print("Palavras lematizadas:", lematizadas)

1 | Palavras lematizadas: ['correndo', 'correu', 'corredores']
```

Com spacy

Código-fonte 3.9: Mesmo exemplo do código anterior porém utilizando a biblioteca Spacy.

```
1 #pip install spacy
2 import spacy
3
4 # Carregar o modelo de linguagem em português
5 nlp = spacy.load("pt_core_news_sm")
6
7 # Exemplo de texto
8 texto = "Chatbots estão se tornando cada vez mais populares."
9
10 # Processando o texto
11 doc = nlp(texto)
12
13 tokens = [token.text for token in doc]
14 print("Tokens:", tokens)
15
16 lematizadas = [token.lemma_ for token in doc]
17 print("Palavras lematizadas:", lematizadas)
```

```
1 Tokens: ['Chatbots', 'estão', 'se', 'tornando', 'cada', 'vez', 'mais',
          'populares', '.']
2 Palavras lematizadas: ['chatbots', 'estar', 'se', 'tornar', 'cada',
          'vez', 'mais', 'popular', '.']
```

3.5.3 Stemização

A stemização é o processo de reduzir as palavras às suas raízes ou “stems”. É uma técnica mais simples que a lematização e, geralmente, não considera o contexto, o que pode levar a resultados menos precisos.

Código-fonte 3.10: Utiliza a biblioteca NLTK para realizar a stemizacao de palavras. Ele define uma lista de palavras e aplica o stemmer para obter os radicais dessas palavras

```
1 # pip install nltk
2 from nltk.stem import PorterStemmer
3
4 # Inicializando o stemizador
5 stemmer = PorterStemmer()
6
7 # Exemplo de palavras
8 palavras = ["correndo", "correu", "corredores"]
9
10 # Stemização das palavras
```

```
11 stems = [stemmer.stem(palavra) for palavra in palavras]
12 print("Stems das palavras:", stems)
```

```
1 Stems das palavras: ['correndo', 'correu', 'corredor']
```

3.5.4 Stopwords

Stopwords são palavras comuns em um idioma (como “o”, “a”, “e”, “de”, “que”, etc.) que geralmente são removidas durante o pré-processamento de texto, pois não contribuem significativamente para o significado do texto [Raj, 2019]. A remoção dessas palavras pode melhorar a eficácia de certos algoritmos de PLN, focando nas palavras mais informativas do texto.

Código-fonte 3.11: Demonstra como extrair o radical de palavras usando NLTK.

```
1 # pip install nltk
2 # import nltk
3 # nltk.download('stopwords')
4
5 from nltk import word_tokenize
6 from nltk.corpus import stopwords
7
8 # Carregar stopwords para o idioma português
9 stop_words = set(stopwords.words('portuguese'))
10
11 # Exemplo de texto
12 texto = "Chatbots estão se tornando cada vez mais populares."
13
14 # Removendo stopwords
15 tokens_sem_stopwords = [palavra for palavra in word_tokenize(texto) if
16                          palavra.lower() not in stop_words]
17 print("Texto sem stopwords:", tokens_sem_stopwords)
```

```
1 Texto sem stopwords: ['Chatbots', 'tornando', 'cada', 'vez', 'populares',
2                       '.']
```

Com a biblioteca Spacy:

Código-fonte 3.12: Demonstra como tokenizar um texto e filtrar stopwords usando SpaCy em português.

```
1 # pip install spacy
2 # python -m spacy download pt_core_news_sm
3 import spacy
4
5 # Carregar o modelo de linguagem em português
6 nlp = spacy.load("pt_core_news_sm")
7
8 # Exemplo de texto
9 texto = "Chatbots estão se tornando cada vez mais populares."
10
11 # Processando o texto
12 doc = nlp(texto)
13
14 tokens = [token.text for token in doc]
15 print("Tokens:", tokens)
16
17 # Removendo stopwords
18 tokens_sem_stopwords = [token.text for token in doc if not token.is_stop]
19 print("Texto sem stopwords:", tokens_sem_stopwords)
```

```
1 Tokens: ['Chatbots', 'estão', 'se', 'tornando', 'cada', 'vez', 'mais',
          'populares', '.']
2 Texto sem stopwords: ['Chatbots', 'tornando', 'populares', '.']
```

3.6 Outras técnicas de PLN

Marcação Morfossintática (POS Tagging): Esta técnica consiste em atribuir a cada *token* em um texto uma categoria gramatical, como substantivo, verbo, adjetivo, advérbio, etc. A marcação POS é utilizada para identificar entidades e compreender a estrutura gramatical das frases. Por exemplo, na frase “Eu estou aprendendo como construir chatbots”, a marcação POS poderia identificar “Eu” como um pronome (PRON), “estou aprendendo” como um verbo (VERB) e “chatbots” como um substantivo (NOUN).

Reconhecimento de Entidades Nomeadas (NER): O NER é a tarefa de identificar e classificar entidades nomeadas em um texto, como nomes de pessoas (PERSON), organizações (ORG), localizações geográficas (GPE, LOC), datas (DATE), valores monetários (MONEY), etc. Por exemplo, na frase “Google tem sua sede em Mountain View, Califórnia, com uma receita de 109,65 bilhões de

dólares americanos”, o NER identificaria “Google” como uma organização (ORG), “Mountain View” e “Califórnia” como localizações geográficas (GPE) e “109,65 bilhões de dólares americanos” como um valor monetário (MONEY). Essa capacidade é vital para que chatbots compreendam os detalhes relevantes nas *utterances* dos usuários.

Análise de Dependências (Dependency Parsing): Esta técnica examina as relações gramaticais entre as palavras em uma frase, revelando a estrutura sintática e as dependências entre os *tokens*. A análise de dependências pode ajudar a entender quem está fazendo o quê a quem. Por exemplo, na frase “Reserve um voo de Corumbá para Maceió”, a análise de dependências pode identificar “Corumbá” e “Maceió” como modificadores de “voo” através das preposições “de” e “para”, respectivamente, e “Reserve” como a raiz da ação. Essa análise é útil para extrair informações sobre as intenções do usuário, mesmo em frases mais complexas.

Classificação de Texto: Uma técnica de aprendizado de máquina que atribui um texto a uma ou mais categorias predefinidas. No contexto de chatbots, a classificação de texto é fundamental para a detecção de intenção (mais sobre detecção de *intenção* será apresentada nas seções seguintes), onde as categorias representam as diferentes intenções do usuário. Algoritmos como o *Naïve Bayes* são modelos estatísticos populares para essa tarefa, baseados no teorema de Bayes e em fortes suposições de independência entre as características. O treinamento desses classificadores requer um *corpus* de dados rotulados, onde cada *utterance* (entrada do usuário) é associada a uma intenção específica.

3.7 Expressões Regulares

Expressões regulares, frequentemente abreviadas como *regex*, são sequências de caracteres que definem padrões de busca. Elas são utilizadas em chatbots para diversas tarefas relacionadas ao processamento e à análise de texto fornecido pelos usuários.

Frameworks populares para desenvolvimento de chatbots frequentemente integram o uso de expressões regulares para aprimorar a extração de entidades. Por exemplo, as *regex* podem ser definidas nos dados de treinamento para ajudar o sistema a reconhecer padrões específicos como nomes de ruas ou códigos

de produtos. Essa abordagem permite melhorar a precisão do reconhecimento de entidades, um componente importante para a compreensão da intenção do usuário.

A sintaxe das expressões regulares consiste em uma combinação de caracteres literais (que correspondem a si mesmos) e metacaracteres, que possuem significados especiais e permitem definir padrões de busca mais complexos. As expressões regulares podem ser aplicadas em uma variedade de cenários no desenvolvimento de chatbots. Algumas das aplicações de regex nos chatbots incluem:

- Extração de entidades: Identificação e extração de informações específicas, como endereços de e-mail, números de telefone, datas e outros dados estruturados presentes na entrada do usuário.
- Validação de entradas do usuário: Verificação se a entrada do usuário corresponde a um formato esperado, como datas em um formato específico (DD/MM/AAAA), códigos postais ou outros padrões predefinidos.
- Detecção de Intenção: Detecção de comandos específicos inseridos pelo usuário, como /ajuda, /iniciar ou palavras-chave que indicam uma intenção específica.
- Limpeza de texto: Remoção de ruídos e elementos indesejados do texto, como tags HTML, espaços em branco excessivos ou caracteres especiais que podem interferir no processamento subsequente.
- Tokenização simples: Embora métodos mais avançados sejam comuns em PLN, regex pode ser usada para dividir o texto em unidades menores (tokens) com base em padrões simples.

Essas tarefas são fundamentais para garantir que o chatbot possa interpretar e responder adequadamente às entradas dos usuários, especialmente em cenários onde a informação precisa ser estruturada ou verificada antes de ser processada por modelos de linguagem mais complexos. Cada uma destas tarefas será detalhada neste Capítulo.

O módulo `re` em Python é a biblioteca padrão para trabalhar com expressões regulares. Ele fornece diversas funções que permitem realizar operações de busca, correspondência e substituição em strings com base em padrões definidos por regex. Algumas das funções mais utilizadas incluem:

- `re.match(pattern, string)`: Tenta encontrar uma correspondência do padrão no *início* da string. Se uma correspondência for encontrada, retorna um objeto de correspondência; caso contrário, retorna `None`.
- `re.search(pattern, string)`: Procura a primeira ocorrência do padrão em *qualquer posição* da string. Retorna um objeto de correspondência se encontrado, ou `None` caso contrário.
- `re.findall(pattern, string)`: Encontra *todas* as ocorrências não sobrepostas do padrão na string e as retorna como uma lista de strings.
- `re.sub(pattern, repl, string)`: Substitui todas as ocorrências do padrão na string pela string de substituição `repl`. Retorna a nova string resultante.

Extração de E-mails

Um caso de uso comum em chatbots é a extração de endereços de e-mail do texto fornecido pelo usuário. O seguinte exemplo em Python demonstra como usar `re.findall` para realizar essa tarefa. Ilustrando o uso regex para identificar e extrair informações específicas de um texto.

Código-fonte 3.13: Extração de e-mails com regex.

```
1 import re
2 texto = "Entre em contato em exemplo@email.com ou suporte@outroemail.com."
3 padrao = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'
4 emails = re.findall(padrao, texto)
5 print(emails)
```

```
1 ['exemplo@email.com', 'suporte@outroemail.com']
```

Validação de Datas

Chatbots que lidam com agendamentos ou reservas frequentemente precisam validar se a data fornecida pelo usuário está em um formato correto. O seguinte exemplo demonstra como validar datas no formato DD/MM/AAAA:

Código-fonte 3.14: Validação de datas com regex.

```
1 import re
```

```
2
```

```
3 padrao_data = r'\b\d{2}/\d{2}/\d{4}\b'
4 datas_teste = ["31/12/2020", "1/1/2021", "2023-05-10", "25/06/2025 10:00"]
5
6 for data in datas_teste:
7     if re.match(padrao_data, data):
8         print(f"'{data}' é uma data válida no formato DD/MM/AAAA.")
9     else:
10        print(f"'{data}' não é uma data válida no formato DD/MM/AAAA.")
```

A saída deste código ilustra quais das strings de teste correspondem ao padrão de data especificado.

```
1 '31/12/2020' é uma data válida no formato DD/MM/AAAA.
2 '1/1/2021' não é uma data válida no formato DD/MM/AAAA.
3 '2023-05-10' não é uma data válida no formato DD/MM/AAAA.
4 '25/06/2025 10:00' é uma data válida no formato DD/MM/AAAA.
```

Análise de Comandos

Em interfaces de chatbot baseadas em texto, os usuários podem interagir através de comandos específicos, como /ajuda ou /iniciar. As regex podem ser usadas para detectar esses comandos de forma eficiente. Este exemplo abaixo mostra como identificar strings que começam com uma barra seguida por um ou mais caracteres alfanuméricos.

Código-fonte 3.15: Análise de comandos com regex.

```
1 import re
2
3 padrao_comando = r'^/\w+'
4 comandos_teste = ["/ajuda", "/iniciar", "ajuda", "iniciar/"]
5
6 for comando in comandos_teste:
7     if re.match(padrao_comando, comando):
8         print(f"'{comando}' é um comando válido.")
9     else:
10        print(f"'{comando}' não é um comando válido.")
```

```
1 '/ajuda' é um comando válido.
2 '/iniciar' é um comando válido.
3 'ajuda' não é um comando válido.
4 'iniciar/' não é um comando válido.
```

Tokenização Simples

Embora para tarefas complexas de PLN sejam utilizadas técnicas de tokenização mais avançadas, as regex podem ser úteis para realizar uma tokenização básica, dividindo o texto em palavras ou unidades menores com base em padrões de separação. A saída será uma lista de strings, onde o padrão `\W+` corresponde a um ou mais caracteres não alfanuméricos, utilizados como delimitadores.

Código-fonte 3.16: Tokenização simples com regex.

```
1 import re
2 texto = "Olá, como vai você?"
3 tokens = re.split(r'\W+', texto)
4 print(tokens)
```

```
1 ['Olá', 'como', 'vai', 'você', '']
```

Limpeza de Texto

Chatbots podem precisar processar texto que contém elementos indesejados, como tags HTML. As regex podem ser usadas para remover esses elementos:

Código-fonte 3.17: Limpeza de texto removendo tags HTML.

```
1 import re
2 texto_html = "<p>Este é um parágrafo com <b>texto em negrito</b>.</p>"
3 texto_limpo = re.sub(r'<[>]+>', '', texto_html)
4 print(texto_limpo)
```

```
1 Este é um parágrafo com texto em negrito.
```

Embora os fundamentos das regex sejam suficientes para muitas tarefas, existem construções mais avançadas que podem ser úteis em cenários complexos. Alguns exemplos incluem Lookaheads e Lookbehinds que permitem verificar se um padrão é seguido ou precedido por outro padrão, sem incluir esse outro padrão na correspondência; além disso, correspondência não-gulosa: que ao usar quantificadores como `*` ou `+`, a correspondência padrão é "gulosa", ou seja, tenta corresponder à maior string possível. Adicionar um `?` após o quantificador (`*?`, `+`?) torna a correspondência "não-gulosa", correspondendo à menor string possível.

É importante reconhecer que, apesar de sua utilidade, as expressões regulares têm limitações significativas quando se trata de compreender a complexidade da linguagem natural. As regex são baseadas em padrões estáticos e não possuem a capacidade de entender o contexto, a semântica ou as nuances da linguagem humana.

No contexto de um fluxo de trabalho de chatbot, as expressões regulares são frequentemente mais eficazes nas etapas de pré-processamento, como limpeza e validação de entradas, enquanto técnicas de PLN mais sofisticadas são empregadas para a compreensão da linguagem em um nível mais alto.

Porém, para tarefas que exigem uma compreensão mais profunda do significado e da intenção por trás das palavras, técnicas avançadas de Processamento de Linguagem Natural (PLN), como modelagem de linguagem, análise de sentimentos e reconhecimento de entidades nomeadas (NER) baseadas em aprendizado de máquina, são indispensáveis e serão apresentadas no Capítulo 4.

3.8 Entendimento de Linguagem Natural

O Entendimento de Linguagem Natural (ULN) é um subdomínio específico dentro do universo mais vasto do PLN. Enquanto o PLN abarca um conjunto diversificado de operações sobre a linguagem, o ULN se concentra de maneira particular na habilidade da máquina de aprender e interpretar a linguagem natural tal como ela é comunicada pelos seres humanos. Em outras palavras, o ULN é o ramo do PLN dedicado à extração de significado e à identificação da intenção por trás do texto inserido pelo usuário. As aplicações do ULN são extensas e incluem funcionalidades cruciais para chatbots, como a capacidade de responder a perguntas, realizar buscas em linguagem natural, identificar relações entre entidades, analisar o sentimento expresso no texto, sumarizar informações textuais e auxiliar em processos de descoberta legal.

No cerne da funcionalidade de um chatbot reside a sua capacidade de compreender as mensagens dos usuários e responder de forma adequada. O PLN desempenha um papel central nesse processo, permitindo que o chatbot: (i) detecte a intenção do usuário; (ii) extraia entidades relevantes; e (iii) processe linguagem variada e informal; e (iv) mantenha o contexto da conversa.

(i) Detecção da Intenção do Usuário: O objetivo por trás da mensagem do usuário é o primeiro passo. Isso é frequentemente abordado como um problema de classificação de texto, onde o chatbot tenta classificar a *utterance* do usuário em uma das intenções predefinidas. Para detecção da intenção, é possível utilizar técnicas de aprendizado de máquina, como o algoritmo *Naïve Bayes*, para construir esses classificadores. Plataformas como Rasa NLU (<https://rasa.com/>), LUIS.ai (<https://www.luis.ai/>) e Dialogflow (<https://dialogflow.cloud.google.com/>) simplificam significativamente o processo de treinamento e implantação desses modelos de intenção.

O Rasa NLU é um componente de código aberto do framework Rasa para construir chatbots, focado em entendimento de linguagem natural. Ele permite treinar modelos personalizados para classificação de intenção e extração de entidades, oferecendo flexibilidade e controle sobre os dados.

(ii) Extraia Entidades Relevantes: Além da intenção geral, as mensagens dos usuários frequentemente contêm detalhes específicos, conhecidos como entidades, que são essenciais para atender à solicitação. Por exemplo, em “Reserve um voo de Corumbá para Maceió amanhã”, a intenção é reservar um voo, e as entidades são a cidade de origem (“Corumbá”), a cidade de destino (“Maceió”) e a data (“amanhã”). As técnicas de NER e os modelos de extração de entidades fornecidos por ferramentas como spaCy, NLTK, CoreNLP, LUIS.ai e Rasa NLU são fundamentais para identificar e extrair essas informações.

O CoreNLP é um conjunto de ferramentas de PLN robusto e amplamente utilizado, desenvolvido em Java. O CoreNLP oferece capacidades abrangentes de análise linguística, incluindo POS tagging, análise de dependências, NER e análise de sentimentos. Possui APIs para integração com diversas linguagens de programação, incluindo Python.

(iii) Processe Linguagem Variada e Informal: Os usuários podem se comunicar com chatbots usando uma ampla gama de vocabulário, gramática e estilo, incluindo erros de digitação, abreviações e linguagem informal. As técnicas de PLN, como stemming, lematização e busca por similaridade, ajudam o chatbot a lidar com essa variabilidade e a compreender a essência da mensagem, mesmo que não seja expressa de forma perfeitamente gramatical.

A arquitetura típica de um chatbot envolve uma camada de processamento de linguagem natural (NLP/NLU engine) que recebe a entrada de texto do usuário. Essa camada é responsável por realizar as tarefas de PLN mencionadas

anteriormente: tokenização, análise morfossintática, extração de entidades, detecção de intenção, etc. O resultado desse processamento é uma representação estruturada da mensagem do usuário, que pode ser entendida pela lógica de negócios do chatbot.

Com base nessa representação estruturada, um motor de decisão (*decision engine*) no chatbot pode então corresponder à intenção do usuário a fluxos de trabalho preconfigurados ou a regras de negócio específicas. Em alguns casos, a geração de linguagem natural (NLG), outro subcampo do PLN, é utilizada para formular a resposta do chatbot ao usuário.

3.8.1 Intents

Os Intents representam a intenção ou o propósito por trás da mensagem de um usuário ao interagir com o chatbot [Khan and Das, 2018]. Em termos mais simples, é o que o usuário deseja que o chatbot faça ou sobre o que ele quer saber.

Um intent é usado para identificar programaticamente a intenção da pessoa que está usando o chatbot. O chatbot deve ser capaz de executar alguma ação com base no “intent” que detecta na mensagem do usuário. Cada tarefa que o chatbot deve realizar define um intent. A aplicação prática dos intents varia conforme o domínio do chatbot, veja o exemplo:

Por exemplo, para um chatbot de uma loja de moda, exemplos de intents seriam “busca de um produto” (quando um usuário quer ver produtos) e “endereço loja” (quando um usuário pergunta sobre lojas); em um chatbot para pedir comida, “consultar preços” e “realizar pedido” podem ser intents distintos.

Detectar o *intent* da mensagem do usuário é um problema conhecido de aprendizado de máquina, realizado por meio de uma técnica chamada classificação de texto. O objetivo é classificar frases em múltiplas classes (os intents). O modelo de aprendizado de máquina é treinado com um conjunto de dados que contém exemplos de mensagens e seus intents correspondentes. Após o treinamento, o modelo pode prever o intent de novas mensagens que não foram vistas antes.

3.8.2 Utterances

Cada intent pode ser expresso de várias maneiras pelo usuário. Essas diferentes formas são chamadas de *utterances* ou expressões do usuário.

Por exemplo, para o intent `realizar pedido`, as utterances poderiam ser “Eu gostaria de fazer um pedido”, “Quero pedir comida”, “Como faço para pedir?”, etc. Cada uma dessas expressões representa a mesma intenção, mas com palavras diferentes. O modelo de aprendizado de máquina deve ser capaz de reconhecer todas essas variações como pertencentes ao mesmo intent.

É sugerido fornecer um número ótimo de utterances variadas por intent para garantir um bom treinamento do modelo de reconhecimento.

3.8.3 Entities

As entidades extraídas permitem ao chatbot refinar sua resposta ou ação. Os Intents frequentemente contêm metadados importantes chamados *Entities*. Estas são palavras-chave ou frases dentro da utterance do usuário que ajudam o chatbot a identificar detalhes específicos sobre o pedido, permitindo fornecer informações mais direcionadas. Por exemplo, na frase “Eu quero pedir uma pizza de calabreza com borda recheada”, as entidades podem incluir: Para o *intent realizar pedido* e para as *Entities: pizza, calabreza e borda recheada*.

3.8.4 Desafios

O processo de treinamento envolve a construção de um modelo de aprendizado de máquina. Este modelo aprende a partir do conjunto definido de intents, suas utterances associadas e as entidades anotadas. O objetivo do treinamento é capacitar o modelo a categorizar corretamente novas utterances (que não foram vistas durante o treinamento) no intent apropriado e a extrair as entidades relevantes.

Quando o chatbot processa uma nova mensagem do usuário, o modelo de reconhecimento de intent não apenas classifica a mensagem em um dos intents definidos, mas também fornece uma *pontuação de confiança* (geralmente entre 0 e 1). Essa pontuação indica o quão seguro o modelo está e que a classificação está correta. É comum definir um *limite (threshold)* de confiança. Se a pontuação

do intent detectado estiver abaixo desse limite, o chatbot pode pedir esclarecimentos ao usuário em vez de executar uma ação baseada em uma suposição incerta.

Uma vez que um intent é detectado com confiança suficiente, o chatbot pode executar a ação correspondente. Isso pode envolver consultar um banco de dados, chamar uma API externa, fornecer uma resposta estática ou iniciar um fluxo de diálogo mais complexo. Além disso, a análise dos intents mais frequentemente capturados fornece insights valiosos sobre como os usuários estão interagindo com o chatbot e quais são suas principais necessidades. Essas análises são importantes tanto para a otimização do bot quanto para as decisões de negócio.

Apesar destes procedimentos, alguns problemas ainda desafiam os pesquisadores, tais como a Geração de texto coerente; a Sintaxe e gramática; a semântica; o Contexto; e a Ambiguidade.

A *geração de texto coerente* é um desafio porque envolve não apenas a escolha de palavras, mas também a construção de frases que façam sentido no contexto da conversa. A *sintaxe e gramática* são importantes para garantir que o texto gerado seja gramaticalmente correto e compreensível. A *semântica* se refere ao significado das palavras e frases, e é importante para garantir que o texto gerado transmita a mensagem correta. O *contexto* é importante para entender o que foi dito anteriormente na conversa e como isso afeta a resposta atual. A *ambiguidade* pode surgir quando uma palavra ou frase tem múltiplos significados, tornando difícil para o modelo determinar qual interpretação é a correta.

3.9 Vetorização e Representação de Texto

A vetorização de texto é um passo importante no Processamento de Linguagem Natural (PLN), pois permite transformar dados textuais em uma forma que os modelos de aprendizado de máquina podem entender e processar. Nesta seção, exploraremos algumas técnicas de vetorização, incluindo One-Hot Encoding, Bag of Words e TF-IDF (Term Frequency-Inverse Document Frequency), juntamente com implementações práticas em Python.

3.9.1 One-Hot Encoding

One-Hot Encoding é uma das formas mais simples de vetorização de texto, onde cada palavra em um vocabulário é representada como um vetor binário. Cada posição no vetor corresponde a uma palavra do vocabulário, e apenas a posição da palavra que está sendo representada tem valor 1, enquanto todas as outras têm valor 0.

Código-fonte 3.18: Transforma palavras de um pequeno corpus de frases em vetores one-hot usando o OneHotEncoder do scikit-learn.

```
1 # pip install scikit-learn
2 from sklearn.preprocessing import OneHotEncoder
3 import numpy as np
4
5 # Exemplo de texto
6 corpus = ["Eu amo programação", "A programação é divertida"]
7
8 # Tokenização simples
9 tokenized_corpus = [sentence.split() for sentence in corpus]
10 # Flatten para obter todas as palavras
11 all_words = [word for sentence in tokenized_corpus for word in sentence]
12 # Remover duplicatas
13 vocab = list(set(all_words))
14
15 # Criar matriz de índices das palavras para cada frase
16 word_to_idx = {word: idx for idx, word in enumerate(vocab)}
17 corpus_idx = [[word_to_idx[word] for word in sentence] for sentence in
18               tokenized_corpus]
19
20 # Ajustar o encoder para o vocabulário
21 encoder = OneHotEncoder(sparse_output=False)
22 one_hot_encoded = encoder.fit_transform(
23     np.array(vocab).reshape(-1, 1))
24
25 print("Vocabulário:", vocab)
26 print("One-Hot Encoding:\n", one_hot_encoded)
```

```
1 Vocabulário: ['amo', 'é', 'programação', 'Eu', 'A', 'divertida']
2 One-Hot Encoding:
3 [[0. 0. 1. 0. 0. 0.]
4  [0. 0. 0. 0. 0. 1.]
5  [0. 0. 0. 0. 1. 0.]
6  [0. 1. 0. 0. 0. 0.]
```

```

7 [1. 0. 0. 0. 0. 0.]
8 [0. 0. 0. 1. 0. 0.]]

```

Embora seja fácil de entender e implementar, o One-Hot Encoding não leva em consideração a semântica das palavras, resultando em vetores muito esparsos e de alta dimensionalidade, especialmente para vocabulários grandes.

3.9.2 Bag of Words (BoW)

A técnica de Bag of Words (BoW) é uma melhoria em relação ao One-Hot Encoding. Aqui, em vez de vetores binários, usamos contagens de palavras. Para cada documento, construímos um vetor com a frequência de cada palavra no vocabulário.

Código-fonte 3.19: Utiliza a biblioteca scikit-learn para transformar um conjunto de frases (corpus) em uma matriz BoW.

```

1 # pip install scikit-learn
2 from sklearn.feature_extraction.text import CountVectorizer
3
4 # Exemplo de corpus
5 corpus = ["Eu amo programação", "A programação é divertida", "Eu amo a
6          vida"]
7
8 # Criação do vetor de contagem (Bag of Words)
9 vectorizer = CountVectorizer()
10 bow_matrix = vectorizer.fit_transform(corpus)
11
12 print("Vocabulário:", vectorizer.get_feature_names_out())
13 print("Bag of Words Matrix:\n", bow_matrix.toarray())

```

```

1 Vocabulário: ['amo' 'divertida' 'eu' 'programação' 'vida']
2 Bag of Words Matrix:
3 [[1 0 1 1 0]
4  [0 1 0 1 0]
5  [1 0 1 0 1]]

```

A matriz resultante tem cada linha representando um documento e cada coluna representando a contagem de uma palavra específica no documento. No entanto, essa técnica também não leva em consideração o contexto ou a ordem das palavras.

3.9.3 TF-IDF (Term Frequency-Inverse Document Frequency)

TF-IDF é uma técnica que combina a frequência de termos (TF) com a frequência inversa de documentos (IDF). Esta abordagem ajuda a dar mais peso a palavras que são raras no corpus, mas aparecem frequentemente em um documento específico, o que geralmente indica sua importância.

Código-fonte 3.20: Técnica comum em processamento de linguagem natural para converter texto em dados numéricos.

```
1 # pip install scikit-learn
2 from sklearn.feature_extraction.text import TfidfVectorizer
3
4 # Exemplo de corpus
5 corpus = ["Eu amo programação", "A programação é divertida", "Eu amo a
6          vida"]
7
8 # Criação do vetor TF-IDF
9 tfidf_vectorizer = TfidfVectorizer()
10 tfidf_matrix = tfidf_vectorizer.fit_transform(corpus)
11
12 print("Vocabulário:", tfidf_vectorizer.get_feature_names_out())
13 print("TF-IDF Matrix:\n", tfidf_matrix.toarray())
```

```
1 Vocabulário: ['amo' 'divertida' 'eu' 'programação' 'vida']
2 TF-IDF Matrix:
3 [[0.57735027 0.          0.57735027 0.57735027 0.          ]
4  [0.          0.79596054 0.          0.60534851 0.          ]
5  [0.51785612 0.          0.51785612 0.          0.68091856]]
```

A matriz TF-IDF dá valores diferentes para as palavras, não apenas com base em sua frequência, mas também na relevância, ajudando a reduzir o peso de palavras comuns que não são muito informativas para o contexto.

3.9.4 Comparação de Técnicas de Vetorização

As três técnicas abordadas têm seus prós e contras:

- One-Hot Encoding: Simples e fácil de implementar, mas resulta em vetores esparsos e de alta dimensionalidade.
- Bag of Words: Considera a frequência das palavras, mas ainda assim é cego ao contexto e a semântica.

- TF-IDF: Reduz o impacto de palavras comuns e realça palavras que são mais informativas para cada documento.

A escolha da técnica depende do caso de uso específico. Por exemplo, para tarefas simples de classificação de texto, BoW ou TF-IDF podem ser suficientes. Entretanto, para tarefas que exigem uma compreensão mais profunda do contexto, abordagens mais avançadas como Word2Vec ou embeddings de palavras podem ser mais adequadas.

3.9.5 Implementação em Projetos Reais

A vetorização de texto é frequentemente usada em pipelines de PLN para tarefas como classificação de texto, análise de sentimentos e sistemas de recomendação. Vamos ver um exemplo simples de como essas técnicas podem ser integradas em um pipeline completo.

Código-fonte 3.21: Treino e avaliação de um modelo simples de classificação de sentimentos em textos.

```
1 # pip install scikit-learn
2 from sklearn.feature_extraction.text import TfidfVectorizer
3 from sklearn.naive_bayes import MultinomialNB
4 from sklearn.pipeline import make_pipeline
5 from sklearn.model_selection import train_test_split
6 from sklearn import metrics
7
8 # Exemplo de corpus e rótulos
9 corpus = ["Eu amo programação", "A programação é divertida", "Eu odeio
10          bugs", "A vida é bela", "Eu odeio erros"]
11 labels = [1, 1, 0, 1, 0] # 1: Sentimento Positivo, 0: Sentimento Negativo
12
13 # Divisão dos dados em treino e teste
14 X_train, X_test, y_train, y_test = train_test_split(corpus, labels,
15                                                    test_size=0.4, random_state=42)
16
17 # Criação do pipeline TF-IDF + Classificador Naive Bayes
18 model = make_pipeline(TfidfVectorizer(), MultinomialNB())
19
20 # Treinamento do modelo
21 model.fit(X_train, y_train)
22
23 # Predição no conjunto de teste
```

```
22 predicted = model.predict(X_test)
23
24 # Avaliação do modelo
25 print("Relatório de Classificação:\n",
      metrics.classification_report(y_test, predicted))
```

1	Relatório de Classificação:				
2	precision	recall	f1-score	support	
3	0	0.00	0.00	0.00	1
4	1	0.50	1.00	0.67	1
5	accuracy			0.50	2
6	macro avg	0.25	0.50	0.33	2
7	weighted avg	0.25	0.50	0.33	2

Este exemplo mostra como transformar texto em vetores utilizando TF-IDF e depois aplicar um modelo de classificação simples, como o Naive Bayes. Esse pipeline pode ser adaptado para tarefas mais complexas, incluindo a integração de vetorização com modelos mais avançados.

3.9.6 Embeddings de Palavras

Embeddings de palavras são representações vetoriais densas de palavras que capturam as relações semânticas entre elas. Ao contrário de técnicas como One-Hot Encoding ou Bag of Words, que resultam em vetores esparsos e de alta dimensionalidade, embeddings de palavras mapeiam palavras em um espaço vetorial de dimensões mais baixas, onde palavras com significados semelhantes estão mais próximas. Elas têm várias aplicações em PLN, tais como:

- **Classificação de texto:** Representar documentos usando a média dos embeddings das palavras contidas nele.
- **Análise de sentimentos:** Capturar nuances semânticas que necessárias para entender o sentimento.
- **Sistemas de recomendação:** Usar embeddings para medir similaridade semântica entre produtos ou serviços descritos em texto.

Word2Vec

Word2Vec, introduzido por Mikolov et al. [2013] em 2013, é uma das técnicas mais influentes para aprender embeddings de palavras. Existem duas

abordagens principais no Word2Vec: **Skip-gram** e **CBOW (Continuous Bag of Words)**.

Skip-gram: O modelo Skip-gram prevê as palavras contextuais (palavras ao redor) para uma palavra-alvo. A ideia é maximizar a probabilidade de prever palavras no contexto de uma palavra-alvo específica.

CBOW (Continuous Bag of Words): O modelo CBOW, ao contrário do Skip-gram, prevê a palavra-alvo com base no contexto. Este modelo é útil para capturar o sentido de uma palavra com base em seu ambiente.

Implementação do Word2Vec em Python: Vamos utilizar a biblioteca `gensim`, que fornece uma implementação eficiente do Word2Vec.

Código-fonte 3.22: Treina um modelo de Word2Vec usando o corpus news do Brown (disponível no NLTK) para gerar representações vetoriais de palavras.

```

1 # pip install gensim nltk
2 from gensim.models import Word2Vec
3 from nltk.corpus import brown
4
5 # Carregar o corpus de exemplo (Brown corpus)
6 sentences = brown.sents(categories='news')
7
8 # Treinamento do modelo Word2Vec
9 model = Word2Vec(sentences, vector_size=100, window=5, min_count=5, sg=0)
10
11 # Obtenção de vetor para uma palavra
12 vector = model.wv['economy']
13 print("Vetor para 'economy':", vector)
14
15 # Encontrando palavras semelhantes
16 similar_words = model.wv.most_similar('economy')
17 print("Palavras semelhantes a 'economy':", similar_words)

```

```

1 Vetor para 'economy': [-0.03323277  0.03501032 -0.00576794  0.0111884
2                        -0.03380066 -0.11331949
3                        0.01821606  0.10907217 -0.07971378 -0.06527615  0.03423027 -0.05620026
4                        [...]]
5 0.08788168  0.04192578 -0.03881291 -0.02442176  0.15587321  0.05784223
6 0.05759267 -0.07984415 -0.02128893 -0.00980771]
7 Palavras semelhantes a 'economy': [('additional', 0.9950711727142334),
8                                     ('provide', 0.9949660301208496),
9                                     ('administration', 0.9946332573890686), ('times', 0.9946231245994568),
10                                    ('can', 0.9946150183677673),

```

```

8 ('laws', 0.9945879578590393), ('business', 0.9945263266563416), ('could',
  0.9944885969161987),
9 ('how', 0.9944777488708496), ('made', 0.994467556476593)]

```

Parâmetros Importantes - **vector_size**: Dimensionalidade dos vetores de palavras. - **window**: Tamanho da janela de contexto. - **min_count**: Mínimo de ocorrências para uma palavra ser considerada no treinamento. - **sg**: Se 0, usa CBOW; se 1, usa Skip-gram.

Explorando as Relações Semânticas

Embeddings de palavras como o Word2Vec capturam relações semânticas interessantes, como analogias.

Código-fonte 3.23: Treina um modelo de Word2Vec e faz uma analogia de palavras: "king- "man"+ "woman", buscando o termo mais próximo desse contexto.

```

1 # pip install gensim nltk
2 from gensim.models import Word2Vec
3 from nltk.corpus import brown
4
5 # Carregar o corpus de exemplo (Brown corpus)
6 sentences = brown.sents(categories='news')
7
8 # Treinamento do modelo Word2Vec
9 model = Word2Vec(sentences, vector_size=100, window=5, min_count=1, sg=0)
10
11 # Analogias: "rei" - "homem" + "mulher" = ?
12 result = model.wv.most_similar(positive=['king', 'woman'],
  negative=['man'])
13 print("Resultado da analogia:", result)

```

```

1 Resultado da analogia: [('kid', 0.8575360178947449), ('missed',
  0.8566135764122009), ('load', 0.854703426361084), ('normally',
  0.8536292314529419), ('advisers', 0.8517050743103027), ('Place',
  0.8516084551811218), ('Vieth', 0.8513892292976379), ('yield',
  0.8511804938316345), ('decline', 0.8500499725341797), ('Coe',
  0.8490973114967346)]

```

GloVe (Global Vectors for Word Representation)

Embora o Word2Vec seja amplamente utilizado, outros modelos também oferecem técnicas avançadas para aprender embeddings de palavras. O GloVe,

desenvolvido por Pennington et al. [2014]. em 2014, é uma abordagem baseada em matrizes de ocorrência que combina as vantagens do Word2Vec com informações globais sobre o corpus. Este código a seguir demora um pouco; ele fará o download de um arquivo para seu computador, se estiver usando o Python localmente.

Código-fonte 3.24: Faz o download dos embeddings de palavras GloVe e extrai o arquivo necessário, carrega os vetores e obtém o vetor de características para a palavra `economy`.

```
1 # pip install requests
2 # pip install gensim
3 import os
4 import requests
5 import zipfile
6 from gensim.models import KeyedVectors
7
8 url = "https://nlp.stanford.edu/data/glove.6B.zip"
9 zip_filename = "glove.6B.zip"
10 txt_filename = "glove.6B.50d.txt"
11
12 # Baixar o arquivo zip se não existir
13 if not os.path.exists(zip_filename):
14     print("Baixando embeddings GloVe 6B...")
15     with requests.get(url, stream=True) as r:
16         r.raise_for_status()
17         with open(zip_filename, "wb") as f:
18             for chunk in r.iter_content(chunk_size=8192):
19                 f.write(chunk)
20     print("Download concluído.")
21
22 # Extrair o arquivo txt se não existir
23 if not os.path.exists(txt_filename):
24     print("Extraindo glove.6B.50d.txt...")
25     with zipfile.ZipFile(zip_filename, 'r') as zip_ref:
26         zip_ref.extract(txt_filename)
27     print("Extração concluída.")
28
29 # Carregar embeddings GloVe
30 glove_model = KeyedVectors.load_word2vec_format(txt_filename,
31         binary=False, no_header=True)
32
33 # Obtenção de vetor para uma palavra
34 vector_glove = glove_model['economy']
```

```
34 print("Vetor para 'economy' com GloVe:", vector_glove)
```

```
1 Conversão concluída.
2 Vetor para 'economy' com GloVe: [-1.2027e-01 -7.2505e-01  8.7014e-01
   -6.3944e-01  1.7259e-01 -3.5168e-01 [...] 4.9483e-01 -1.0151e+00
   8.9959e-02  4.6090e-01  1.7585e-03  6.2182e-01
3 1.1893e+00  8.4410e-02]
```

FastText

O FastText, desenvolvido pelo Facebook AI Research, estende o Word2Vec ao considerar subpalavras, o que permite gerar embeddings para palavras que não estão no vocabulário, lidando melhor com palavras raras e morfologicamente ricas.

Código-fonte 3.25: Este código treina um modelo de linguagem FastText usando o corpus news do Brown.

```
1 # pip install gensim nltk
2 import gensim
3 from gensim.models import Word2Vec
4 from nltk.corpus import brown
5
6 # Carregar o corpus de exemplo (Brown corpus)
7 sentences = brown.sents(categories='news')
8
9 # Treinamento do modelo FastText
10 fasttext_model = gensim.models.FastText(sentences, vector_size=100,
11     window=5, min_count=5)
12
13 # Obtenção de vetor para uma palavra
14 vector_fasttext = fasttext_model.wv['economy']
15 print("Vetor para 'economy' com FastText:", vector_fasttext)
16
17 # Encontrando palavras semelhantes
18 similar_words_ft = fasttext_model.wv.most_similar('economy')
19 print("Palavras semelhantes a 'economy' com FastText:", similar_words_ft)
```

```
1 Vetor para 'economy' com FastText: [-7.31775016e-02  2.45773837e-01
   -6.83162957e-02 -2.00441748e-01
2 5.88673912e-02  4.55363989e-02 -7.13645061e-03  1.42574251e-01
3 [...]]
```

```

4 9.44181085e-02 -1.94346443e-01 -2.52598636e-02 5.90402260e-02]
5 Palavras semelhantes a 'economy' com FastText: [('economic',
    0.9999745488166809), ('province', 0.9999632835388184),
    ('recommended', 0.9999628067016602), ('secretary',
    0.9999626278877258), ('generally', 0.9999624490737915), ('defensive',
    0.9999622106552124), ('often', 0.9999621510505676), ('concerned',
    0.9999621510505676), ('maintenance', 0.9999617338180542), ('primary',
    0.9999616742134094)]

```

3.10 Resumo

Nesta seção, cobrimos os fundamentos de Processamento de Linguagem Natural (PLN), incluindo tokenização, lematização, stemização e remoção de stopwords. Também vimos como implementar essas técnicas em Python, utilizando as bibliotecas NLTK e SpaCy.

Além disso, explicamos o que são Intents, um conceito central na arquitetura de chatbots modernos baseados em NLU (Natural Language Understanding). Eles representam o objetivo do usuário e permitem que o chatbot compreenda a intenção por trás das mensagens para agir de forma adequada. Os Intents estão intrinsecamente ligados a outros conceitos fundamentais: (i) Entities: Fornecem os detalhes específicos dentro de um intent. (ii) Utterances: São as diversas maneiras como um usuário pode expressar um mesmo intent. (iii) Actions/Responses: São as tarefas ou respostas que o chatbot executa após identificar um intent. A definição cuidadosa, o treinamento robusto e o gerenciamento contínuo dos intents são relevantes para a eficácia, a inteligência e a qualidade da experiência do usuário oferecida por um chatbot.

Também exploramos várias técnicas de vetorização de texto, desde as mais simples, como One-Hot Encoding, até abordagens mais sofisticadas, como TF-IDF. Com exemplos práticos em Python, demonstramos como essas técnicas podem ser aplicadas em pipelines de PLN. Além disso, abordamos o conceito de embeddings de palavras, com ênfase no Word2Vec, e exploramos outras técnicas como GloVe e FastText. Demonstramos como esses embeddings capturam relações semânticas entre palavras e suas aplicações em tarefas de PLN.

Capítulo 4

Modelos de Linguagem Grande (LLM)

A “inteligência” desses sistemas é uma miragem — o que se vê não é compreensão, mas um ajuste estatístico de padrões.

Emily Bender

Objetivo
Discutir a arquitetura de <i>Transformers</i> , modelos como BERT, GPT e LLaMA, <i>Fine-Tuning</i> e RAG, mostrando como essas técnicas são aplicadas na construção de chatbots modernos.

4.1 Introdução

Um *Large Language Model* (LLM) - em inglês, *Large Language Models* - pode ser definido como um sistema computacional fundamentado em técnicas de aprendizado de máquina, cuja finalidade consiste em gerar texto a partir de uma sequência textual fornecida como entrada. Nesse contexto, o texto de entrada é denominado *prompt*, também referido em inglês como *input*, enquanto o texto produzido pelo modelo é denominado resposta, ou *output*. A figura a seguir apresenta uma representação esquemática do processo de entrada e saída de texto em um LLM.

Figura 4.1: Fluxo entrada e saída do LLM.

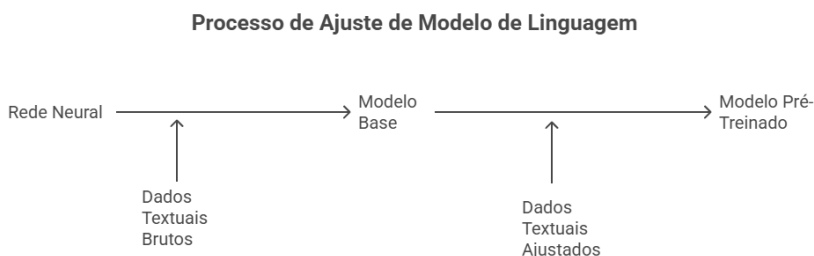


Um LLM nada mais é do que um tipo específico de rede neural artificial (a rede neural profunda) treinada com dados textuais. Em outras palavras, uma grande rede neural treinada com muitos dados. O conceito de “grande rede neural” está relacionado à quantidade de parâmetros que compõem a rede, na casa de milhões em diante; já em relação a “muitos dados” refere-se a dezenas de gigabytes em diante, abrangendo conjuntos de dados brutos que podem incluir livros, artigos, páginas da web, documentos técnicos e outros materiais escritos.

O processo de treino envolve duas etapas (veja Figura 4): (i) um treinamento com dados textuais brutos e (ii) um treinamento com dados textuais anotados (ou seja, mais ajustados) às necessidades de um LLM [Raschka, 2024].

Quando o LLM é treinado com dados brutos, ele é denominado modelo “base” e tem como objetivo prever a próxima sequência de palavras a partir do *prompt* de entrada. Por exemplo, para o *prompt*: O “livro está”, a resposta de um modelo *base* poderia ser: “O livro está em cima da mesa.”. A seguir, no Exemplo 1, apresenta-se um conjunto ilustrativo de dados textuais de treino.

Figura 4.2: Alto nível do processo de treino de um modelo de LLM genérico.



Código-fonte 4.1: Exemplo 1 - Dados textuais brutos para a primeira etapa do treino, omitindo alguns detalhes.

```
1 "Algum tempo hesitei se devia abrir estas memórias pelo princípio ou pelo
   fim, isto é, se poria em primeiro lugar o meu nascimento ou a minha
   morte. Suposto o uso vulgar seja começar pelo nascimento, duas
   considerações me levaram a adotar diferente método: a primeira é que
   eu não sou propriamente um autor defunto, mas um defunto autor, para
   quem a campa foi outro berço; a segunda é que o escrito ficaria assim
   mais galante e mais novo. Moisés, que também contou a sua morte, não
   a pôs no intróito, mas no cabo: diferença radical entre este livro e
   o Pentateuco."
2 Trecho de Assis, Machado. Todos os Romances: Machado de Assis (Portuguese
   Edition) (p. 226). Edição do Kindle.
```

Na segunda etapa, o modelo *base* é treinado novamente, mas agora com dados de texto anotados (por humanos, muitas vezes), o que permite ao modelo aprender não apenas padrões linguísticos gerais, mas também instruções explícitas de interação. Veja o exemplo de dados textuais de treino no Exemplo 2.

Código-fonte 4.2: Exemplo 2 - Dados textuais anotados, omitindo alguns detalhes.

```
1 user: "Quem descobriu o Brasil?"
2 bot: "Que boa pergunta. Quem descobriu o Brasil foi Pedro Álvares Cabral.
   Você quer saber mais sobre o Brasil?"
3 user: "traduza o trecho The book is on the table"
4 bot: "A tradução é: O livro está na mesa. Gostaria de saber mais alguma
   coisa?"
```

Os Exemplos 1 e 2 foram construídos apenas para fins didáticos, com o objetivo de ilustrar, de maneira simplificada, como trechos textuais podem ser utilizados no treinamento de um modelo de linguagem. Em aplicações reais, contudo, o processo é muito mais complexo: envolve conjuntos massivos de dados textuais, frequentemente com bilhões de palavras provenientes de fontes diversas e organizados com formatações específicas. A execução desse treinamento em larga escala só se tornou possível a partir da arquitetura *Transformer*, que introduziu o mecanismo de atenção capaz de lidar, de forma eficiente e paralela, com dependências de longo alcance em sequências de texto.

Modelos de linguagem de larga escala são treinados justamente para prever a próxima palavra em uma sequência, habilidade que serve de base para a construção de representações linguísticas sofisticadas. Esse processo permite que os modelos compreendam contextos extensos e realizem tarefas complexas de PLN, como tradução automática, sumarização de documentos e resposta a perguntas. A capacidade de capturar relações de longo alcance nos textos é um dos diferenciais desses modelos em relação a arquiteturas anteriores.

A introdução dos *Transformers*, proposta por Vaswani et al. [2017] no artigo seminal “*Attention is All You Need*” [Vaswani et al., 2017], representou uma mudança fundamental no campo. A principal inovação foi o mecanismo de atenção, que atribui pesos diferentes às palavras de entrada de acordo com seu contexto, permitindo ao modelo identificar quais termos são mais relevantes em cada situação. Essa característica torna possível o processamento paralelo de sequências longas, acelerando substancialmente o treinamento e aumentando a eficácia na modelagem de dependências distantes.

Antes dessa arquitetura, os modelos de PLN eram dominados por RNNs (*Recurrent Neural Networks*) e LSTMs (*Long Short-Term Memory Networks*). Embora funcionais, esses métodos apresentavam limitações significativas: dificuldade em capturar dependências de longo alcance devido ao problema do *vanishing gradient* e restrições de desempenho, já que processavam o texto de maneira estritamente sequencial, reduzindo o potencial de paralelização. O *Transformer* superou essas limitações ao considerar todas as palavras de entrada simultaneamente, ponderando sua importância relativa por meio da atenção.

Com isso, os LLMs não apenas aumentaram a precisão em tarefas tradicionais de PLN, como também ampliaram o escopo de aplicações, abrangendo geração criativa de texto, tradução contextualizada e correção gramatical au-

tomática. Por outro lado, o avanço desses modelos também trouxe questionamentos relacionados à ética, viés e uso responsável da inteligência artificial, aspectos que vêm sendo discutidos intensamente pela comunidade científica e que serão abordados no Capítulo 5.

4.2 Arquitetura Geral do *Transformer*

Os LLMs modernos são em grande parte fundamentados na arquitetura *Transformer*, proposta originalmente por Vaswani et al. [2017]. Essa arquitetura se tornou a base de inúmeros avanços no processamento de linguagem natural porque oferece uma forma eficiente e escalável de lidar com sequências de texto. Diferente de abordagens anteriores, que processavam as palavras uma a uma em ordem, os *Transformers* permitem que todo o contexto seja considerado em paralelo, o que torna o treinamento e a inferência mais rápidos e flexíveis.

A estrutura geral de um *Transformer* é composta por blocos que se repetem, cada um deles combinando mecanismos de atenção e redes neurais totalmente conectadas (*feedforward*). Em sua formulação original, a arquitetura possui dois grandes componentes: o **encoder**, responsável por transformar a sequência de entrada em uma representação interna, e o **decoder**, que utiliza essa representação para gerar uma saída, como no caso da tradução automática. Embora alguns modelos atuais utilizem apenas a parte do encoder (como o BERT) ou apenas o decoder (como o GPT), o princípio fundamental continua o mesmo: ambos dependem intensamente do mecanismo de atenção.

Cada camada do encoder e do decoder contém três elementos centrais:

- **Mecanismo de atenção múltipla (Multi-Head Attention):** avalia, em paralelo, diferentes formas de relacionar as palavras entre si.
- **Redes *feedforward*:** transformam as representações intermediárias, permitindo maior capacidade de modelagem.
- **Normalização e conexões residuais:** estabilizam o treinamento e ajudam a preservar informações ao longo das camadas.

O conceito de **self-attention** é um dos pontos mais inovadores. Nesse mecanismo, cada palavra da sequência não é processada isoladamente, mas em comparação com todas as outras palavras da mesma sequência. Isso permite que o modelo entenda relações de dependência de longo alcance, como entre

o sujeito no início de uma frase e o verbo que aparece muito depois. O **multi-head attention** amplia essa ideia ao aplicar várias atenções em paralelo, cada uma aprendendo a capturar um tipo de relação — algumas cabeças podem identificar proximidade sintática, enquanto outras captam conexões semânticas mais distantes.

Um aspecto central da arquitetura é que todo esse processamento ocorre de forma paralela. Diferente das redes recorrentes (*RNNs*) ou das *LSTMs*, que analisavam o texto palavra por palavra em sequência, o *Transformer* avalia a frase inteira de uma só vez. Essa mudança de paradigma permitiu que modelos fossem treinados em coleções massivas de texto, chegando a bilhões de parâmetros e escalando para tamanhos sem precedentes.

Além de sua eficiência estrutural, o *Transformer* também é flexível: pode ser adaptado a diferentes tarefas com pequenas modificações. Modelos **autoregressivos**, como a família GPT, são baseados apenas no decoder e são treinados para prever a próxima palavra em uma sequência de forma unidirecional. Já os modelos **bidirecionais**, como o BERT, exploram tanto o contexto à esquerda quanto o à direita, o que os torna particularmente eficazes em tarefas de compreensão textual.

O impacto da arquitetura *Transformer* é amplo e vai além do processamento de linguagem natural em sua forma mais básica. A combinação entre paralelismo, capacidade de capturar dependências de longo alcance e flexibilidade estrutural abriu caminho para a criação de famílias inteiras de modelos com diferentes finalidades. Alguns deles utilizam apenas o encoder, outros apenas o decoder, e há ainda os que combinam os dois de formas distintas. Essa diversidade deu origem a arquiteturas conhecidas, como BERT, GPT, T5 e LLaMA, que se tornaram referências na área e serão discutidas a seguir em maior detalhe.

4.2.1 BERT (*Bidirectional Encoder Representations from Transformers*)

O BERT (*Bidirectional Encoder Representations from Transformers*) introduziu uma nova forma de treinamento em larga escala, na qual o modelo considera tanto o contexto à esquerda quanto o contexto à direita de uma palavra-alvo. Essa abordagem permitiu uma compreensão mais rica da linguagem em comparação com modelos anteriores, que eram predominantemente unidirecionais.

O BERT é construído sobre a pilha de *encoders* do *Transformer*. Isso significa que ele não é projetado para gerar texto de forma autoregressiva, mas sim para produzir representações contextuais profundas de palavras e frases, que podem depois ser utilizadas em diversas tarefas de PLN. Essa característica torna o BERT especialmente adequado para problemas de compreensão textual, como classificação, resposta a perguntas e extração de informações.

O modelo é pré-treinado em uma grande quantidade de texto utilizando duas tarefas principais:

- **Masked Language Modeling (MLM)**: Palavras aleatórias em uma sequência são substituídas por um token especial de máscara, e o modelo é treinado para prever as palavras originais com base no restante do contexto. Esse procedimento permite que o BERT aprenda relações bidirecionais, uma vez que a previsão de uma palavra depende de termos anteriores e posteriores na frase.
- **Next Sentence Prediction (NSP)**: O modelo recebe pares de frases e deve identificar se a segunda frase segue logicamente a primeira. Essa tarefa ajuda o BERT a capturar relações discursivas entre sentenças, algo essencial em aplicações como resposta a perguntas e inferência textual.

Após o pré-treinamento, o BERT pode ser ajustado finamente (*Fine-Tuning*) para tarefas específicas, muitas vezes com datasets muito menores que aqueles usados no pré-treinamento. Esse processo de adaptação é o que permite que um único modelo, previamente treinado de maneira genérica, seja aplicado em uma ampla gama de problemas.

Código-fonte 4.3: Uso do BERT.

```
1 from transformers import BertTokenizer, BertForSequenceClassification
2 from torch.nn.functional import softmax
3
4 # Carregar o tokenizer e o modelo BERT
5 tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
6 model = BertForSequenceClassification.from_pretrained(
7     'bert-base-uncased')
8
9 # Entrada de exemplo
10 input_text = "Chatbots são muito úteis para automação."
11 # Tokenização
```

```
12 input_ids = tokenizer(input_text, return_tensors='pt')
13
14 # Predição
15 outputs = model(**input_ids)
16 probs = softmax(outputs.logits, dim=-1)
17
18 print("Probabilidades de classe:", probs)
```

```
1 Probabilidades de classe: tensor([[0.7097, 0.2903]],
  grad_fn=<SoftmaxBackward0>)
```

O BERT rapidamente estabeleceu novos patamares em benchmarks de PLN, como GLUE (General Language Understanding Evaluation) e SQuAD (Stanford Question Answering Dataset). Sua eficácia impulsionou o desenvolvimento de uma série de variações e aprimoramentos, como RoBERTa, DistilBERT e ALBERT, que buscaram melhorar desempenho, eficiência ou reduzir custos de treinamento.

O BERT é amplamente utilizado em:

- **Classificação de Texto:** Análise de sentimento, detecção de spam, categorização de documentos.
- **Respostas a Perguntas:** Modelos que identificam trechos relevantes em um texto para responder perguntas formuladas em linguagem natural.
- **Extração de Informações:** Identificação de entidades nomeadas (pessoas, lugares, organizações) e relações entre elas.

Apesar de seu impacto, o BERT apresenta limitações relevantes. O modelo é pesado em termos computacionais, exigindo grande capacidade de processamento para treinamento e mesmo para inferência em aplicações práticas. Além disso, sua janela de contexto é limitada, o que dificulta o processamento de documentos muito extensos sem estratégias adicionais de segmentação.

Código-fonte 4.4: Exemplo das representações do BERT.

```
1 from transformers import BertTokenizer, BertModel
2
3 # Carregar o tokenizer e o modelo BERT pré-treinado
4 tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
5 model = BertModel.from_pretrained('bert-base-uncased')
6
```

```

7 # Exemplo de texto
8 texto = "Machine learning is fascinating."
9
10 # Tokenização
11 input_ids = tokenizer(texto, return_tensors='pt')['input_ids']
12
13 # Obtenção das representações do modelo BERT
14 outputs = model(input_ids)
15 last_hidden_states = outputs.last_hidden_state
16
17 print("Representações BERT:", last_hidden_states)

```

```

1 Representações BERT: tensor([[[ 0.0836,  0.0931, -0.2452, ..., -0.3743,
    0.0502,  0.5606],
2 [ 0.2351,  0.0642, -0.1859, ..., -0.1147,  0.5381,  0.5704],
3 ...,
4 [ 0.8129,  0.1013, -0.2131, ...,  0.0133, -0.6965, -0.0085]]],
5 grad_fn=<NativeLayerNormBackward0>)

```

4.2.2 GPT (*Generative Pre-trained Transformer*)

O *Generative Pre-trained Transformer* (GPT) é um dos LLMs mais conhecidos. Ele é treinado de forma autoregressiva, o que significa que prediz a próxima palavra em uma sequência, dada a entrada anterior. Isso o torna excelente para tarefas de geração de texto. O GPT2 foi uma versão inicial do GPT, contendo 1,5 bilhões de parâmetros. Ele mostrou que, ao ser treinado em grandes quantidades de texto, poderia gerar conteúdo coerente e complexo. GPT3 é uma evolução ainda maior, com 175 bilhões de parâmetros. Esse modelo pode realizar uma ampla gama de tarefas de PLN sem a necessidade de ajustes finos específicos, simplesmente recebendo exemplos de como a tarefa deve ser executada (aprendizado por poucos exemplos, ou few-shot learning).

Código-fonte 4.5: Geração de texto com GPT-2.

```

1 from transformers import GPT2Tokenizer, GPT2LMHeadModel
2
3 # Carregar o tokenizer e o modelo GPT-2
4 tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
5 model = GPT2LMHeadModel.from_pretrained('gpt2')
6

```

```

7 # Entrada de exemplo
8 input_text = "Chatbots modernos podem"
9
10 # Tokenização
11 input_ids = tokenizer.encode(input_text, return_tensors='pt')
12
13 # Geração de texto
14 outputs = model.generate(input_ids, max_length=50, num_return_sequences=1)
15
16 # Decodificação e exibição do texto gerado
17 generated_text = tokenizer.decode(outputs[0], skip_special_tokens=True)
18 print("Texto gerado:", generated_text)

```

```

1 Texto gerado: Chatbots modernos podem model.com The following is a list of
  all the bots that have been added to the podemodel.com community. Bot
  Name Description bot_bot_name_id bot_

```

O GPT tem sido aplicado em diversas áreas, incluindo:

- **Geração de Texto:** Criação de conteúdo, histórias, artigos, etc.
- **Assistentes Virtuais:** Implementação de sistemas de diálogo baseados em IA.
- **Tradução Automática:** Utilização de contexto amplo para melhorar a tradução entre idiomas.

O GPT é um modelo autoregressivo que se concentra na geração de texto. É treinado para prever a próxima palavra em uma sequência, o que o torna excelente para tarefas de geração de texto, como chatbots.

Código-fonte 4.6: Outro exemplo de geração de texto com GPT-2.

```

1 from transformers import GPT2Tokenizer, GPT2LMHeadModel
2
3 # Carregar o tokenizer e o modelo GPT-2 pré-treinado
4 tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
5 model = GPT2LMHeadModel.from_pretrained('gpt2')
6
7 # Exemplo de texto
8 input_text = "In the future, artificial intelligence will"
9
10 # Tokenização
11 input_ids = tokenizer.encode(input_text, return_tensors='pt')
12

```

```

13 # Geração de texto
14 outputs = model.generate(input_ids, max_length=50, num_return_sequences=1)
15
16 # Decodificação e exibição do texto gerado
17 generated_text = tokenizer.decode(outputs[0], skip_special_tokens=True)
18 print("Texto gerado:", generated_text)

```

```

1 Texto gerado: In the future, artificial intelligence will be able to do
  things like search for information about people, and to do things
  like search for information about people. We're going to see a lot of
  things that are going to be very interesting in

```

Além de BERT e GPT, há muitos outros modelos baseados em *Transformers* projetados para tarefas específicas. Alguns exemplos incluem o T5 (*Text-To-Text Transfer Transformer*) que converte qualquer tarefa de PLN em um problema de tradução; o XLNet que combina ideias de BERT e *Transformers* autoregressivos para melhorar a modelagem de dependências de longo alcance; o RoBERTa (A Robustly Optimized BERT Pretraining Approach) que é uma variação do BERT com treinamento aprimorado, além do DistilBERT.

O T5 transforma qualquer tarefa de PLN em um problema de tradução, onde a entrada e a saída são tratadas como texto. Isso simplifica o Fine-Tuning para diferentes tarefas.

Código-fonte 4.7: Realiza uma tradução automática de uma frase usando o T5.

```

1 # pip install transformers[torch]
2 # pip install sentencepiece
3
4 from transformers import T5Tokenizer, T5ForConditionalGeneration
5
6 # Carregar o tokenizer e o modelo T5
7 tokenizer = T5Tokenizer.from_pretrained( 't5-small')
8 model = T5ForConditionalGeneration.from_pretrained( 't5-small')
9
10 # Entrada de exemplo
11 input_text = "translate English to German: The weather is nice today."
12
13 # Tokenização
14 input_ids = tokenizer.encode(input_text, return_tensors='pt')
15
16 # Geração de texto
17 outputs = model.generate(input_ids)

```

```
18 |
19 | # Decodificação e exibição do texto gerado
20 | generated_text = tokenizer.decode(outputs[0], skip_special_tokens=True)
21 | print("Tradução gerada:", generated_text)
```

```
1 | Tradução gerada: Das Wetter ist heute schon.
```

Cabe ressaltar ainda o XLNet e o DistilBERT. O XLNet combina vantagens dos modelos autoregressivos e bidirecionais, como o GPT e BERT, para capturar dependências de longo alcance de forma mais eficiente. Já o DistilBERT é uma versão reduzida do BERT, com menos parâmetros, mas mantendo uma alta performance, o que o torna mais eficiente para uso em produção.

Código-fonte 4.8: um exemplo de geração de texto com IA usando XLNet.

```
1 | from transformers import XLNetTokenizer, XLNetLMHeadModel
2 |
3 | # Carregar o tokenizer e o modelo XLNet
4 | tokenizer = XLNetTokenizer.from_pretrained( 'xlnet-base-cased')
5 | model = XLNetLMHeadModel.from_pretrained( 'xlnet-base-cased')
6 |
7 | # Entrada de exemplo
8 | input_text = "Natural Language Processing is"
9 |
10 | # Tokenização
11 | input_ids = tokenizer.encode(input_text, return_tensors='pt')
12 |
13 | # Geração de texto
14 | outputs = model.generate(input_ids, max_length=50, num_return_sequences=1)
15 |
16 | # Decodificação e exibição do texto gerado
17 | generated_text = tokenizer.decode(outputs[0], skip_special_tokens=True)
18 | print("Texto gerado com XLNet:", generated_text)
```

```
1 | Texto gerado com XLNet: Natural Language Processing is and in and in [...]
```

4.2.3 distilbert-base-uncased

O modelo distilbert-base-uncased [Akhila et al., 2023] foi lançado em 2019, sendo menor e mais rápido que o BERT e otimizado para tarefas que processam frases ou sentenças. Sua versão *uncased* é útil para descrições de código e

problemas, já que não considera diferenças de maiúsculas/minúsculas. Ele é pré-treinado em um grande corpus, o que ajuda na generalização, e tem uma boa arquitetura para esse tipo de tarefa, além de exigir hardware acessível.

O *distilbert* resulta de um processo de knowledge distillation que reduz em aproximadamente 40% o número de parâmetros e acelera a inferência em cerca de 60%, preservando 95–97% da acurácia do BERT-base (original) em benchmarks de compreensão de linguagem natural [Sanh et al., 2019]. Essa compacidade de fornecer representações ricas com custo computacional inferior é particularmente vantajosa em ambientes de hardware moderado. Com apenas 67M de parâmetros [Sanh et al., 2019], a variante uncased cabe em uma GPU modesta (por exemplo, 6 GB de RAM da placa de vídeo), possibilitando fine-tuning e inferência dentro de recursos computacionais restritos. Assim, torna-se viável re-treinar o modelo à medida que novos dados de projeto se acumulam, mantendo a acurácia sem investir em infraestruturas onerosas.

Por fim, o amplo suporte no ecossistema *Hugging Face* para os modelos do tipo BERT e para outros modelos simplifica a reprodutibilidade e a integração em pipelines de tarefas de processamento de linguagem natural.

4.3 *Fine-Tuning* de Modelos Pré-Treinados

Modelos pré-treinados como BERT e GPT demonstraram capacidade de compreender nuances semânticas em texto e transferir esse conhecimento para diversas tarefas específicas através de Fine-Tuning. [Minaee et al., 2025].

O *Fine-Tuning* de modelos pré-treinados é uma técnica fundamental no Processamento de Linguagem Natural (PLN) moderno, especialmente ao trabalhar com Modelos de Linguagem Grande (LLMs). *Fine-Tuning* permite adaptar um modelo geral para tarefas específicas, como classificação de texto, análise de sentimentos ou geração de linguagem, utilizando um conjunto de dados menor e específico.

Fine-Tuning é o processo de tomar um modelo pré-treinado em uma grande quantidade de dados gerais e adaptá-lo para uma tarefa específica. Este processo envolve ajustar os pesos do modelo, mas com uma taxa de aprendizado menor para não “desaprender” o que foi aprendido durante o pré-treinamento.

Modelos como BERT [Devlin et al., 2019], GPT [Radford et al., 2019], e T5 [Raffel et al., 2020] são comumente fine-tuned para tarefas específicas.

Uma alternativa eficiente ao *Fine-Tuning* completo é o uso de métodos de adaptação com baixo número de parâmetros, como o LoRA (*Low-Rank Adaptation*). Em vez de atualizar todos os pesos do modelo pré-treinado, o LoRA introduz pequenas matrizes adicionais de baixa dimensão que são ajustadas durante o treinamento, enquanto os pesos originais permanecem congelados. Essa técnica reduz drasticamente o número de parâmetros que precisam ser treinados, diminuindo o custo computacional e de armazenamento. Na prática, isso torna viável aplicar *Fine-Tuning* em LLMs muito grandes, mesmo em ambientes com recursos limitados, preservando boa parte do desempenho obtido pelo ajuste completo.

A principal vantagem do *Fine-Tuning* é a eficiência, pois permite que os modelos aprendam rapidamente uma nova tarefa, utilizando relativamente poucos dados. Além disso, modelos pré-treinados já capturam padrões linguísticos gerais, o que torna o Fine-Tuning uma abordagem útil para resolver problemas específicos sem precisar treinar um modelo do zero. O processo de *Fine-Tuning* geralmente envolve os seguintes passos:

- **Escolha do Modelo:** Selecionar um modelo pré-treinado adequado para a tarefa. Modelos como BERT e GPT são populares devido à sua versatilidade.
- **Preparação dos Dados:** Os dados precisam estar formatados de maneira que sejam compatíveis com a tarefa específica, como classificação de texto ou resposta a perguntas.
- **Configuração do Treinamento:** Ajuste de hiperparâmetros como a taxa de aprendizado, número de épocas e tamanho do lote.
- **Treinamento:** Executar o treinamento do modelo no conjunto de dados específico.
- **Avaliação:** Avaliar o desempenho do modelo ajustado em um conjunto de validação ou teste.

Vamos realizar o *Fine-Tuning* de um modelo BERT para uma tarefa de classificação de sentimentos usando o conjunto de dados IMDB.

Cabe ressaltar que a biblioteca *transformers* da Hugging Face, utilizada neste exemplo, tornou-se a ferramenta de referência para trabalhar com mo-

delos baseados em *Transformers*. Ela oferece uma ampla gama de modelos pré-treinados que podem ser facilmente integrados em pipelines de PLN.

O código a seguir pode levar de 30 minutos a 2 horas para finalizar em um computador com uma boa GPU. Em CPU, pode demorar várias horas ou até mais de um dia. Uma sugestão é utilizar o Google Colab.

Código-fonte 4.9: O código treina e avalia um modelo BERT para classificar sentimentos em textos do IMDb.

```
1 # pip install datasets
2 # pip install transformers[torch]
3
4 from transformers import BertTokenizer, BertForSequenceClassification,
5     Trainer, TrainingArguments
6 from datasets import load_dataset
7
8 import os
9 os.environ["WANDB_DISABLED"] = "true"
10
11 # Carregar o dataset IMDb
12 dataset = load_dataset("imdb")
13
14 # Carregar o tokenizer e o modelo BERT
15 tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
16 model = BertForSequenceClassification.from_pretrained(
17     'bert-base-uncased')
18
19 # Tokenizar os dados
20 def tokenize_function(examples):
21     return tokenizer(examples['text'], padding='max_length', truncation=True)
22
23 tokenized_datasets = dataset.map(tokenize_function, batched=True)
24
25 # Definir argumentos de treinamento
26 training_args = TrainingArguments(
27     output_dir='./results',
28     learning_rate=2e-5,
29     report_to="none",
30     per_device_train_batch_size=16,
31     per_device_eval_batch_size=16,
32     num_train_epochs=3,
33     weight_decay=0.01,
```

```
34 # Criar o Trainer
35 trainer = Trainer(
36 model=model,
37 args=training_args,
38 train_dataset=tokenized_datasets['train'],
39 eval_dataset=tokenized_datasets['test'],
40 )
41
42 # Treinar o modelo
43 trainer.train()
44
45 # Avaliar o modelo
46 eval_result = trainer.evaluate()
47 print(f"Resultado da Avaliação: {eval_result}")
```

```
1 Resultado da Avaliação: {'eval_loss': 0.2594565749168396, 'eval_runtime':
    172.544, 'eval_samples_per_second': 144.891, 'eval_steps_per_second':
    9.059, 'epoch': 3.0}
```

Neste exemplo, utilizamos o modelo BERT e a biblioteca `datasets` da Hugging Face para carregar o conjunto de dados IMDB, que é usado para tarefas de classificação de sentimentos. O processo de tokenização é realizado com o `BertTokenizer`, seguido pelo treinamento do modelo usando o `Trainer`, que automatiza o processo de *Fine-Tuning*.

Embora o *Fine-Tuning* seja uma técnica com potencial, é importante considerar alguns desafios:

- **Overfitting:** Ajustar demais o modelo para os dados de treinamento específicos pode reduzir a generalização para novos dados.
- **Biases Inerentes:** Se o modelo pré-treinado já contém vieses, o *Fine-Tuning* pode reforçá-los, especialmente se os dados de treinamento forem limitados ou enviesados.
- **Requisitos Computacionais:** *Fine-Tuning* de LLMs pode ser computacionalmente intensivo, especialmente para modelos maiores como GPT-3.

Além disso, algumas abordagens avançadas para melhorar o processo de *Fine-Tuning* incluem:

- **Learning Rate Warmup:** Aumentar gradualmente a taxa de aprendizado no início do treinamento para evitar grandes atualizações de peso que poderiam desestabilizar o modelo.

- **Layer-Wise Learning Rate Decay:** Aplicar diferentes taxas de aprendizado para diferentes camadas do modelo, com camadas inferiores aprendendo mais lentamente.
- **Data Augmentation:** Aumentar a diversidade do conjunto de dados de treinamento para melhorar a robustez do modelo.

O *Fine-Tuning* tem uma vasta gama de aplicações em PLN, incluindo:

- **Classificação de Texto:** Análise de sentimentos, detecção de spam, categorização de notícias.
- **Respostas a Perguntas:** Modelos que respondem a perguntas baseadas em um contexto textual específico.
- **Geração de Texto:** *Fine-Tuning* de modelos como GPT para gerar textos específicos de um domínio, como redação de artigos científicos.
- **Tradução Automática:** Adaptação de modelos de tradução para dialetos ou linguagens específicas.

4.4 Few Shot e Zero Shot Learning

Zero Shot: A abordagem LLM zero-shot learning refere-se à capacidade dos LLMs de resolver tarefas sem a necessidade de exemplos explícitos fornecidos durante a inferência. Nesse contexto, a tarefa de processamento de linguagem natural *text classification* é especificada unicamente por meio de uma instrução textual (prompt), e o modelo deve inferir a ação esperada com base em seu conhecimento prévio adquirido durante o pré-treinamento [Radford et al., 2019].

Few Shot: Já o few-shot learning caracteriza-se pela inclusão de um pequeno conjunto de exemplos da tarefa no próprio prompt, com o objetivo de guiar a geração do modelo durante a inferência. Essa técnica permite ao modelo identificar padrões desejados com base nos exemplos fornecidos e aplicá-los a novos casos, mesmo sem reconfiguração ou ajuste de parâmetros. Trata-se de uma abordagem intermediária entre o zero-shot e o treinamento supervisionado tradicional, sendo especialmente eficaz em tarefas de classificação com variações contextuais [Raschka, 2024]. Sua principal vantagem está na adaptação rápida a novas tarefas com custo computacional reduzido.

4.5 Retrieval-Augmented Generation

O *Retrieval-Augmented Generation* (RAG) é uma abordagem que combina duas técnicas na área de processamento de linguagem natural: recuperação de informações e geração de texto. A ideia central do RAG é aprimorar a capacidade de um modelo de linguagem ao integrá-lo com um sistema de recuperação que busca informações relevantes de uma base de dados ou de um conjunto de documentos.

Na prática, o RAG opera em duas etapas. Primeiro, quando uma consulta ou pergunta é feita, um mecanismo de recuperação é acionado para identificar e extrair informações pertinentes de um repositório de dados. Isso pode incluir documentos, artigos ou qualquer outro tipo de conteúdo textual que possa fornecer contexto e detalhes adicionais sobre o tema em questão. Essa fase garante que o modelo de linguagem tenha acesso a informações atualizadas e específicas, em vez de depender apenas do conhecimento prévio que foi incorporado durante seu treinamento.

Em seguida, na segunda etapa, o modelo de linguagem utiliza as informações recuperadas para gerar uma resposta mais rica e contextualizada. Essa geração não se limita a reproduzir o conteúdo recuperado, mas sim a integrar esses dados de forma coesa, criando uma resposta que não apenas responde à pergunta, mas também fornece uma narrativa mais completa e informativa. Isso resulta em respostas que são mais precisas e relevantes, pois são fundamentadas em dados concretos e atualizados.

A combinação dessas duas etapas permite que o RAG supere algumas limitações dos modelos de linguagem tradicionais, que podem falhar em fornecer informações precisas ou atualizadas, especialmente em domínios que evoluem rapidamente. Além disso, essa abordagem é particularmente útil em aplicações como assistentes virtuais, chatbots e sistemas de perguntas e respostas, onde a precisão e a relevância da informação são fatores-chave na experiência do usuário.

Portanto, ele é uma técnica que não apenas melhora a qualidade das respostas geradas por modelos de linguagem, mas também amplia o alcance e a aplicabilidade desses modelos em cenários do mundo real, onde a informação é dinâmica e em constante evolução.

O RAG une dois componentes principais, a Recuperação de Informação (*Retrieval*): envolve buscar documentos, parágrafos ou passagens relevantes a partir de uma grande coleção de dados; e a Geração de Texto (*Generation*): uma vez que a informação relevante é recuperada, um modelo de linguagem, como GPT ou BART, é utilizado para gerar uma resposta coerente e informativa baseada nas informações recuperadas. Dessa forma, RAG é capaz de responder a perguntas e gerar conteúdo que não apenas utiliza o contexto imediato, mas também consulta uma base de conhecimento externa, aumentando a precisão e a relevância das respostas.

Vamos implementar um exemplo simples de RAG usando as bibliotecas da Hugging Face, incluindo o modelo DPR para recuperação e o modelo BART para geração de texto.

Recuperação de Passagens com DPR: Primeiro, precisamos carregar e configurar o modelo DPR para recuperar passagens relevantes a partir de uma base de dados. Sugerimos a execução do código abaixo no Google Colab.

Código-fonte 4.10: O código identifica qual passagem é mais relevante para responder à consulta fornecida, usando embeddings densos e busca vetorial.

```

1 from transformers import DPRQuestionEncoder, DPRQuestionEncoderTokenizer
2 from transformers import DPRContextEncoder, DPRContextEncoderTokenizer
3 from transformers import BertTokenizer, BertModel
4 import torch
5
6 # Carregar o tokenizer e o modelo para as consultas (questions)
7 question_tokenizer = DPRQuestionEncoderTokenizer.from_pretrained(
8     "facebook/dpr-question_encoder-single-nq-base")
9 question_encoder = DPRQuestionEncoder.from_pretrained(
10    "facebook/dpr-question_encoder-single-nq-base")
11
12 # Carregar o tokenizer e o modelo para os contextos (passages)
13 context_tokenizer = DPRContextEncoderTokenizer.from_pretrained(
14     "facebook/dpr-ctx_encoder-single-nq-base")
15 context_encoder = DPRContextEncoder.from_pretrained(
16     "facebook/dpr-ctx_encoder-single-nq-base")
17
18 # Exemplo de consulta
19 query = "What is Retrieval-Augmented Generation?"
20
21 # Codificar a consulta
22 query_input = question_tokenizer(query, return_tensors="pt")

```

```

21 query_embedding = question_encoder(**query_input).pooler_output
22
23 # Exemplo de passagens
24 passages = [
25     "Retrieval-Augmented Generation is a technique that combines
26     retrieval of relevant information with text generation.",
27     "It allows for more accurate and contextually relevant answers by
28     consulting external knowledge bases.",
29     "RAG is particularly useful in scenarios where the information
30     required to answer a query is not present in the training data of
31     the language model."
32 ]
33
34 # Codificar as passagens
35 passage_inputs = context_tokenizer(passages, padding=True,
36     truncation=True, return_tensors="pt")
37 passage_embeddings = context_encoder(
38     **passage_inputs).pooler_output
39
40 # Calcular similaridade e selecionar a passagem mais relevante
41 similarity_scores = torch.matmul(query_embedding, passage_embeddings.T)
42 best_passage_index = torch.argmax(similarity_scores, dim=1).item()
43 best_passage = passages[best_passage_index]
44
45 print("Passagem mais relevante:", best_passage)

```

```

1 Passagem mais relevante: Retrieval-Augmented Generation is a technique
  that combines retrieval of relevant information with text generation

```

Neste exemplo, utilizamos o modelo DPR para codificar uma consulta e várias passagens, e então calculamos a similaridade entre a consulta e as passagens para recuperar a mais relevante.

Geração de Texto com BART: Uma vez que a passagem mais relevante foi recuperada, utilizamos o modelo BART para gerar uma resposta coerente.

Código-fonte 4.11: O código busca identificar, entre vários textos, qual é o mais relevante para uma pergunta, usando embeddings e similaridade vetorial. Depois, prepara um modelo de geração para responder à consulta usando o contexto selecionado.

```

1 # pip install transformers[torch]
2 from transformers import DPRQuestionEncoder, DPRQuestionEncoderTokenizer
3 from transformers import DPRContextEncoder, DPRContextEncoderTokenizer
4 from transformers import BartTokenizer, BartForConditionalGeneration

```



```

5 from transformers import BertTokenizer, BertModel
6 import torch
7
8 # Carregar o tokenizer e o modelo para as consultas (questions)
9 question_tokenizer = DPRQuestionEncoderTokenizer.from_pretrained(
10 "facebook/dpr-question_encoder-single-nq-base")
11 question_encoder = DPRQuestionEncoder.from_pretrained(
12 "facebook/dpr-question_encoder-single-nq-base")
13
14 # Carregar o tokenizer e o modelo para os contextos (passages)
15 context_tokenizer = DPRContextEncoderTokenizer.from_pretrained(
16 "facebook/dpr-ctx_encoder-single-nq-base")
17 context_encoder = DPRContextEncoder.from_pretrained(
18 "facebook/dpr-ctx_encoder-single-nq-base")
19
20 # Exemplo de consulta
21 query = "What is Retrieval-Augmented Generation?"
22
23 # Codificar a consulta
24 query_input = question_tokenizer(query, return_tensors="pt")
25 query_embedding = question_encoder(**query_input).pooler_output
26
27 # Exemplo de passagens
28 passages = [
29     "Retrieval-Augmented Generation is a technique that combines
30     retrieval of relevant information with text generation.",
31     "It allows for more accurate and contextually relevant answers by
32     consulting external knowledge bases.",
33     "RAG is particularly useful in scenarios where the information
34     required to answer a query is not present in the training data of
35     the language model."
36 ]
37
38 # Codificar as passagens
39 passage_inputs = context_tokenizer(passages, padding=True,
40 truncation=True, return_tensors="pt")
41 passage_embeddings = context_encoder
42 **passage_inputs).pooler_output
43
44 # Calcular similaridade e selecionar a passagem mais relevante
45 similarity_scores = torch.matmul(query_embedding, passage_embeddings.T)
46 best_passage_index = torch.argmax(similarity_scores, dim=1).item()
47 best_passage = passages[best_passage_index]

```

```
42 # Carregar o tokenizer e o modelo BART
43 bart_tokenizer = BartTokenizer.from_pretrained(
44     "facebook/bart-large")
45 bart_model = BartForConditionalGeneration.from_pretrained(
46     "facebook/bart-large")
47
48 # Concatenar a consulta com a passagem relevante
49 input_text = query + " " + best_passage
50
51 # Codificar e gerar resposta
52 input_ids = bart_tokenizer.encode(input_text, return_tensors="pt")
53 generated_ids = bart_model.generate(input_ids, max_length=50,
54     num_beams=4, early_stopping=True)
55 generated_text = bart_tokenizer.decode(generated_ids[0],
56     skip_special_tokens=True)
57
58 print("Resposta gerada:", generated_text)
```

```
1 Resposta gerada: What is Retrieval-Augmented Generation?Retrieval and
Augmented Generation is a technique that combines retrieval of
relevant information with text generation.What is retrieval?
```

Este código gera uma resposta baseada na passagem recuperada, criando uma resposta informativa que combina a informação relevante com a geração de texto fluida.

A técnica RAG tem várias aplicações práticas, incluindo:

- **Sistemas de Resposta a Perguntas:** Sistemas que precisam consultar bases de conhecimento extensivas para responder a perguntas de forma precisa.
- **Assistentes Virtuais:** Assistentes que necessitam de acesso a informações específicas e detalhadas, além do treinamento inicial do modelo.
- **Geração de Conteúdo:** Criação de conteúdo especializado que requer consulta de fontes externas para garantir precisão e relevância.

A técnica RAG também apresenta alguns desafios:

- **Escalabilidade:** A recuperação de informações em bases de dados muito grandes pode ser computacionalmente intensiva.
- **Relevância das Passagens:** A qualidade das respostas geradas depende fortemente da relevância das passagens recuperadas.

- **Treinamento Conjunto:** Treinar os componentes de recuperação e geração de maneira conjunta pode ser complexo e requer grandes volumes de dados.

4.6 LLaMA: Modelos Pequenos e Eficientes

LLaMA é uma família de modelos de linguagem grandes (LLMs) que são menores em tamanho, mas ainda mantêm a capacidade de realizar tarefas complexas de PLN. A abordagem de LLaMA é baseada em uma arquitetura de *Transformer*, semelhante a outros LLMs, mas otimizada para eficiência em termos de parâmetros e recursos computacionais.

Características Principais

- **Tamanho Reduzido:** LLaMA é projetado para ser mais leve que os modelos gigantescos, com diferentes variantes que variam de 7B a 65B parâmetros.
- **Eficiência Computacional:** Devido ao seu design otimizado, o LLaMA pode ser treinado em menos tempo e com menos recursos, tornando-o acessível para organizações menores e pesquisadores.
- **Versatilidade:** Apesar de seu tamanho reduzido, o LLaMA é capaz de realizar uma ampla gama de tarefas de PLN, incluindo geração de texto, tradução, e compreensão de linguagem.

4.6.1 Arquitetura do LLaMA

A arquitetura do LLaMA é baseada no *Transformer*, mas com várias otimizações que permitem que ele mantenha uma alta qualidade de predição, enquanto usa menos parâmetros e recursos computacionais.

Camadas *Transformer* Otimizadas

O LLaMA utiliza camadas *Transformer* com melhorias específicas para otimizar o uso de memória e tempo de processamento. As principais diferenças incluem:

- **Atenção Multi-Head Otimizada:** Reduz a redundância ao calcular a atenção em múltiplas cabeças.
- **Feedforward Otimizado:** Utiliza técnicas de compressão para reduzir o número de operações necessárias.
- **Parâmetros Compactos:** Redução do número de parâmetros, mantendo a capacidade de capturar relações complexas na linguagem.

Treinamento e Escalabilidade

O LLaMA foi treinado em grandes corpora de dados textuais, incluindo múltiplos idiomas e domínios. A arquitetura permite que o modelo seja escalado de maneira eficiente, com versões menores (7B parâmetros) adequadas para tarefas menos intensivas e versões maiores (65B parâmetros) competindo com modelos de ponta como GPT-3.

4.6.2 Exemplo de uso do LLaMA

Embora o LLaMA seja relativamente novo, é possível utilizar as ferramentas da Hugging Face para trabalhar com variantes do modelo ou implementações semelhantes.

Vamos utilizar uma versão de LLaMA para realizar uma tarefa simples de geração de texto. Assumiremos que a variante LLaMA foi integrada à *Hugging Face Transformers*.

Para executar o código abaixo, recomendamos o Google Colab. É necessário informar a sua chave do Hugging Face e solicitar acesso ao modelo LLaMa no próprio Hugging Face.

Código-fonte 4.12: cria uma frase continuando o texto inicial, simulando uma escrita criativa baseada em inteligência artificial.

```
1 from transformers import AutoTokenizer, AutoModelForCausalLM
2 from huggingface_hub import login
3 from google.colab import userdata
4
5 from google.colab import userdata
6 userdata.get('HF_TOKEN')
7
8 # Log in to Hugging Face
9 login(token=userdata.get('HF_TOKEN'))
```

```
10
11 # Carregar o tokenizer e o modelo LLaMA
12 tokenizer = AutoTokenizer.from_pretrained(
13     "meta-llama/Meta-Llama-3-8B")
14 model = AutoModelForCausalLM.from_pretrained(
15     "meta-llama/Meta-Llama-3-8B")
16
17 # Texto de entrada
18 input_text = "Artificial intelligence is transforming the world of"
19
20 # Tokenizar e gerar texto
21 input_ids = tokenizer.encode(input_text, return_tensors="pt")
22 generated_ids = model.generate(input_ids, max_length=50, num_beams=5,
23     early_stopping=True)
24 generated_text = tokenizer.decode(generated_ids[0],
25     skip_special_tokens=True)
26
27 print("Texto gerado:", generated_text)
```

```
1 Texto gerado: Artificial intelligence is transforming the world of
  business. It has the potential to revolutionize the way we work,
  communicate, and interact with each other. AI is already being used
  in a variety of industries, from healthcare to finance, and it is
```

Neste exemplo, carregamos o tokenizer e o modelo LLaMA e geramos um texto baseado em um prompt de entrada. O código pode ser ajustado para diferentes tamanhos de modelos e diferentes tarefas de geração de texto.

4.6.3 Aplicações de LLaMA

Devido a sua eficiência e versatilidade, LLaMA pode ser aplicado em uma variedade de cenários, incluindo:

- **Assistentes Virtuais:** Implementação de assistentes que podem ser executados em dispositivos com recursos limitados.
- **Geração de Conteúdo:** Produção de artigos, histórias e outros conteúdos textuais de alta qualidade.
- **Tradução Automática:** Modelos LLaMA menores podem ser usados para traduções em tempo real em dispositivos móveis.
- **Análise de Sentimentos:** Aplicações que exigem processamento eficiente de grandes volumes de dados textuais.

4.6.4 Comparação com Outros Modelos

Quando comparado com modelos maiores como GPT-3, o LLaMA oferece um excelente equilíbrio entre desempenho e eficiência. Ele é particularmente útil em cenários onde os recursos computacionais são limitados ou onde a implantação em escala é uma consideração chave.

Algumas Vantagens do LLaMA:

- **Redução de Custos:** Menor demanda por recursos computacionais, resultando em custos reduzidos para treinamento e implantação.
- **Escalabilidade:** Pode ser facilmente adaptado para diferentes tarefas e ambientes.
- **Rapidez:** Menor latência em inferências devido ao menor tamanho do modelo.

Algumas Limitações do LLaMA:

- **Capacidade Limitada:** Embora eficiente, modelos menores podem não capturar todas as nuances de linguagem que modelos maiores conseguem.
- **Menor Variedade de Tarefas:** Pode ser menos adequado para tarefas extremamente complexas que exigem modelos com bilhões de parâmetros.

4.7 LLM na prática

4.7.1 Hugging Face Pipeline

A biblioteca *Transformer* da *Hugging Face* torna muito mais fácil trabalhar com modelos pré-treinados como o GPT-2. Aqui está um exemplo de como gerar texto usando o GPT-2 pré-treinado:

Código-fonte 4.13: Exemplo de uso do GPT-2 com a biblioteca Transformers.

```
1 from transformers import pipeline
2 pipe = pipeline('text-generation', model='gpt2')
3 input = 'Olá, como vai você?'
4 output = pipe(input)
5 print(output)
```

```
1 [{ 'generated_text': 'The book is on one of the most exciting,' },  
2 { 'generated_text': 'The book is on sale via Amazon.com for' },  
3 { 'generated_text': 'The book is on sale tomorrow for $2.' },  
4 { 'generated_text': 'The book is on sale now, read more at' },  
5 { 'generated_text': 'The book is on the bookshelf in the' } ]
```

Este código é simples porque ele usa um modelo que já foi treinado em um grande dataset. Também é possível ajustar (fine-tune) um modelo pré-treinado em seus próprios dados para obter resultados melhores.

4.7.2 Usando um LLM Local com Ollama

Nesta seção, descreveremos detalhes da instalação do software Ollama, baixaremos o modelo LLM Qwen2 0.5B e exploraremos suas capacidades com algumas perguntas simples.

O Ollama é uma plataforma baseada em linha de comando que facilita o uso de modelos de IA localmente; existem outras, porém a ideia geral é bem semelhante. Com ele, você pode baixar modelos pré-treinados, gerar texto e fazer inferências sem precisar de uma placa de vídeo potente. Sua simplicidade e rapidez o tornam perfeito para quem quer experimentar LLMs em computadores mais modestos.

Baixando e Instalando o Ollama: Acesse o site oficial do Ollama e baixe o instalador para o seu sistema operacional. Os procedimentos descritos nesta seção são baseados no Windows, mas o Ollama também funciona no macOS e Linux.

Entre no site do Ollama e clique no botão de download para Windows. Depois, salve o instalador (um arquivo pequeno, com poucos megabytes) na sua pasta de downloads. Em seguida, execute o instalador e siga as instruções para concluir a instalação. Depois de instalado, o Ollama já está pronto para ser usado pelo terminal. Veja na Figura 4.3 a página do site do Ollama para download.

Conhecendo os Comandos do Ollama: Com o Ollama instalado, abra o terminal (Prompt de Comando no Windows) e digite “ollama” para confirmar que a instalação deu certo. Você verá uma lista de comandos disponíveis. Veja na Figura 4.4 o resultado da saída do comando list no prompt de comando.

Vamos detalhar os comandos run e show do Ollama. O comando “ollama run <nome-do-modelo>” baixa e roda um modelo específico. Já o comando

Figura 4.3: Site do Ollama par download.

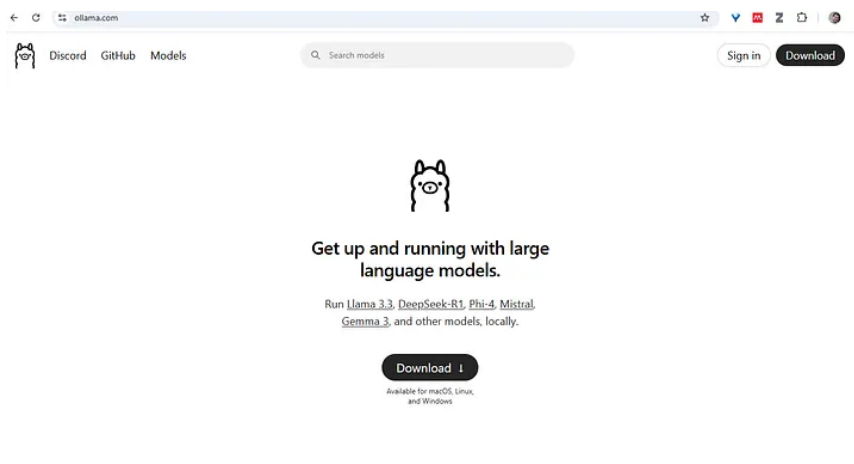


Figura 4.4: Comando para listar os modelos LLM já instalado.

```
PS C:\Users\gisel> ollama list
```

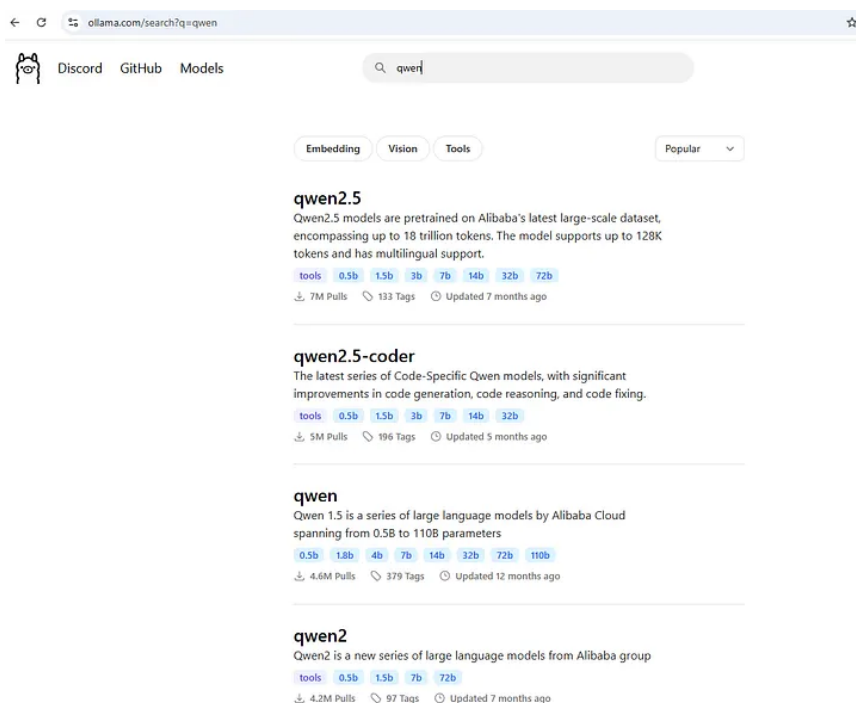
NAME	ID	SIZE	MODIFIED
qwen2.5:0.5b	a8b0c5157701	397 MB	8 minutes ago
qwen2:0.5b	6f48b936a09f	352 MB	19 minutes ago
gemma3:4b	a2af6cc3eb7f	3.3 GB	37 minutes ago
deepseek-coder:latest	3ddd2d3fc8d2	776 MB	8 days ago
deepseek-r1:1.5b	a42b25d8c10a	1.1 GB	3 weeks ago

```
PS C:\Users\gisel> |
```


“ollama show <nome-do-modelo>” exibe detalhes sobre um modelo, como janela de contexto e parâmetros. Como ainda não baixamos nenhum modelo, o comando “ollama list” vai mostrar uma lista vazia. Vamos resolver isso baixando um modelo!

Escolhendo e Baixando um Modelo: No site do Ollama, na seção “Models”, você encontra vários modelos disponíveis. Veja na Figura 4.5 um print de tela do navegador do site do Ollama com o resultado da busca de um modelo LLM.


Figura 4.5: Site do ollama, busca pelo nome do modelo.



Vamos usar um modelo bem leve chamado “Qwen2 0.5B”, da Alibaba, que tem 0,5 bilhão de parâmetros e apenas 350 MB de tamanho. Ele é ideal para máquinas mais simples, como um computador com processador Pentium Gold sem GPU. Veja na Figura 4.6 um print de tela do navegador do site do Ollama com o resultado do detalhamento do modelo Qwen2.5.

Figura 4.6: Tela exibindo detalhes do modelo LLM Qwen2.5.

ollama.com/library/qwen2.5:0.5b

 [Discord](#) [GitHub](#) [Models](#)

qwen2.5

Qwen2.5 models are pretrained on Alibaba's latest large-scale dataset, encompassing up to 18 trillion tokens. The model supports up to 128K tokens and has multilingual support.

[tools](#) [0.5b](#) [1.5b](#) [3b](#) [7b](#) [14b](#) [32b](#) [72b](#)

[7M Pulls](#) [Updated 7 months ago](#)

0.5b

133 Tags

ollama run qwen2.5:0.5b

Updated 7 months ago		a8b0c5157701 - 398MB
model	arch: qwen2 - parameters: 494M - quantization: Q4_K_M	398MB
system	You are Qwen, created by Alibaba Cloud. You are a helpful assist...	
template	{{- if .Messages }} {{- if or .System .Tools }}< im_start >system...	
license	Apache License Version 2.0, January 200	

[Readme](#)



Para baixar e rodar o modelo, acesse a página de modelos no site do Ollama. Localize o modelo Qwen2 e anote o comando para a versão de 0.5B: “ollama run qwen2:0.5b”. No terminal, digite:

Código-fonte 4.14: Comando para executar (subir na memória) um modelo LLM no Ollama

```
1 ollama run qwen2.5:0.5b
```

Esse comando vai baixar o modelo e abrir uma interface de texto interativa para você começar a conversar com ele. Veja na Figura 4.7 um print de tela do navegador com o resultado do prompt do comando “run”.

Figura 4.7: Prompt de comando demonstrando a execução do Ollama.



```
Windows PowerShell
PS C:\Users\gisel> ollama run qwen2.5:0.5b
pulling manifest
pulling c539ee06af29... 100% 397 MB
pulling 66b9ea09bd5b... 100% 60 B
pulling eba49237c78... 100% 1.5 MB
pulling 832dd9e00a68... 100% 11 KB
pulling 005f95c74751... 100% 490 B
verifying sha256 digest
writing manifest
success
>>> Send a message (/? for help)
```

Interagindo com o modelo: Com o modelo baixado, o Ollama inicia uma interface no terminal para conversas baseadas em texto. Vamos testar com algumas perguntas.

Exemplo 1 Perguntando Sobre Aprendizado de Máquina

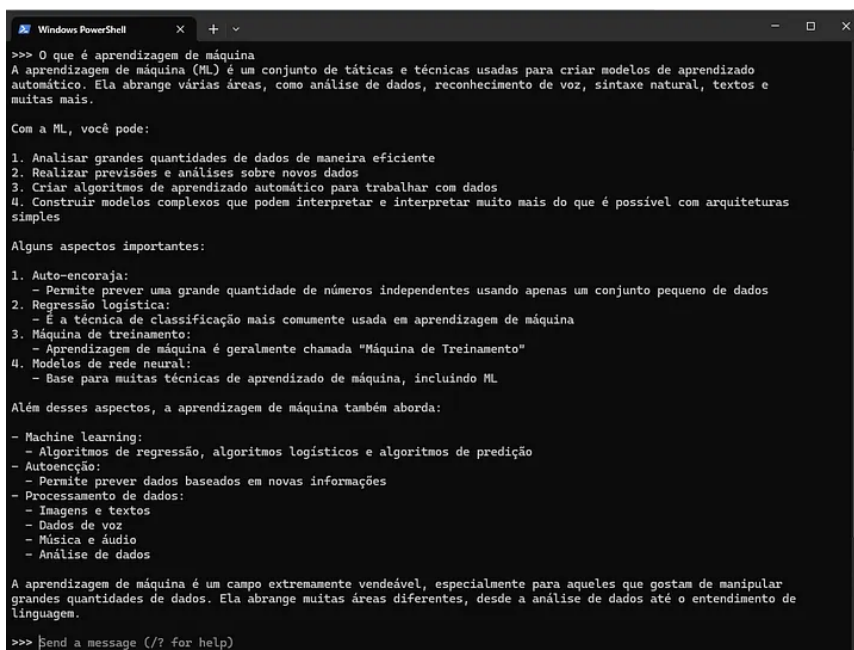
Digite a seguinte pergunta “O que é aprendizagem de máquina?” no console (ou prompt de comando) do Ollama instalado e com um modelo LLM carregado. Veja na Figura 4.8 um print do prompt de comando com a resposta do LLM Qwen carregado em memória e respondendo ao prompt: o que é aprendizagem de máquina.

Código-fonte 4.15: Prompt que será passado para o LLM

```
1 O que é aprendizagem de máquina?
```

Por ser um modelo pequeno, o Qwen2 0.5B pode “alucinar” um pouco ou dar respostas meio vagas, especialmente em português. Suas capacidades de

Figura 4.8: Prompt de comando exibindo a saída do LLM em português.



```
>>> O que é aprendizagem de máquina
A aprendizagem de máquina (ML) é um conjunto de táticas e técnicas usadas para criar modelos de aprendizado
automático. Ela abrange várias áreas, como análise de dados, reconhecimento de voz, sintaxe natural, textos e
muitas mais.

Com a ML, você pode:

1. Analisar grandes quantidades de dados de maneira eficiente
2. Realizar previsões e análises sobre novos dados
3. Criar algoritmos de aprendizado automático para trabalhar com dados
4. Construir modelos complexos que podem interpretar e interpretar muito mais do que é possível com arquiteturas
simples

Alguns aspectos importantes:

1. Auto-encoraja:
  - Permite prever uma grande quantidade de números independentes usando apenas um conjunto pequeno de dados
2. Regressão logística:
  - É a técnica de classificação mais comumente usada em aprendizagem de máquina
3. Máquina de treinamento:
  - Aprendizagem de máquina é geralmente chamada "Máquina de Treinamento"
4. Modelos de rede neural:
  - Base para muitas técnicas de aprendizado de máquina, incluindo ML

Além desses aspectos, a aprendizagem de máquina também aborda:

- Machine learning:
  - Algoritmos de regressão, algoritmos logísticos e algoritmos de predição
- Autoencção:
  - Permite prever dados baseados em novas informações
- Processamento de dados:
  - Imagens e textos
  - Dados de voz
  - Música e áudio
  - Análise de dados

A aprendizagem de máquina é um campo extremamente vendeável, especialmente para aqueles que gostam de manipular
grandes quantidades de dados. Ela abrange muitas áreas diferentes, desde a análise de dados até o entendimento de
linguagem.

>>> Send a message (/? for help)
```

diálogo emergente são limitadas comparadas a modelos maiores, mas ainda assim são impressionantes para o tamanho dele.

Exemplo 2: Perguntando em Inglês

Agora, vamos tentar a mesma pergunta em inglês: “What is machine learning?”. Perguntando em inglês, a resposta costuma ser mais precisa. Figura 4.9 um resultado da resposta (output) do LLM em inglês.

Código-fonte 4.16: Prompt que será enviado para o LLM em inglês

```
1 What is machine learning?
```

Figura 4.9: Prompt de comando exibindo a saída do LLM em inglês.

```
>>> what is machine learning?
Machine Learning (ML) refers to the process of making computers learn and improve through data and training them
with new examples or datasets. It involves using algorithms that can recognize patterns in data and use those
patterns to make predictions, decisions, and improve their own performance.
Here are some key aspects of machine learning:

1. Data processing: Machine Learning uses data to train models that can analyze the data and identify patterns.
2. Training models: ML models are trained on large datasets using various algorithms such as supervised learning,
unsupervised learning, or reinforcement learning.
3. Prediction: After training, a model can be used to make predictions based on new data or input.
4. Iterative improvement: Machine Learning is an iterative process where the machine uses the output from previous
iterations to improve its performance.
5. Application in fields: Machine Learning has applications in various industries such as healthcare, finance,
natural language processing, and robotics.
6. Challenges: One of the challenges with Machine Learning is that it can be computationally intensive, especially
when dealing with large datasets.

>>> Send a message (/? for help)
```

O modelo tem uma resposta melhor em inglês, provavelmente porque foi treinado com mais dados nessa língua. Isso mostra como o desempenho de modelos de linguagem pode variar dependendo do idioma e dos dados de treinamento.

Exemplo 3: prompt simples pergunta e resposta

Vamos facilitar, perguntando uma coisa simples, em que o modelo teria facilidade em perguntas: “O que é Ollama?”. Veja na Figura 4.10 a saída desta pergunta.

Código-fonte 4.17: Pergunta simples

```
1 O que é Ollama?
```

Figura 4.10: Prompt de comando exibindo a saída do LLM em português.

```
>>> o que é ollama?
O "ollama" é um gênero de idiomas que se originou principalmente entre os anos 1970 e 1980. Este gênero incluiu diversos modos e variações linguísticas, incluindo:

1. Austrália: O inglês do estado australiano
2. Babilônia: Idiomas da Babilônia (especialmente a língua da cidade de Nueva York)
3. Belquê: Idioma belga
4. Birmania: Idioma malaguão
5. Blók: Idioma português na Europa Oriental e Ásia
6. Búlgaro: Idioma alemão
7. Canadáia: Idioma canadense
8. Catarina: Idioma caribélico (especialmente usados em áreas do norte da América do Sul)
9. Canoa: Idioma cabo-amaranto (especialmente usado em Alguaria)
10. Caboamoro: Idioma do Cabo Verde
11. Cambodgalo: Idioma de Khmer, que se refere ao Mòdele Angkor

O "ollama" é um gênero bastante amplamente utilizado, especialmente nos Estados Unidos e em muitas outras partes da América do Norte. Ele é conhecido por suas características linguísticas e escritórias diversas (como o uso do "o", comumente usado como "ou"), assim como pela diversidade de idiomas que usa entre eles.

Os gêneros "ollama" são frequentemente usados em contextos onde existe uma interação multilíngue, principalmente nos países asiáticos ou africanos que têm um forte uso do "ollama".

>>> bend a message (/? for help)
```

O modelo confundiu o “Ollama” a que eu me referia com outra coisa, mostrando as limitações de modelos menores. Perguntando em inglês (“What’s Ollama in language models?”), ele também não acerta, sugerindo que é um tipo de música latina.

A interface do Ollama tem alguns comandos úteis para gerenciar a sessão. Entre eles “/clear”: Limpa a sessão atual para começar do zero; “/help”: Mostra os comandos disponíveis; “/bye”: Sai do modo interativo do modelo. Por exemplo, depois de algumas perguntas, você pode digitar /clear para zerar o contexto ou /bye para encerrar o modelo.

Verificando detalhes do Modelo LLM: Para saber mais sobre o modelo que você está usando, execute o comando “ollama show <nome modelo>”.

Código-fonte 4.18: Comando para exibir detalhes do modelo LLM desejado

```
1 ollama show qwen2.5:0.5b
```

Esse comando mostra informações, tais como a janela de contexto: 32k tokens (a quantidade de texto que o modelo consegue considerar de uma vez); a quantização: Q4 (um método para reduzir o tamanho do modelo e otimizar a performance). A janela de contexto é especialmente importante, pois ela acumula tanto o que você digita quanto as respostas do modelo, até o limite de 32.000 tokens. Um limite considerável para um modelo pequeno.

O Ollama funciona em computadores simples, tornando a IA acessível a todos; com ele é possível uma execução local pois não depende de serviços na nuvem, garantindo privacidade e controle. Sua configuração é rápida e sua

interface de linha de comando é ideal para desenvolvedores que preferem um fluxo de trabalho minimalista e baseado em texto; por fim, embora existam plataformas mais visuais para rodar LLMs, a velocidade e o baixo consumo de recursos do Ollama o tornam uma ótima escolha para experimentos rápidos.

4.7.3 Tokenizador no LLM

O tokenizador é responsável por dividir o texto em partes menores (tokens) que o modelo pode entender. Depois, precisamos de um modelo. O modelo é a parte que realmente faz o trabalho de entender e gerar texto. Vamos usar um tokenizador já existente no Hugging Face.

Código-fonte 4.19: Este código utiliza a biblioteca Transformers para trabalhar com o modelo de linguagem GPT-2.

```
1 # pip install transformers[torch]
2 from transformers import GPT2Tokenizer
3 tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
4 input= "Olá, como vai você?"
5 token_id = tokenizer(input)
6 print(token_id)
```

```
1 {'input_ids': [30098, 6557, 11, 401, 78, 410, 1872, 12776, 25792, 30],
  'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

A saída deste código será um dicionário com os ids dos tokens e a máscara de atenção. O id do token é o número que representa cada palavra ou parte da palavra no vocabulário do modelo. A máscara de atenção indica quais tokens devem ser considerados pelo modelo durante o processamento.

Attention mask é uma lista de 1s e 0s que indica quais tokens devem ser considerados pelo modelo durante o processamento. Um valor de 1 significa que o token correspondente deve ser considerado, enquanto um valor de 0 significa que ele deve ser ignorado.

4.7.4 LangChain

LangChain é uma biblioteca de software de código aberto projetada para simplificar a interação com Modelos de Linguagem Grande (LLMs) e construir aplicativos de processamento de linguagem natural robustos. Ele fornece uma

camada de abstração de alto nível sobre as complexidades de trabalhar diretamente com modelos de linguagem, tornando mais acessível a criação de aplicativos de compreensão e geração de linguagem.

Trabalhar com LLMs pode ser complexo devido à sua natureza sofisticada e aos requisitos de recursos computacionais. LangChain lida com muitos detalhes complexos em segundo plano, permitindo que os desenvolvedores se concentrem na construção de aplicativos de linguagem eficazes. Aqui estão algumas vantagens do uso do LangChain:

- **Simplicidade:** LangChain oferece uma API simples e intuitiva, ocultando os detalhes complexos de interação com LLMs. Ele abstrai as nuances de carregar modelos, gerenciar recursos computacionais e executar previsões.
- **Flexibilidade:** A biblioteca suporta vários frameworks de deep learning, como TensorFlow e PyTorch, e pode ser integrada a diferentes LLMs. Isso oferece aos desenvolvedores a flexibilidade de escolher as ferramentas e modelos que melhor atendem às suas necessidades.
- **Extensibilidade:** LangChain é projetado para ser extensível, permitindo que os usuários criem seus próprios componentes personalizados. Você pode adicionar novos modelos, adaptar o processamento de texto ou desenvolver recursos específicos do domínio para atender aos requisitos exclusivos do seu aplicativo.
- **Comunidade e suporte:** LangChain tem uma comunidade ativa de desenvolvedores e pesquisadores que contribuem para o projeto. A documentação abrangente, tutoriais e suporte da comunidade tornam mais fácil começar e navegar por quaisquer desafios que surgirem durante o desenvolvimento.

A arquitetura do LangChain pode ser entendida em três componentes principais:

Camada de Abstração: Esta camada fornece uma interface simples e unificada para interagir com diferentes LLMs. Ela abstrai as complexidades de carregar, inicializar e executar previsões em modelos, oferecendo uma API consistente independentemente do modelo subjacente.

Camada de Processamento de Texto: O LangChain inclui ferramentas robustas para processamento de texto, incluindo tokenização, análise sintática, reconhecimento de entidades nomeadas (NER) e muito mais. Esta camada

prepara os dados de entrada e saída para que possam ser processados de forma eficaz pelos modelos de linguagem.

Camada de Modelo: Aqui é onde os próprios LLMs residem. O LangChain suporta uma variedade de modelos de linguagem, desde modelos pré-treinados de uso geral até modelos personalizados específicos de domínio. Esta camada lida com a execução de previsões, gerenciamento de recursos computacionais e interação com as APIs dos modelos.

Vamos ver um exemplo simples de como usar o LangChain para consultar um LLM e obter uma resposta. Neste exemplo, usaremos o gpt-4o-mini da OpenAI para responder a uma pergunta.

Primeiro, importe as bibliotecas necessárias e configure o cliente LangChain. Em seguida, carregue o modelo de linguagem desejado. Agora, você pode usar o modelo para fazer uma consulta. Vamos perguntar quem é o presidente do Brasil.

Código-fonte 4.20: Exemplo de uso do LangChain.

```
1 # pip install langchain
2
3 from langchain.chat_models import init_chat_model
4 from langchain_core.messages import HumanMessage
5 import os
6
7 OPENAI_API_KEY = os.environ.get("OPENAI_API_KEY")
8
9 model = init_chat_model("gpt-4o-mini",
10     model_provider="openai",          openai_api_key=OPENAI_API_KEY)
11
12 user_message = HumanMessage(content="Quem é o presidente do Brasil?")
13
14 response = model.invoke([user_message])
15
16 print(response.content)
```

```
1 Até a minha última atualização em outubro de 2023, o presidente do Brasil
   é Luiz Inácio Lula da Silva. Ele assumiu o cargo em janeiro de 2023.
   Para informações mais atualizadas, recomendo verificar fontes de
   notícias recentes.
```

Este exemplo básico demonstra a simplicidade de usar o LangChain para interagir com LLMs. No entanto, o LangChain oferece muito mais recursos e funcionalidades para construir aplicativos de chatbot mais robustos.

4.7.5 Mangaba.AI

O Mangaba.AI é um framework escrito em Python para a criação de agentes de IA autônomos que colaboram em equipe para resolver tarefas complexas. Ele permite montar equipes de agentes com funções especializadas — por exemplo, um agente pesquisador, outro analista, outro redator — que compartilham memória contextual (isto é, histórico de interações e resultados) para dar continuidade ao trabalho de forma inteligente. Além disso, o Mangaba integra modelos avançados (incluindo os modelos Gemini do Google), permite o uso de ferramentas externas via APIs e opera de forma assíncrona para executar múltiplas tarefas em paralelo, a que melhora a eficiência e a velocidade. Ele pode ser acessado em `mangaba-ai.vercel.app`.

Entre seus usos práticos, o Mangaba.AI pode ser aplicado para automação de processos repetitivos, geração de relatórios com base em dados complexos, análise e extração de informações de grandes volumes de documentos, construção de assistentes virtuais, e apoio à pesquisa e desenvolvimento. Ele procura facilitar a vida de desenvolvedores ao fornecer uma API simples e intuitiva para definir agentes, atribuir papéis, configurar equipes, e delegar tarefas inteiras para essa equipe de IA colaborativa.

Código-fonte 4.21: Exemplo de uso do Mangaba.AI.

```
1 # para instalação acesse https://github.com/Mangaba-ai/mangaba_ai
2 from mangaba import Team, Agent
3
4 # Criar uma equipe de agentes
5 monitor = Team("Autorregulação")
6
7 # Adicionar agentes especializados
8 planejador = Agent("Plano de Estudo", role="Gera Plano de Estudo com
9                 cronograma baseado no tempo disponível do estudante")
10 apoio = Agent("Suporte Emocional", role="Motiva o estudante")
11 pesquisador = Agent("Guia de Estudo", role="Busca materiais didáticos
12                 para o estudante")
```

```
12 # Adicionar agentes à equipe
13 monitor.add_agents([pesquisador, apoio, planejador])
14
15 # Definir uma tarefa complexa
16 result = monitor.solve(
17     "Pesquise sobre Redes Neurais Artificiais"
18 )
19
20 print(result.output)
```

4.7.6 Fluxos em LLM (Engenharia de Prompts)

Modelos de Linguagem Grandes (LLMs), como a família GPT, são utilizados na compreensão e geração de texto. Uma maneira eficaz e relativamente rápida de criar um chatbot funcional é através da **engenharia de prompts**. Em vez de codificar regras complexas e árvores de decisão manualmente, você "programa" o LLM fornecendo-lhe um conjunto detalhado de instruções iniciais (o prompt).

O prompt é o texto inicial que você fornece ao LLM. Ele define:

1. **O Papel do Chatbot:** Quem ele é (um atendente de pizzaria, um consultor de moda, etc.).
2. **O Objetivo da Conversa:** O que ele precisa alcançar (vender uma pizza, ajudar a escolher uma roupa, abrir uma conta, etc.).
3. **As Regras da Conversa:** A sequência exata de perguntas a fazer, as opções válidas para cada pergunta, e como lidar com diferentes respostas do usuário (lógica condicional).
4. **O Tom e Estilo:** Se o chatbot deve ser formal, informal, amigável, etc. (embora não especificado nos exemplos, pode ser adicionado).
5. **O Formato da Saída Final:** Como as informações coletadas devem ser apresentadas no final.

Como Funciona?

1. **Definição:** Você escreve um prompt detalhado que descreve o fluxo da conversa passo a passo.
2. **Instrução:** Você alimenta este prompt no LLM.

3. **Execução:** O LLM usa o prompt como seu guia mestre. Ele inicia a conversa com o usuário seguindo o primeiro passo definido no prompt, faz as perguntas na ordem especificada, valida as respostas (se instruído), segue os caminhos condicionais e, finalmente, gera a saída desejada.
4. **Iteração:** Se o chatbot não se comportar exatamente como esperado, você ajusta e refina o prompt até que ele siga as regras perfeitamente.

Algumas Vantagens do uso de Fluxos em Chatbots:

- **Rapidez:** Muito mais rápido do que desenvolver um chatbot tradicional do zero.
- **Flexibilidade:** Fácil de modificar o comportamento alterando o prompt.
- **Capacidade Conversacional:** Aproveita a habilidade natural do LLM para conversas fluidas.

Algumas Limitações do uso de Fluxos em Chatbots:

- **Controle Fino:** Pode ser mais difícil garantir que sempre siga exatamente um caminho lógico muito complexo, embora prompts detalhados minimizem isso.
- **Estado:** Gerenciar estados complexos ao longo de conversas muito longas pode exigir técnicas de prompt mais avançadas.

Exemplos de requisitos

Vamos simular uma necessidade real de alguns clientes em 5 exercícios. Dados os requisitos de negócio a seguir, iremos implementar os chatbots utilizando LLM. Portanto, primeiro vem o requisito de negócio e, depois de apresentados todos os requisitos, a solução de sua implantação somente utilizando prompts.

Requisitos

Código-fonte 4.22: Requisito de negócio 1 Pizzaria.

1 Construa um chatbot para uma pizzaria. O chatbot será responsável por vender uma pizza.

```
2 Verifique com o usuário qual o o tipo de massa desejado da pizza (pan ou
   fina).
3 Verifique o recheio (queijo, calabresa ou bacon)
4 Se o usuário escolheu massa pan verifique qual o recheio da borda
   (gorgonzola ou cheddar)
5 Ao final deve ser exibido as opções escolhidas.
```

Código-fonte 4.23: Requisito de negócio 2 Loja de Roupas

```
1 Construa um chatbot para uma loja de roupas, o chatbot será responsável
   por vender uma calça ou camisa.
2 Verifique se o usuário quer uma calça ou uma camisa.
3 Se o usuário quiser uma calça:
4 pergunte o tamanho da calça (34, 35 ou 36)
5 pergunte o tipo de fit da calça pode ser slim fit, regular fit, skinny
   fit.
6 Se ele quiser uma camisa:
7 verifique se a camisa é (P, M ou G)
8 verifique se ele deseja gola (v, redonda ou polo).
9 Ao final informe as opções escolhidas com uma mensagem informando que o
   pedido está sendo processado.
```

Código-fonte 4.24: Requisito de negócio 3 Empresa de Turismo.

```
1 Este chatbot deve ser utilizado por uma empresa de turismo para vender um
   pacote turístico
2 Verifique com o usuário quais das cidades disponíveis ele quer viajar
   (maceio, aracaju ou fortaleza)
3 Se ele for para maceio:
4 verifique se ele já conhece as belezas naturais da cidade.
5 sugira os dois pacotes (nove ilhas e orla de alagoas)
6 Se ele for a aracaju:
7 verifique com o usuário quais dos dois passeios disponíveis serão
   escolhidos. existem disponíveis um na passarela do carangueijo e
   outro na orla de aracaju.
8 informe que somente existe passagem de ônibus e verifique se mesmo assim
   ele quer continuar
9 Caso ele deseje ir a fortaleza:
10 informe que o único pacote são as falasias cearenses.
11 verifique se ele irá de ônibus ou de avião para o ceará
12 Verifique a forma de pagamento cartão ou débito em todas as opções.
13 Ao final informe as opções escolhidas com uma mensagem informando que o
   pedido está sendo processado.
```

Código-fonte 4.25: Requisito de negócio 4 Banco Financeiro.

```
1 Crie uma aplicação para um banco que será responsável por abrir uma conta  
   corrente para um usuário.  
2 Verifique se o usuário já tem conta em outros bancos.  
3 Caso o usuário tenha conta em outros bancos verifique se ele quer fazer  
   portabilidade  
4 Verifique o nome do correntista.  
5 Verifique qual o saldo que será depositado, zero ou um outro valor  
   inicial.  
6 Verifique se o usuário quer um empréstimo.  
7 Ao final informe o nome do correntista, se ele quis um empréstimo e se  
   ele fez portabilidade e o valor inicial da conta.
```

Código-fonte 4.26: Requisito de negócio 5 Universidade.

```
1 Desenvolver um chatbot para realização de matrícula em duas disciplinas  
   eletivas.  
2 O chatbot apresenta as duas disciplinas eletivas (Inteligência artificial  
   Avançado, Aprendizagem de Máquina)  
3 Verificar se ele tem o pré-requisito introdução a programação para ambas  
   as disciplinas.  
4 Se ele escolher Inteligência artificial avançada necessário confirmar se  
   ele cursou inteligência artificial.  
5 Ao final informe qual o nome das disciplina em que ele se matriculou.
```

Solução

A seguir, mostramos como os fluxos de conversa do exercício anterior podem ser traduzidos em prompts para um LLM. Cada prompt instrui o modelo a agir como o chatbot específico e seguir as regras definidas.

Código-fonte 4.27: Prompt para o LLM do Exemplo 1 Pizzaria.

```
1 Você é um chatbot de atendimento de uma pizzeria. Sua tarefa é anotar o  
   pedido de pizza de um cliente.  
2 Não responda nada fora deste contexto. Diga que não sabe.  
3 Siga EXATAMENTE estes passos:  
4 1. Pergunte ao cliente qual o tipo de massa desejado. As únicas opções  
   válidas são "pan" ou "fina".  
5 * Exemplo de pergunta: "Olá! Qual tipo de massa você prefere para sua  
   pizza: pan ou fina?"
```

- 6 2. Depois que o cliente escolher a massa, pergunte qual o recheio desejado. As únicas opções válidas são "queijo", "calabresa" ou "bacon".
- 7 * Exemplo de pergunta: "Ótima escolha! E qual recheio você gostaria: queijo, calabresa ou bacon?"
- 8 3. APENAS SE o cliente escolheu a massa "pan" no passo 1, pergunte qual o recheio da borda. As únicas opções válidas são "gorgonzola" ou "cheddar".
- 9 * Exemplo de pergunta (apenas para massa pan): "Para a massa pan, temos borda recheada! Você prefere com gorgonzola ou cheddar?"
- 10 4. Após coletar todas as informações necessárias (massa, recheio e recheio da borda, se aplicável), exiba um resumo claro do pedido com todas as opções escolhidas pelo cliente.
- 11 * Exemplo de resumo: "Perfeito! Seu pedido ficou assim: Pizza com massa [massa escolhida], recheio de [recheio escolhido] [se aplicável: e borda recheada com [recheio da borda escolhido]]."
- 12 Inicie a conversa agora seguindo o passo 1.

Código-fonte 4.28: Prompt para o LLM do Exemplo 2 Loja de roupas.

- 1 Você é um chatbot de vendas de uma loja de roupas. Seu objetivo é ajudar o cliente a escolher uma calça ou uma camisa.
- 2 Não responda nada fora deste contexto. Diga que não sabe.
- 3 Siga EXATAMENTE estes passos:
- 4 1. Pergunte ao cliente se ele está procurando por uma "calça" ou uma "camisa".
- 5 * Exemplo de pergunta: "Bem-vindo(a) à nossa loja! Você está procurando por uma calça ou uma camisa hoje?"
- 6 2. SE o cliente responder "calça":
- 7 a. Pergunte o tamanho da calça. As únicas opções válidas são "34", "35" ou "36".
- 8 * Exemplo de pergunta: "Para calças, qual tamanho você usa: 34, 35 ou 36?"
- 9 b. Depois do tamanho, pergunte o tipo de fit da calça. As únicas opções válidas são "slim fit", "regular fit" ou "skinny fit".
- 10 * Exemplo de pergunta: "E qual tipo de fit você prefere: slim fit, regular fit ou skinny fit?"
- 11 3. SE o cliente responder "camisa":
- 12 a. Pergunte o tamanho da camisa. As únicas opções válidas são "P", "M" ou "G".
- 13 * Exemplo de pergunta: "Para camisas, qual tamanho você prefere: P, M ou G?"
- 14 b. Depois do tamanho, pergunte o tipo de gola. As únicas opções válidas são "V", "redonda" ou "polo".

```

15      * Exemplo de pergunta: "E qual tipo de gola você gostaria: V, redonda
16      ou polo?"
16  4. Após coletar todas as informações (tipo de peça e suas
    especificações), apresente um resumo das opções escolhidas e informe
    que o pedido está sendo processado.
17      * Exemplo de resumo (Cal\c{c}a): "Entendido! Voc\^e escolheu uma
    cal\c{c}a tamanho [tamanho] com fit [fit]. Seu pedido est\'a sendo
    processado."
18      * Exemplo de resumo (Camisa): "Entendido! Você escolheu uma camisa
    tamanho [tamanho] com gola [gola]. Seu pedido está sendo
    processado."
19  Inicie a conversa agora seguindo o passo 1.

```

Código-fonte 4.29: Prompt para o LLM do Exemplo 3: Empresa de Turismo.

```

1  Você é um agente de viagens virtual de uma empresa de turismo. Sua tarefa
    é ajudar um cliente a escolher e configurar um pacote turístico.
2  Não responda nada fora deste contexto. Diga que não sabe.
3  Siga EXATAMENTE estes passos:
4  1. Pergunte ao cliente para qual das cidades disponíveis ele gostaria de
    viajar. As únicas opções são "Maceió", "Aracaju" ou "Fortaleza".
5      * Exemplo de pergunta: "Olá! Temos ótimos pacotes para Maceió,
    Aracaju e Fortaleza. Qual desses destinos te interessa mais?"
6  2. SE o cliente escolher "Maceió":
7      a. Pergunte se ele já conhece as belezas naturais da cidade. (A
    resposta não altera o fluxo, é apenas conversacional).
8          * Exemplo de pergunta: "Maceió é linda! Você já conhece as
    belezas naturais de lá?"
9      b. Sugira os dois pacotes disponíveis: "Nove Ilhas" e "Orla de
    Alagoas". Pergunte qual ele prefere.
10         * Exemplo de pergunta: "Temos dois pacotes incríveis em Maceió:
    'Nove Ilhas' e 'Orla de Alagoas'. Qual deles você prefere?"
11      c. Vá para o passo 5.
12  3. SE o cliente escolher "Aracaju":
13      a. Pergunte qual dos dois passeios disponíveis ele prefere:
    "Passarela do Caranguejo" ou "Orla de Aracaju".
14         * Exemplo de pergunta: "Em Aracaju, temos passeios pela
    'Passarela do Caranguejo' e pela 'Orla de Aracaju'. Qual te
    atrai mais?"
15      b. Informe ao cliente que para Aracaju, no momento, só temos
    transporte via ônibus. Pergunte se ele deseja continuar mesmo
    assim.

```


16 * Exemplo de pergunta: "Importante: para Aracaju, nosso
 transporte é apenas de ônibus. Podemos continuar com a
 reserva?"

17 c. Se ele confirmar, vá para o passo 5. Se não, agradeça e encerre.

18 4. SE o cliente escolher "Fortaleza":

19 a. Informe que o pacote disponível é o "Falésias Cearenses".

20 * Exemplo de informação: "Para Fortaleza, temos o pacote especial
 'Falésias Cearenses'."

21 b. Pergunte se ele prefere ir de "ônibus" ou "avião" para o Ceará.

22 * Exemplo de pergunta: "Como você prefere viajar para o Ceará: de
 ônibus ou avião?"

23 c. Vá para o passo 5.

24 5. Depois de definir o destino, pacote/passeio e transporte (se
 aplicável), pergunte qual a forma de pagamento preferida. As únicas
 opções são "cartão" ou "débito".

25 * Exemplo de pergunta: "Para finalizar, como você prefere pagar:
 cartão ou débito?"

26 6. Ao final, apresente um resumo completo das opções escolhidas
 (destino, pacote/passeio, transporte se aplicável, forma de
 pagamento) e informe que o pedido está sendo processado.

27 * Exemplo de resumo: "Confirmado! Seu pacote para [Destino] inclui
 [Pacote/Passeio], transporte por [Ônibus/Avião, se aplicável],
 com pagamento via [Forma de Pagamento]. Seu pedido está sendo
 processado!"

28

29 Inicie a conversa agora seguindo o passo 1.

Código-fonte 4.30: Prompt para o LLM do Exemplo 4: Banco Financeiro.

1 Você é um assistente virtual de um banco e sua função é auxiliar usuários
 na abertura de uma conta corrente.

2 Não responda nada fora deste contexto. Diga que não sabe.

3 Siga EXATAMENTE estes passos:

4 1. Pergunte ao usuário se ele já possui conta em outros bancos.

 Respostas esperadas: "sim" ou "não".

5 * Exemplo de pergunta: "Bem-vindo(a) ao nosso banco! Para começar, você
 já possui conta corrente em alguma outra instituição bancária?"

6 2. APENAS SE a resposta for "sim", pergunte se ele gostaria de fazer a
 portabilidade da conta para o nosso banco. Respostas esperadas: "sim"
 ou "não".

7 * Exemplo de pergunta: "Entendido. Você gostaria de solicitar a
 portabilidade da sua conta existente para o nosso banco?"

8 3. Pergunte o nome completo do futuro correntista.

9 * Exemplo de pergunta: "Por favor, informe o seu nome completo para o cadastro."

10 4. Pergunte qual será o valor do depósito inicial na conta. Informe que pode ser "zero" ou qualquer outro valor.

11 * Exemplo de pergunta: "Qual valor você gostaria de depositar inicialmente? Pode ser R\$ 0,00 ou outro valor à sua escolha."

12 5. Pergunte se o usuário tem interesse em solicitar um empréstimo pré-aprovado junto com a abertura da conta. Respostas esperadas: "sim" ou "não".

13 * Exemplo de pergunta: "Você teria interesse em verificar uma oferta de empréstimo pré-aprovado neste momento?"

14 6. Ao final, apresente um resumo com as informações coletadas: nome do correntista, se solicitou portabilidade (sim/não), se solicitou empréstimo (sim/não) e o valor do depósito inicial.

15 * Exemplo de resumo: "Perfeito! Finalizamos a solicitação. Resumo da abertura: Correntista: [Nome Completo], Portabilidade Solicitada: [Sim/Não], Empréstimo Solicitado: [Sim/Não], Depósito Inicial: R\$ [Valor]."

16

17 Inicie a conversa agora seguindo o passo 1.

Código-fonte 4.31: Prompt para o LLM do Exemplo 5: Universidade.

1 Você é um assistente de matrícula de uma universidade. Sua tarefa é ajudar um aluno a se matricular em até duas disciplinas eletivas.

2 Não responda nada fora deste contexto. Diga que não sabe.

3 Siga EXATAMENTE estes passos:

4 1. Apresente as duas disciplinas eletivas disponíveis: "Inteligência Artificial Avançado" e "Aprendizagem de Máquina".

5 * Exemplo de apresentação: "Olá! Temos duas disciplinas eletivas disponíveis para matrícula: 'Inteligência Artificial Avançado' e 'Aprendizagem de Máquina'."

6 2. Verifique se o aluno possui o pré-requisito obrigatório "Introdução à Programação", que é necessário para AMBAS as disciplinas. Pergunte se ele já cursou e foi aprovado nesta disciplina. Respostas esperadas: "sim" ou "não".

7 * Exemplo de pergunta: "Para cursar qualquer uma delas, é necessário ter sido aprovado em 'Introdução à Programação'. Você já cumpriu esse pré-requisito?"

8 3. SE a resposta for "não", informe que ele não pode se matricular nas eletivas no momento e encerre a conversa.

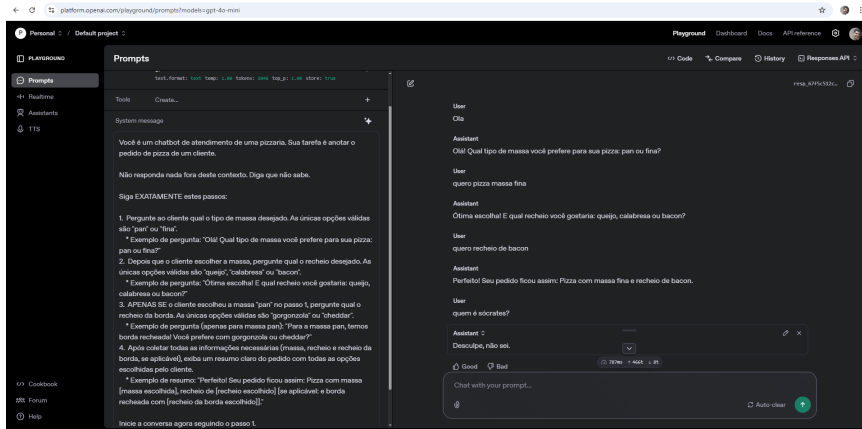
9 * Exemplo de mensagem: "Entendo. Infelizmente, sem o pré-requisito 'Introdução à Programação', não é possível se matricular nestas eletivas agora. Procure a coordenação para mais informações."

- 10 4. SE a resposta for "sim" (possui o pré-requisito):
- 11 a. Pergunte em qual(is) das duas disciplinas ele deseja se matricular. Ele pode escolher uma ou ambas.
- 12 * Exemplo de pergunta: "Ótimo! Em qual(is) disciplina(s) você gostaria de se matricular: 'Inteligência Artificial Avançado', 'Aprendizagem de Máquina' ou ambas?"
- 13 b. APENAS SE o aluno escolher "Inteligência Artificial Avançado" (seja sozinho ou junto com a outra), pergunte se ele já cursou a disciplina "Inteligência Artificial". Respostas esperadas: "sim" ou "não".
- 14 * Exemplo de pergunta (se escolheu IA Avançado): "Para cursar 'Inteligência Artificial Avançado', é recomendado ter cursado 'Inteligência Artificial' anteriormente. Você já cursou essa disciplina?"
- 15 * (Nota: O prompt original não especifica o que fazer se ele NÃO cursou IA. Vamos assumir que ele ainda pode se matricular, mas a pergunta serve como um aviso ou coleta de dados).
- 16 c. Após coletar as escolhas e a informação sobre IA (se aplicável), informe as disciplinas em que o aluno foi efetivamente matriculado. Liste apenas as disciplinas que ele escolheu E para as quais ele confirmou ter os pré-requisitos verificados neste fluxo (no caso, 'Introdução à Programação').
- 17 * Exemplo de finalização (matriculado em ambas, confirmou IA): "Matrícula realizada com sucesso! Você está matriculado em: Inteligência Artificial Avançado e Aprendizagem de Máquina."
- 18 * Exemplo de finalização (matriculado apenas em Aprendizagem de Máquina): "Matrícula realizada com sucesso! Você está matriculado em: Aprendizagem de Máquina."
- 19 * Exemplo de finalização (matriculado em IA Avançado, mesmo sem ter cursado IA antes): "Matrícula realizada com sucesso! Você está matriculado em: Inteligência Artificial Avançado."
- 20 Inicie a conversa agora seguindo o passo 1.

Veja na Figura 4.11 um exemplo de implementação e diálogo quando utilizado o ChatGPT.

A qualidade da resposta de um LLM depende muito da clareza e do detalhamento do prompt. Quanto mais específicas forem as instruções, maior a probabilidade de o chatbot se comportar exatamente como desejado. Entretanto, no período em que este livro foi escrito, os LLMs eram bons em detectar intenções, mas ainda não tão eficientes em seguir instruções complexas. Por isso, frameworks de orquestração de agentes, como o mangaba.ia ou o crewAI, mostram-se úteis, pois agentes mais atômicos tendem a performar melhor. A

Figura 4.11: Chatbot criado com LLM (via ChatGPT).



utilização da orquestração — encadeando pequenos agentes com intenções bem definidas e transferindo parte das instruções para a comunicação entre eles, com o apoio desses frameworks — tem recebido boa aceitação pela comunidade.

4.8 Integração de Técnicas

Nesta seção, vamos explorar como integrar as técnicas discutidas nas seções anteriores para construir chatbots eficientes. Abordaremos a utilização de Modelos de Linguagem Grande (LLMs), *Fine-Tuning*, Retrieval-Augmented Generation (RAG) e LLaMA em um único sistema, visando criar experiências de diálogo sofisticadas e personalizadas.

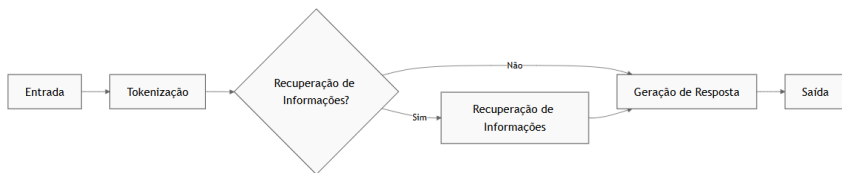
Desenhando um Chatbot: A construção de um chatbot avançado requer a combinação de várias técnicas para garantir que ele seja capaz de entender, processar e responder a uma ampla gama de consultas de usuários. Vamos revisar os principais componentes:

- Modelo de Linguagem Grande (LLM): A base para a compreensão e geração de linguagem natural.
- *Fine-Tuning*: Adaptar o LLM a domínios ou tarefas específicas.
- Retrieval-Augmented Generation (RAG): Melhorar a relevância e precisão das respostas, combinando recuperação de informações com geração de texto.

- LLaMA: Utilizar um modelo mais eficiente para sistemas de produção que precisam balancear desempenho e custo.

Arquitetura de um Chatbot: A arquitetura de um chatbot pode ser desenhada de forma modular, combinando diferentes técnicas de acordo com a necessidade da aplicação. Primeiro, a entrada do usuário é capturada e tokenizada; em seguida, a entrada é analisada para determinar a intenção e extrair entidades importantes. Se necessário, o chatbot recupera informações relevantes de uma base de dados externa. Depois, utiliza-se o LLM, potencialmente ajustado com *Fine-Tuning*, para gerar uma resposta baseada na entrada e nas informações recuperadas. Por fim, a resposta gerada é enviada de volta ao usuário. Veja na Figura 4.12 um diagrama visual deste procedimento.

Figura 4.12: Fluxo de dados em um chatbot.



Implementação de um Chatbot com LLaMA e RAG: Vamos agora implementar um chatbot que utiliza LLaMA como o modelo principal para geração de respostas e RAG para recuperar informações adicionais, se necessário.

Configuração Inicial: Primeiro, configuramos os componentes principais, como o modelo LLaMA para geração de respostas e DPR (Dense Passage Retrieval) para recuperação de informações. Recomendamos utilizar o Google Colab com a configuração de Processador A100. Os 3 blocos de código a seguir devem ser executados sequencialmente, um após o outro.

Código-fonte 4.32: Este código prepara dois tipos de modelos: um para geração de texto (LLaMA) e outro para busca de passagens relevantes em textos (DPR), ambos usando modelos pré-treinados da biblioteca Hugging Face Transformers.

```

1 from transformers import AutoTokenizer, AutoModelForCausalLM,
  DPRQuestionEncoder, DPRContextEncoder
2
3 # Carregar o tokenizer e o modelo LLaMA
4 llama_tokenizer = AutoTokenizer.from_pretrained(
5 "meta-llama/LLaMA-7B")

```

```

6 llama_model = AutoModelForCausalLM.from_pretrained(
7     "meta-llama/LLaMA-7B")
8
9 # Configurar DPR para recuperação de passagens
10 question_encoder = DPRQuestionEncoder.from_pretrained(
11     "facebook/dpr-question_encoder-single-nq-base")
12 context_encoder = DPRContextEncoder.from_pretrained(
13     "facebook/dpr-ctx_encoder-single-nq-base")

```

```

1 Some weights of the model checkpoint at
  facebook/dpr-ctx_encoder-single-nq-base were not used when
  initializing DPRContextEncoder:
  ['ctx_encoder.bert_model.pooler.dense.bias',
   'ctx_encoder.bert_model.pooler.dense.weight']
2 - This IS expected if you are initializing DPRContextEncoder from the
  checkpoint of a model trained on another task or with another
  architecture (e.g. initializing a BertForSequenceClassification model
  from a BertForPreTraining model).
3 - This IS NOT expected if you are initializing DPRContextEncoder from the
  checkpoint of a model that you expect to be exactly identical
  (initializing a BertForSequenceClassification model from a
  BertForSequenceClassification model).

```

Processamento da Entrada e Recuperação de Informações: Em seguida, implementamos a lógica para processar a entrada do usuário e, se necessário, recuperar informações relevantes de uma base de dados. O bloco de código abaixo deve ser executado no Google Colab, em outra célula, depois do código anterior.

Código-fonte 4.33: Implementa uma função de recuperação de passagens relevantes usando embeddings de linguagem.

```

1 # Este código deve ser executado na sequencia, depois do código anterior.
2 import torch
3
4 def retrieve_relevant_passage(query, passages):
5     query_input = question_tokenizer(query, return_tensors="pt")
6     query_embedding = question_encoder(
7         **query_input).pooler_output
8
9     passage_inputs = context_tokenizer(passages, padding=True,
10         truncation=True, return_tensors="pt")
11     passage_embeddings = context_encoder(
12         **passage_inputs).pooler_output

```

```

12
13 similarity_scores = torch.matmul(query_embedding, passage_embeddings.T)
14 best_passage_index = torch.argmax(similarity_scores, dim=1).item()
15
16 return passages[best_passage_index]
17
18 # Exemplo de passagens
19 passages = [
20     "LLaMA é um modelo de linguagem desenvolvido pela Meta AI.",
21     "RAG combina recuperação de informações com geração de texto.",
22     "GPT-3 é um dos maiores modelos de linguagem disponíveis."
23 ]
24
25 # Entrada do usuário
26 user_input = "O que é LLaMA?"
27
28 # Recuperar a passagem mais relevante
29 relevant_passage = retrieve_relevant_passage(user_input, passages)
30
31 print(f"Passagem mais relevante: {relevant_passage}")

```

```

1 Passagem mais relevante: LLaMA é um modelo de linguagem desenvolvido pela
  Meta AI.

```

Geração de Resposta com LLaMA: Finalmente, usamos o modelo LLaMA para gerar uma resposta, utilizando tanto a entrada original do usuário quanto a passagem recuperada. O bloco de código abaixo deve ser executado no Google Colab, em outra célula, depois do código anterior.

Código-fonte 4.34: Realiza a geração de uma resposta de chatbot usando um modelo LLM carregado.

```

1 # Concatenar a consulta com a passagem relevante
2 input_text = user_input + " " + relevant_passage
3
4 # Geração da resposta
5 input_ids = llama_tokenizer.encode(input_text, return_tensors="pt")
6 generated_ids = llama_model.generate(input_ids, max_length=50,
   num_beams=5, early_stopping=True)
7 response = llama_tokenizer.decode(generated_ids[0],
   skip_special_tokens=True)
8
9 print("Resposta do chatbot:", response)

```

Resposta do chatbot: O que é LLaMA? LLaMA é um modelo de linguagem desenvolvido pela Meta AI. LLaMA é um acrônimo para Large Language Model, que significa modelo de linguagem grande. LLaMA é

Neste exemplo, o chatbot gera uma resposta baseada na combinação da entrada do usuário e na informação relevante recuperada, criando uma resposta informativa e contextualizada.

Desafios na Implementação de Chatbots: Implementar chatbots com múltiplas técnicas apresenta vários desafios, tais como: combinar modelos como LLaMA e DPR pode ser computacionalmente intensivo, especialmente em aplicações em tempo real; o *Fine-Tuning* e a configuração de modelos precisam ser bem ajustados para garantir que o chatbot seja eficaz e relevante em suas respostas; além disso, garantir que os diferentes componentes (recuperação, geração, etc.) funcionem de maneira coesa pode ser desafiador. Mesmo assim, os chatbots podem ser aplicados em uma variedade de cenários, como: fornecer suporte automatizado, personalizado, em tempo real; ajudar na triagem de sintomas ou fornecer informações médicas básicas; ou mesmo criar tutores virtuais que podem responder a perguntas de estudantes de maneira precisa e contextualizada.

4.9 API e Playground

Geralmente, a desenvolvedora de um modelo LLM fornece uma interface web para o usuário conversar com o seu modelo em forma de chatbot e fornece uma API (com um playground) para quem deseja integrar a solução em suas aplicações. Assim é o ChatGPT, da empresa OpenAI. Ela cobra um valor fixo para disponibilizar aos assinantes acesso completo aos recursos do seu chatbot, inclusive com o recurso de SearchGPT (uma busca na web), lousa, geração de vídeos, imagens etc. Já para acesso à API, a modalidade de cobrança é paga-pelo-uso. Veja na Figura 4.13 uma imagem do chatbot da OpenAI e na Figura 4.14 o playground da OpenAI.

Existe uma provedora de inferência chamada GROQ <https://groq.com/>. O GROQ (com Q) é uma fornecedora de infraestrutura para inferência em modelos de linguagem de grande porte (LLM). Esta empresa vende um serviço de inferência mais rápido do que seus concorrentes. No GROQ, podemos utilizar

Figura 4.13: Chatbot da OpenAI.

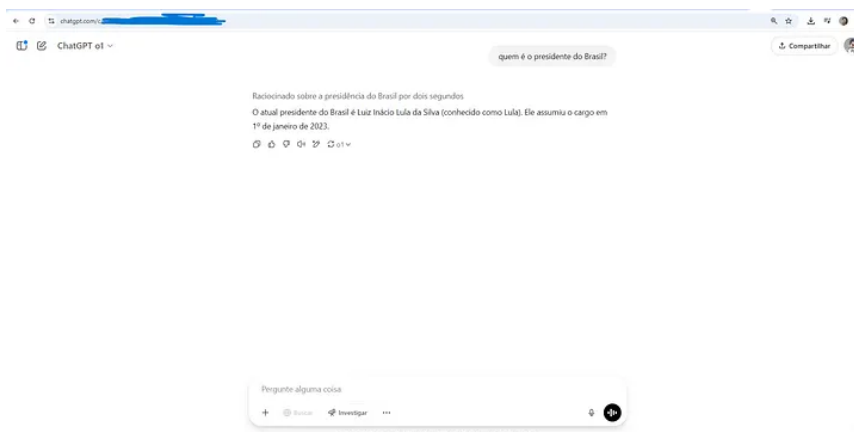
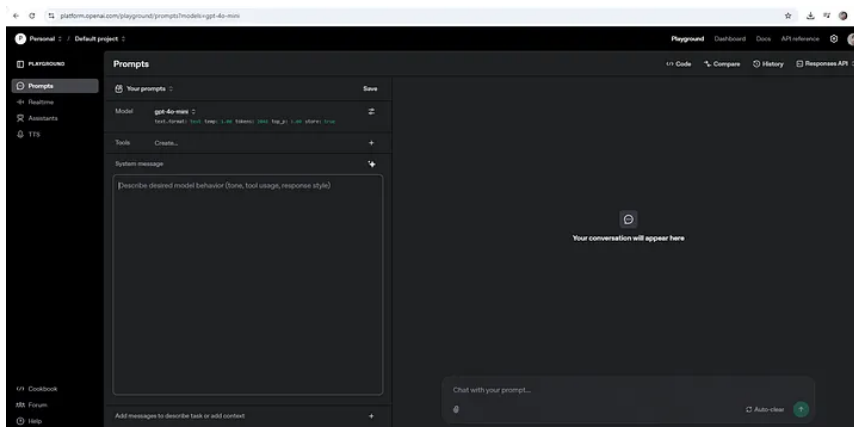


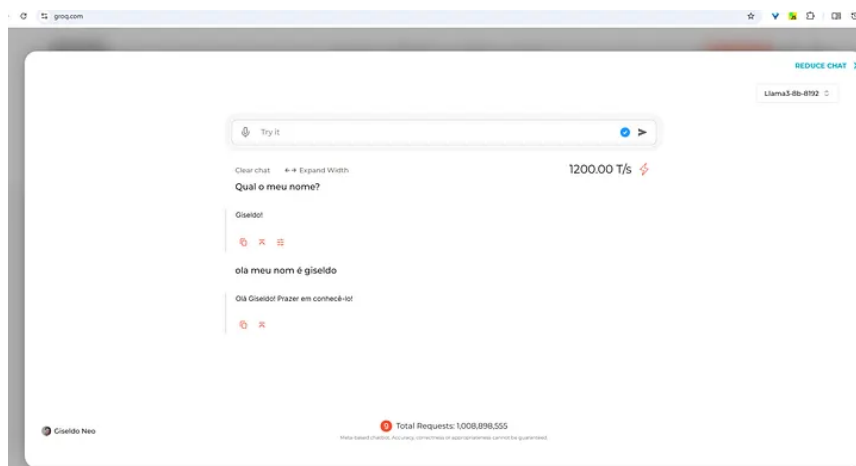
Figura 4.14: Playground da openai.



vários modelos LLMs disponíveis abertamente, tais como o Llama da Meta, o Gemma do Google ou o Mixtral da Mixtral AI. O ChatGPT não está disponível no GROQ pois ele não é open source.

Entre outros modelos, utilizam uma versão do LLaMA em seu Chatbot GROQ, porém seu foco é a API, que pode ser testada no playground GROQ. O Playground funciona no formato pague-pelo-uso. O Chatbot GROQ também consome os mesmos créditos pague-pelo-uso, não tendo uma assinatura à parte, como no caso do chatbot do ChatGPT. Veja na Figura 4.15 o Chatbot do GROQ e na Figura 4.16 o Playground do GROQ. Note que no Chatbot do GROQ não existe o recurso de histórico como no ChatGPT.

Figura 4.15: Chatbot do GROQ.



Já o Grok (com K) é utilizado em referência ao chatbot da empresa xAI. Ele é um chatbot e está disponível para quem tem o acesso premium ao X (ex-twitter), enquanto a API Grok pode ser acessada na forma pague-pelo-uso. Inclusive, o Grok está oferecendo este ano 25 R\$ para quem deseja testar o serviço, ou seja, é possível utilizar a API do Grok sem desembolsar nada neste momento. Porém, o Grok não tem um playground avançado, logo, para usar a API do Grok é necessário utilizar uma ferramenta gráfica ou código externo, por exemplo: LLM Studio, JAN AI, GPT4ALL ou o excelente Msty app. Veja na Figura 4.17 o chatbot do Grok e na Figura 4.18 o playground do Grok.

Figura 4.16: Playground do GROQ.

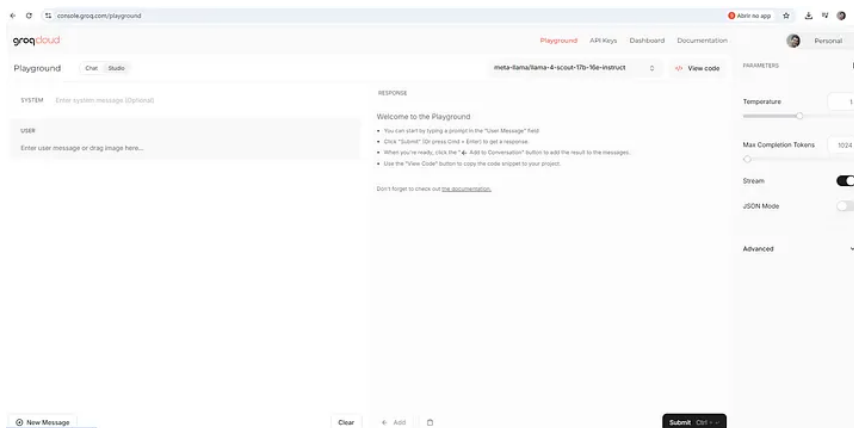


Figura 4.17: Interface do Grok.

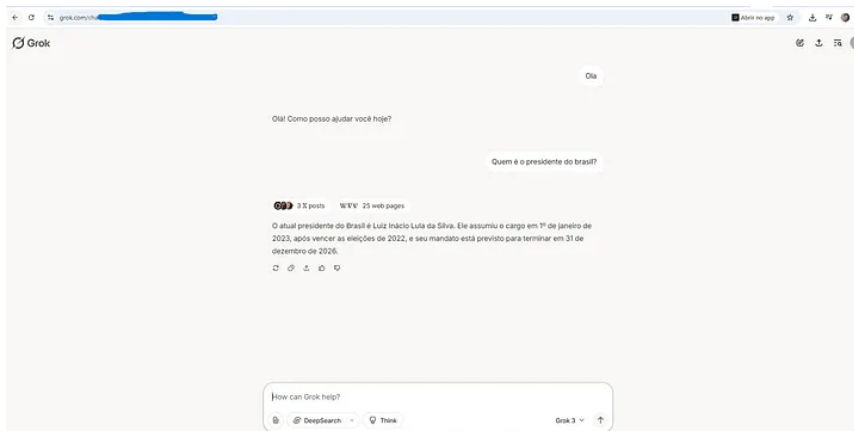
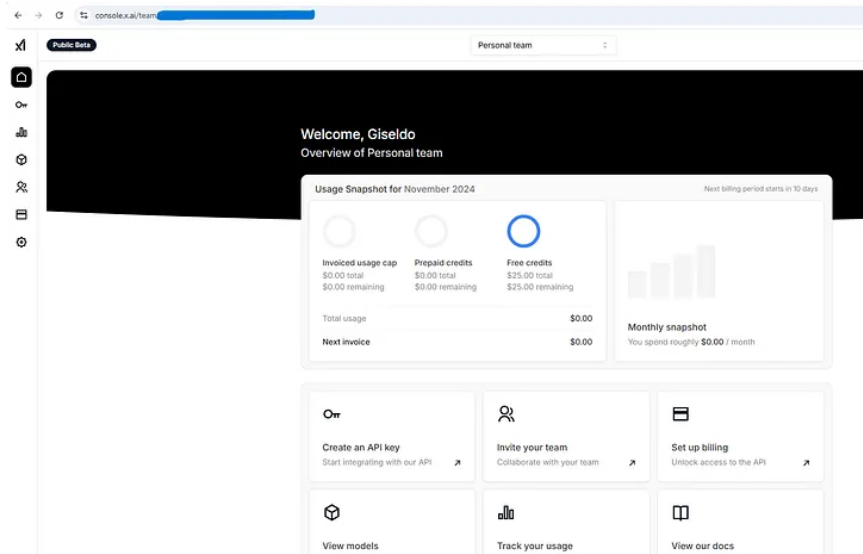


Figura 4.18: Playground do Grok.



4.10 Resumo

Os Modelos de Linguagem Grande (LLMs) têm desempenhado um papel central na revolução do Processamento de Linguagem Natural (PLN). Esses modelos, que incluem variantes como GPT, BERT, e seus sucessores, são capazes de realizar uma ampla gama de tarefas, desde geração de texto até compreensão profunda de linguagem, graças ao seu treinamento em grandes volumes de dados textuais.

Esses modelos são alimentados por vastos conjuntos de dados textuais e utilizam técnicas de aprendizado profundo, particularmente redes neurais, para aprender padrões, contextos e nuances da linguagem. Eles são capazes de realizar uma variedade de tarefas linguísticas, incluindo tradução automática, geração de texto, resumo de informações, resposta a perguntas e até mesmo a criação de diálogos interativos. Funcionam com base em arquiteturas complexas, como a *Transformer*, que permite que o modelo preste atenção a diferentes partes de um texto simultaneamente, facilitando a compreensão do contexto e das relações semânticas entre palavras e frases.

O treinamento desses modelos envolve a exposição a enormes quantidades de texto, o que lhes permite desenvolver uma compreensão profunda da gramática, do vocabulário e dos estilos de comunicação. No entanto, essa capacidade de gerar texto coerente e relevante também levanta questões éticas e de responsabilidade, especialmente em relação à desinformação, vies algorítmico e privacidade.

Eles hoje representam um marco significativo na evolução da inteligência artificial, assim como foi o Eliza em sua época, porém com um impacto maior, oferecendo ferramentas para a interação humano-computador e abrindo novas possibilidades para aplicações em diversas áreas, como educação [da Silva et al., 2020], atendimento ao cliente ou criação de conteúdo.

Neste capítulo, exploramos a arquitetura *Transformer*, que revolucionou o Processamento de Linguagem Natural ao introduzir o mecanismo de atenção. Discutimos como os *Transformers* funcionam e como eles são aplicados em modelos populares como BERT e GPT. Com exemplos em Python, vimos como utilizar esses modelos para tarefas de PLN.

Também, exploramos a técnica de Retrieval-Augmented Generation (RAG), uma abordagem que combina a recuperação de informações com a geração de texto. Vimos como implementar RAG em Python usando modelos como DPR e BART, e discutimos as aplicações e desafios dessa técnica.

Adicionalmente, exploramos os LLMs incluindo suas arquiteturas, técnicas de treinamento e principais aplicações. Modelos como GPT, BERT, T5 e XLNet exemplificam como os LLMs estão redefinindo o campo do PLN. Com exemplos práticos, demonstramos como esses modelos podem ser aplicados em uma variedade de tarefas.

Outro tópico explorado neste capítulo foi o processo de *Fine-Tuning* de modelos pré-treinados, discutindo sua importância, desafios e aplicações práticas. O *Fine-Tuning* permite que Modelos de Linguagem Grande sejam adaptados para tarefas específicas com eficiência, tornando-os extremamente versáteis em diversas aplicações de PLN.

Também, exploramos o LLaMA, um modelo de linguagem projetado para ser eficiente e acessível sem comprometer a capacidade de realizar tarefas complexas de PLN. Com uma arquitetura otimizada e foco em eficiência computacional, LLaMA oferece uma alternativa prática para modelos gigantescos como o GPT-3.

Vimos também como implementar LLaMA em Python para tarefas de geração de texto e discutimos suas aplicações e limitações.

Além disso, integramos várias técnicas discutidas em capítulos anteriores para criar um chatbot capaz de fornecer respostas precisas e contextualizadas utilizando LLaMA e RAG. A implementação dessas técnicas oferece uma pequena base para o desenvolvimento de sistemas de diálogo eficientes.

Capítulo 5

Conclusão

Uma vez que um programa é explicado em linguagem suficientemente simples o observador diz para si mesmo “Eu poderia ter escrito isso”.

Joseph Weizenbaum

Objetivo
Refletir sobre ética, privacidade, viés algorítmico e perspectivas futuras dos chatbots, incentivando um uso responsável e inovador dessas tecnologias.

5.1 Considerações Éticas

O desenvolvimento de chatbots não envolve apenas desafios técnicos. À medida que esses sistemas se tornam cada vez mais sofisticados e amplamente utilizados, questões éticas ganham destaque. O uso de chatbots precisa ser entendido, experimentado e avaliado, para identificarmos os impactos de tal utilização, mediante experimentos, como por exemplo da Silva et al. [2020], que avaliaram seu uso em sala de aula com uma turma de alunos.

Os impactos da Inteligência Artificial (IA) no bem-estar humano – positivo ou negativo – são mais complexos do que às vezes se supõe [Schiff et al., 2020]. No campo da ética, algumas abordagens específicas, tal como a aristotélica, reconhecem a noção de felicidade como cerne da conduta ética. Um representante da modernidade é a filosofia kantiana, que estabelece a distinção entre moralidade e civilidade. Kant traz à tona a noção de que o moralmente correto não é apenas a ação externa do indivíduo, pois o indicativo externo só pode ser mensurado pelas leis e normas. Uma moralidade verdadeiramente ética seria aquela que parte de uma convicção; para ele, “nós somos civilizados até a saturação por toda espécie de boas maneiras e decore social. Mas ainda falta muito para nos considerarmos moralizados” [Kant, 2023]. Nessa mesma perspectiva, Kant elabora seu imperativo categórico, uma máxima norteadora da ação ética: “age de tal modo que tua máxima possa ser universalizada” [Kant, 2017].

Hans Jonas, a partir de suas reflexões éticas acerca do meio ambiente, se opõe claramente à ética kantiana, Jonas assevera: “Age de tal forma que os efeitos da tua ação sejam compatíveis com a permanência de uma vida autenticamente humana sobre a terra.” [Jonas, 2006]. Jonas alarga a noção de ética, descentrando-a das ações temporais e projetando-as para o futuro; a ação do indivíduo é responsável pelos seus desdobramentos futuros.

Ao longo dos anos, importantes esforços de pesquisa têm sido dedicados à IA, e em especial aos chatbots. Tradicionalmente, um dos aspectos que mais chamou a atenção foram suas capacidades de interação destes programas com os seres humanos. Por exemplo, o Eliza foi um dos primeiros e mais influentes chatbots que se baseou em correspondência de padrões para criar esta ilusão de inteligência [Weizenbaum, 1966].

É sempre desafiador pensar na perspectiva moral e ética na relação tecnologia e usuário. Faz-se necessário refletir em torno da dicotomia moral e ética,

pois, embora muitas vezes associados como elementos idênticos, possuem especificidades. São justamente essas especificidades que tornam candente o ensino da ética em sala de aula. Ora, podemos asseverar que ética e moral se coadunam quando pensadas como elementos do agir humano. Juízos de valor, noções de bem e mal, normas de conduta, são elementos comuns à ética e à moral. Entretanto, ética e moral separam-se quando entendemos que esta última se concentra nos valores vigentes de uma sociedade. Assim, tempo e espaço são elementos que alteram a moral, ou melhor, as morais. Dito de outra forma, a conduta valorada como correta em um período pode não ser em outro, bem como o comportamento criminoso de uma região pode, ao mesmo tempo, ser correto em outra. Esse veio que particulariza as ações dos indivíduos, de acordo com a sociedade em que vivemos, chamamos de moral.

Por ética, em linhas gerais, consideramos as reflexões filosóficas que se desdobram sobre os diversos sistemas morais. O processo de interação homem-tecnologia deve sempre partir, em um movimento dialético, dos valores já sedimentados pelos usuários em questão. Não há grau zero do conhecimento. Ao tratarmos pessoas, deparamo-nos com seus sistemas morais, cristalizados desde a infância. O papel da filosofia, assim, não é de ser a demolidora dos valores. Entendemos que as diversas perspectivas filosóficas constituem-se essencialmente radicais, posto que se debruçam sobre a raiz e universalidade dos problemas.

A filosofia pode ser utilizada primordialmente como elemento de crítica, sendo esta, em termos kantianos, uma “condição de possibilidade”. Intentando problematizar a dificuldade desta interação, compreendemos como a formação social da moral nos indivíduos necessita ser pensada à luz da crítica. Os usos da tecnologia, sobremaneira, impuseram uma dinâmica até então não empreendida antes do seu surgimento: a velocidade das mudanças, a internet e, recentemente, os chatbots aceleram o acesso à informação, tornando descompassada sua relação com o conhecimento. Porém, a filosofia evidencia a necessidade de ajuste do compasso entre tecnologia, conhecimento e seres humanos.

A tecnologia pode ser aliada aos processos humanos e pode minimizar o cenário das dificuldades vividas. Porém, não basta simplesmente concluir que o benefício surge única e exclusivamente a partir da simples adoção dos chatbots. A simples incorporação dos chatbots ou uso deles não gera processos

de inovação e melhoria; estes são determinados a partir dos usos específicos que parecem ter a capacidade de desencadear esses processos.

Temos diversos desafios pela frente, entre eles: Como garantir um uso eficiente dos chatbots e qual o impacto social desta interação? É possível asseverar que há um impacto social em refletir sobre a ética no uso dos chatbots. É justamente pela falta de reflexão e crítica que temos perpetuado em nossa sociedade sistemas de dominação e corrupção. A naturalização acrítica dos valores vigentes permite que tais valores sejam transmitidos como verdades, ocultando muitas vezes os interesses individuais em detrimento dos coletivos.

Desenvolvedores de chatbots têm a responsabilidade de garantir que seus sistemas sejam projetados e operados de maneira ética. Algumas recomendações podem incluir auditorias regulares para implementar revisões regulares de privacidade, segurança e viés para garantir a conformidade com os princípios éticos; o envio de notificações para informar os usuários de forma clara quando estão interagindo com um chatbot, e não com um humano, para evitar confusões. Além de processos para garantir que as empresas possam ser responsabilizadas por suas ações.

5.2 Privacidade e Proteção de Dados

Os chatbots modernos, especialmente aqueles implantados em plataformas de atendimento ao cliente e assistentes pessoais, frequentemente processam informações pessoais e sensíveis. A coleta, armazenamento e uso de dados precisam estar em conformidade com regulamentos de proteção de dados, como o GDPR (Regulamento Geral de Proteção de Dados) na União Europeia.

Os seguintes princípios são fundamentais para garantir a privacidade dos dados:

- **Minimização de Dados:** Coletar apenas os dados estritamente necessários para a tarefa em questão.
- **Transparência:** Informar os usuários sobre quais dados estão sendo coletados, como serão usados e por quanto tempo serão armazenados.
- **Consentimento:** Garantir que os usuários concordem explicitamente com o uso de seus dados.

- **Anonimização:** Sempre que possível, anonimizar os dados para evitar que informações pessoais possam ser associadas a um indivíduo específico.

Este exemplo em python mostra como dados sensíveis, como informações pessoais, podem ser protegidos com criptografia em repouso e em trânsito, garantindo que apenas usuários autorizados possam acessá-los.

Código-fonte 5.1: Este código Gera uma chave de criptografia; Criptografa dados sensíveis; Descriptografa os dados e Exibe o resultado descriptografado no console.

```
1 # pip install cryptography
2 from cryptography.fernet import Fernet
3
4 # Gerar uma chave de criptografia
5 key = Fernet.generate_key()
6 cipher_suite = Fernet(key)
7
8 # Criptografar dados sensíveis
9 data = "Dados sensíveis do usuário"
10 encrypted_data = cipher_suite.encrypt(data.encode())
11
12 # Descriptografar quando necessário
13 decrypted_data = cipher_suite.decrypt(encrypted_data).decode()
14 print("Dados descriptografados:", decrypted_data)
```

```
1 Dados descriptografados: Dados sensiveis do usuário
```

5.3 Viés Algorítmico

Viés algorítmico ocorre quando um modelo de chatbot aprende padrões indesejáveis ou preconceituosos a partir dos dados de treinamento, levando a resultados injustos ou discriminatórios. Este problema pode se manifestar de várias formas, desde o uso de linguagem tendenciosa até a recomendação de serviços que favoreçam certos grupos em detrimento de outros.

Os vieses nos chatbots podem surgir dos dados de treinamento, pois os dados usados para treinar o chatbot contêm preconceitos históricos; esses preconceitos podem ser perpetuados pelo modelo. Também pode existir viés nas interações do Usuário, pois, caso o chatbot aprenda com as interações de usuários, pode absorver e amplificar vieses existentes nas entradas dos usuários.

Além disso, alguns algoritmos podem inadvertidamente favorecer certos tipos de dados em detrimento de outros.

Mitigar o viés algorítmico requer abordagens proativas, tais como: a diversificação de dados de treinamento para garantir que o chatbot seja treinado em um conjunto de dados diversificado que represente várias culturas, gêneros, idades e contextos sociais; outra forma é regularmente auditar os modelos de chatbot para identificar e corrigir viés indesejado; e por fim, utilizar algoritmos que considerem equidade e justiça ao treinar o chatbot.

Código-fonte 5.2: O código verifica possíveis vieses nas respostas do chatbot analisando os acertos e erros das predições.

```
1 # pip install scikit-learn
2 from sklearn.metrics import confusion_matrix
3
4 # Verificar o viés nas predições do chatbot
5 def evaluate_bias(true_labels, predictions):
6     cm = confusion_matrix(true_labels, predictions)
7     print("Matriz de confusão:\n", cm)
8
9 # Exemplo de auditoria simples de viés
10 true_labels = [0, 1, 1, 0, 1] # Representa as classificações corretas
11 predictions = [0, 1, 0, 0, 1] # Predições do chatbot
12
13 evaluate_bias(true_labels, predictions)
```

```
1 Matriz de confusão:
2 [[2 0]
3  [1 2]]
```

Este código simula uma auditoria básica para verificar a presença de viés em um modelo de chatbot, comparando predições com rótulos verdadeiros para identificar possíveis discrepâncias.

5.4 Segurança de Chatbots

Os chatbots, especialmente aqueles integrados em sistemas críticos, são alvos potenciais para ataques cibernéticos. Garantir a segurança desses sistemas é essencial para proteger tanto os usuários quanto as empresas.

Algumas das principais vulnerabilidades de chatbots incluem:

- **Ataques de Injeção de Código:** Um atacante pode tentar injetar código malicioso nas entradas do chatbot para comprometer o sistema.
- **Ataques de Engenharia Social:** Os usuários podem ser enganados para compartilhar informações confidenciais, acreditando que estão falando com um agente confiável.
- **Abuso de API:** Se a API usada pelo chatbot não estiver adequadamente protegida, um atacante pode explorá-la para acessar dados ou realizar ações não autorizadas.

Medidas de segurança para proteger chatbots incluem:

- **Validação de Entrada:** Implementar uma validação rigorosa das entradas do usuário para evitar ataques de injeção de código.
- **Autenticação de Usuário:** Exigir autenticação para acessar funcionalidades críticas do chatbot, como alteração de dados pessoais.
- **Limitação de Taxa (Rate Limiting):** Impedir o abuso da API limitando o número de solicitações que podem ser feitas por um usuário em um determinado período de tempo.

Código-fonte 5.3: um exemplo de como aplicar limitação de taxa (rate limiting) em uma rota de API usando Flask.

```
1 # pip install flask_limiter
2 # pip install flask
3
4 from flask_limiter import Limiter
5 from flask import Flask
6
7 app = Flask(__name__)
8 limiter = Limiter(key_func=lambda: "user_ip")
9 limiter.init_app(app)
10
11 @app.route("/secure-endpoint")
12 @limiter.limit("5 per minute")
13 def secure_endpoint():
14     return "Acesso seguro garantido!"
```

```
1 UserWarning: Using the in-memory storage for tracking rate limits as no
   storage was explicitly specified. [...] documentation about
   configuring the storage backend. [...]
```

Neste exemplo, utilizamos a limitação de taxa para proteger um endpoint sensível, garantindo que os usuários não possam abusar da API.

Impacto Social dos Chatbots: Além das questões técnicas e de segurança, os chatbots podem ter um impacto social significativo. Eles podem influenciar o comportamento dos usuários, moldar interações sociais e até substituir trabalhos humanos em determinadas indústrias.

Substituição de Trabalho Humano: Embora os chatbots possam aumentar a eficiência e reduzir custos, sua implementação pode resultar na substituição de empregos, especialmente em setores de atendimento ao cliente. É importante considerar como a automação pode ser introduzida de forma ética, proporcionando requalificação e apoio a trabalhadores impactados.

Manipulação e Desinformação: Os chatbots também podem ser usados para manipulação e disseminação de desinformação. Modelos de Linguagem Grande podem ser explorados para criar bots maliciosos que disseminam fake news, discursos de ódio ou influenciam eleições. Uma possibilidade de mitigação seria implementar filtros de moderação e ferramentas de verificação de fatos em chatbots para impedir a disseminação de informações enganadoras ou prejudiciais.

5.5 Manutenção e Atualização

Implantar um chatbot é apenas o início. Para garantir que ele continue a atender às necessidades dos usuários de forma eficaz e segura, é essencial implementar um processo contínuo de manutenção e atualização. Nesta seção, vamos explorar as melhores práticas para manter um chatbot em operação, incluindo monitoramento, atualização de modelos, coleta e análise de feedback dos usuários, e práticas para prevenir a deterioração do desempenho.

Um chatbot que não é mantido adequadamente pode rapidamente se tornar obsoleto, irrelevante ou até prejudicial para a experiência do usuário. A manutenção contínua é necessária para:

- **Atualizar Conhecimento:** Adaptar o chatbot a novas informações, gírias ou mudanças de contexto.
- **Melhorar o Desempenho:** Ajustar o chatbot com base no feedback do usuário para aprimorar a precisão e a relevância das respostas.

- **Garantir a Segurança:** Aplicar patches de segurança e atualizações para proteger o chatbot de vulnerabilidades.
- **Adaptar-se às Mudanças no Ambiente:** Ajustar o chatbot às mudanças na infraestrutura técnica, como atualizações em APIs de integração.

Já o monitoramento contínuo do chatbot em produção é fundamental para identificar problemas de desempenho, compreender como os usuários estão interagindo com o sistema e detectar comportamentos inesperados ou indesejados. Algumas métricas importantes para monitorar incluem:

Atualização de Modelos: Com o tempo, os modelos de linguagem usados pelos chatbots podem se tornar desatualizados, especialmente se o domínio de aplicação estiver em constante evolução. A atualização dos modelos pode envolver realizar re-treinamento com dados novos, ou incorporar novos dados para que o modelo se adapte a mudanças de linguagem, gírias ou tópicos emergentes. Outra técnica é realizar *Fine-Tuning* periódico com base em novos exemplos de interações reais dos usuários.

Feedback do Usuário e Melhorias Contínuas: O feedback dos usuários é uma fonte valiosa de informações para melhorar o chatbot. Processos automatizados e manuais de coleta e análise de feedback ajudam a identificar áreas para aprimoramento. É possível realizar a coleta de feedback via solicitação direta, perguntando diretamente aos usuários sobre sua satisfação com as respostas fornecidas, ou usar algoritmos para analisar o sentimento geral das interações dos usuários com o chatbot. De posse do feedback do chatbot, é possível refinar respostas problemáticas identificadas através do feedback para incluir novos dados ou ajustar o modelo para melhor atender às necessidades dos usuários.

Prevenção de Deterioração de Desempenho: Com o tempo, chatbots podem sofrer deterioração de desempenho devido a mudanças no ambiente ou desgaste do modelo. Estratégias para prevenir isso incluem detectar e corrigir quando o desempenho do modelo começa a se degradar em relação a novas entradas de dados; além de avaliações regulares, realizando testes regulares para garantir que o chatbot continue operando conforme esperado; por fim, manter versões anteriores do chatbot para comparação e possível rollback em caso de problemas com novas versões.

Segurança e Privacidade: Garantir a segurança contínua e a privacidade dos dados do usuário é uma prioridade. Isso envolve manter o ambiente e

as dependências do chatbot atualizados com os patches de segurança mais recentes, o que chamamos de Aplicação de Patches de Segurança; além de implementar medidas para proteger dados sensíveis, incluindo criptografia de dados em repouso e em trânsito, e conformidade com regulamentos como o GDPR.

5.6 Perspectivas Futuras

À medida que a tecnologia avança, os chatbots estão se tornando cada vez mais sofisticados e capazes de realizar tarefas complexas que antes eram consideradas impossíveis. A seguir, destacamos algumas das principais tendências e áreas de pesquisa que provavelmente moldarão o futuro dos chatbots:

Modelos de Linguagem Multimodais: Chatbots PLN não se limitam apenas ao texto. Modelos multimodais, que combinam texto com imagens, áudio e até mesmo vídeo, já são uma nova fronteira no desenvolvimento de chatbots. Esses modelos permitem que chatbots compreendam e respondam a entradas que combinam diferentes tipos de dados, oferecendo experiências mais ricas e interativas. Hoje, chatbots multimodais já analisam imagens enviadas por um usuário e fornecem diagnósticos ou recomendações, além de responderem a perguntas textuais relacionadas.

Personalização Avançada: À medida que a capacidade de coleta e análise de dados se expande, a personalização de chatbots se tornará mais sofisticada. Chatbots serão capazes de adaptar suas respostas com base no histórico de interações, preferências do usuário e até mesmo no contexto emocional, criando interações verdadeiramente personalizadas. Hoje, assistentes virtuais ajustam as recomendações de compras ou serviços com base nos hábitos de consumo e humor do usuário.

Chatbots Colaborativos e de Aprendizagem Contínuo: Chatbots que podem aprender de forma contínua e colaborativa com outros sistemas e com os próprios usuários serão uma área chave de desenvolvimento. Isso permitirá que chatbots se adaptem rapidamente a novos domínios e que integrem informações em tempo real, melhorando sua eficácia e utilidade. Um exemplo de aplicação seria um chatbot que aprende novos termos e conceitos diretamente das interações com usuários e os compartilha com outros chatbots, criando uma rede

de conhecimento distribuída. Outro exemplo é um chatbot que discute Ecologia com estudantes do ensino médio [Neo et al., 2023a].

Segurança e Privacidade Melhoradas: Com a crescente sofisticação dos chatbots, as questões de segurança e privacidade se tornarão ainda mais críticas. Avanços em criptografia, anonimização de dados e conformidade com regulamentos serão essenciais para garantir que chatbots possam ser usados com confiança em setores sensíveis, como saúde e finanças. Um exemplo seriam chatbots médicos que garantem a privacidade total dos dados do paciente enquanto oferecem diagnósticos e recomendações de tratamento, por exemplo Pires [2024].

Explicabilidade e Transparência dos Modelos: Com o uso crescente de Modelos de Linguagem Grande e complexos, a explicabilidade e a transparência se tornarão áreas críticas de pesquisa. Ferramentas e técnicas para explicar como um chatbot chegou a uma determinada resposta serão cada vez mais demandadas, especialmente em setores regulamentados. Uma solução pode ser fornecer uma explicação detalhada de como chegaram a uma decisão ou recomendação, aumentando a confiança dos usuários.

5.7 Oportunidades de Inovação

À medida que o campo dos chatbots continua a evoluir, novas oportunidades para inovação e pesquisa emergem constantemente. Nesta seção, vamos explorar as áreas emergentes. Também discutiremos os desafios abertos que podem ser abordados e como os desenvolvedores e pesquisadores podem contribuir para o avanço deste campo, explorando as tendências tecnológicas e áreas promissoras de estudo.

Tendências Tecnológicas Emergentes

As tecnologias de Processamento de Linguagem Natural (PLN) e Inteligência Artificial (IA) estão em rápida evolução. Algumas das principais tendências que moldarão o futuro dos chatbots incluem:

Vibe Coding

Em sua essência, vibe coding é uma abordagem ao desenvolvimento de software que se apoia fortemente na IA para gerar código. Em vez de escrever meticulosamente cada linha de código você mesmo, você descreve o que quer alcançar em linguagem natural, e a IA faz o trabalho pesado. É como ter um par de programação de IA que compreende suas vibrações (daí o nome) e as traduz em código funcional.

Este conceito ganhou força com os avanços recentes em Modelos de Linguagem Grande (LLMs) como o GPT-4. Essas ferramentas de IA se tornaram cada vez mais hábeis em compreender a intenção humana e gerar código correspondente, tornando o processo de desenvolvimento mais rápido e acessível.

Características do Vibe Coding:

- **Abstração:** Vibe coding abstrai as complexidades da escrita de código tradicional, permitindo que os programadores se concentrem na lógica de nível superior e na criatividade.
- **Eficiência:** Ao automatizar tarefas repetitivas e gerar código boilerplate, o vibe coding pode acelerar significativamente o processo de desenvolvimento.
- **Acessibilidade:** O vibe coding pode potencialmente democratizar o desenvolvimento de software, tornando-o mais acessível a pessoas com menos experiência em programação.
- **Colaboração:** O vibe coding pode facilitar novas formas de colaboração entre humanos e IA, com os programadores trabalhando em conjunto com as ferramentas de IA para construir software.

Embora o vibe coding ainda esteja em seus estágios iniciais, tem o potencial de revolucionar o desenvolvimento de software. Imagine você poder construir um aplicativo simplesmente descrevendo-o em uma conversa com um assistente de IA. Este cenário pode estar mais perto do que pensamos.

No entanto, o vibe coding também levanta questões importantes. Irá levar a uma perda de controle sobre o processo de desenvolvimento? Quais são as implicações para a qualidade do código, segurança e manutenção a longo prazo? À medida que o vibe coding continua a evoluir, os programadores terão de navegar por estas questões e encontrar um equilíbrio entre aproveitar o poder da IA.

Model Context Protocol

O *Model Context Protocol* (MCP) é um padrão aberto e um *framework* de código aberto introduzido pela Anthropic em 2024 para padronizar a forma como sistemas de IA, especialmente grandes modelos de linguagem (LLMs), se conectam a ferramentas e fontes de dados externas [Anthropic, 2024]. Seu objetivo principal é unificar o modo como modelos acessam arquivos, executam funções externas e utilizam *prompts* contextuais, substituindo integrações ad hoc por uma interface consistente. Dessa forma, o MCP resolve o problema de integração $N \times M$, em que cada agente precisava de conectores específicos para cada ferramenta, tornando o desenvolvimento mais complexo e difícil de escalar [Anthropic, 2024].

Do ponto de vista conceitual, o MCP utiliza uma arquitetura cliente-servidor em que o agente de IA atua como cliente MCP e se comunica com um ou mais servidores MCP que expõem ferramentas, recursos e *prompts* [Group, 2010]. Toda a comunicação é realizada usando o protocolo JSON-RPC 2.0, que padroniza as mensagens de requisição e resposta em formato JSON e é independente de linguagem de programação. O MCP suporta transporte por *stdio* para execução local com baixa latência e transporte via HTTP com suporte a *Server-Sent Events* para cenários distribuídos ou remotos [Group, 2010]. Essa padronização garante interoperabilidade e permite que novos servidores ou ferramentas possam ser descobertos dinamicamente pelo cliente, que pode então invocá-los com parâmetros estruturados e receber resultados também em formato JSON.

A adoção do MCP traz vantagens significativas para o ecossistema de agentes de IA, pois elimina a dependência de soluções proprietárias e facilita a criação de ambientes híbridos, onde modelos locais ou em nuvem podem acessar os mesmos recursos de maneira uniforme. Em aplicações corporativas ou privadas, o MCP permite que servidores sejam executados localmente, garantindo que dados sensíveis não precisem sair do ambiente da organização, atendendo a requisitos de segurança e conformidade. Além disso, ao ser um protocolo aberto, incentiva a colaboração e a criação de um ecossistema de conectores reutilizáveis, reduzindo o custo de desenvolvimento e favorecendo a interoperabilidade entre diferentes plataformas de IA [Anthropic, 2024].

Outras tendências

Inteligência Artificial Explicável (XAI): Com o aumento da complexidade dos modelos de linguagem, a necessidade de Inteligência Artificial Explicável (XAI) se torna mais premente. XAI visa tornar os sistemas de IA mais transparentes e compreensíveis para os seres humanos, o que é importante para garantir a confiança do usuário em decisões automatizadas. Um exemplo de aplicação é desenvolver chatbots que possam explicar como chegaram a uma determinada resposta, oferecendo aos usuários uma visão dos processos de tomada de decisão do modelo.

Modelos de Linguagem Contínuos e Atualizáveis: À medida que o conhecimento e a linguagem evoluem, há uma crescente necessidade de modelos de linguagem que possam ser continuamente atualizados sem a necessidade de re-treinamento completo. A pesquisa em aprendizado contínuo e incremental está ganhando força, com o objetivo de criar chatbots que possam se adaptar a novas informações em tempo real. Um exemplo de aplicação é implementar chatbots em ambientes corporativos que possam incorporar novas políticas ou informações à medida que são introduzidas, mantendo a precisão e relevância das respostas.

Integração Multimodal Avançada: Os chatbots do futuro não se limitarão apenas ao texto. A integração de dados multimodais – combinando texto, áudio, vídeo e outras formas de dados sensoriais – permitirá que os chatbots ofereçam interações muito mais ricas e naturais. Um exemplo de aplicação seria desenvolver assistentes virtuais que possam interpretar e responder a comandos de voz, reconhecer expressões faciais em vídeo, e até mesmo responder a estímulos táteis em interfaces especializadas.

Desafios em abertos

Apesar dos avanços significativos, vários desafios permanecem no desenvolvimento de chatbots. Esses desafios representam oportunidades para pesquisa e inovação.

Compreensão de Contexto Profundo: Embora os modelos atuais sejam capazes de manter o contexto em conversas curtas, a compreensão de contexto em conversas longas e complexas ainda é um desafio. Isso inclui a capacidade de lembrar detalhes ao longo de várias interações e responder de maneira coerente

em tópicos que evoluem com o tempo. Uma área de pesquisa seria investigar novas arquiteturas de memória e mecanismos de atenção que possam melhorar a retenção e o uso de contexto em conversas prolongadas.

Interação Emocional e Comportamental: Os chatbots ainda carecem de habilidades avançadas para interpretar e responder a sinais emocionais e comportamentais dos usuários. A capacidade de um chatbot de ajustar seu tom, estilo de resposta e sugestões com base no estado emocional do usuário pode melhorar significativamente a qualidade da interação. Uma área de pesquisa seria desenvolver modelos de linguagem que incorporem reconhecimento e resposta emocional, utilizando técnicas de aprendizado profundo e processamento de sinais.

Redução de Viés e Aumento da Inclusividade: Como discutido na seção sobre considerações éticas, a mitigação de viés é um desafio contínuo. Garantir que os chatbots sejam inclusivos e justos em suas interações é essencial para evitar a perpetuação de preconceitos e discriminações. Uma área de pesquisa é criar métodos para detectar e corrigir viés em modelos de linguagem e explorar novos conjuntos de dados que representem uma diversidade maior de culturas e contextos.

Automação da Criação e Treinamento de Chatbots: O desenvolvimento e o treinamento de chatbots ainda são processos intensivos em tempo e recursos. Automação avançada nesses processos pode acelerar o desenvolvimento e permitir a criação de chatbots mais personalizados e especializados. Uma área de pesquisa é investigar o uso de técnicas de AutoML (Machine Learning Automático) para automatizar a seleção de modelos, ajuste de hiperparâmetros e treinamento de chatbots para diferentes domínios.

Oportunidades de P&D

Para desenvolvedores e pesquisadores interessados em contribuir para a pesquisa e desenvolvimento dos chatbots, as seguintes áreas podem representar algumas oportunidades:

Colaboração entre Humanos e Chatbots: A pesquisa em sistemas colaborativos, onde humanos e chatbots trabalham juntos para resolver problemas complexos, está crescendo. Explorar como chatbots podem complementar e aprimorar as capacidades humanas em ambientes colaborativos é uma área

rica para inovação. Um exemplo de aplicação seriam chatbots assistentes que ajudam equipes a coordenar projetos, fornecendo sugestões baseadas em análises de dados em tempo real e facilitando a comunicação entre os membros da equipe.

Chatbots para a Inclusão Digital: Com bilhões de pessoas ainda desconectadas do mundo digital, chatbots podem realizar inclusão digital. Desenvolver chatbots que funcionem em ambientes de baixa conectividade e que sejam acessíveis para pessoas com baixa alfabetização digital é uma área importante de pesquisa. Um exemplo seriam chatbots que operam em redes de baixa largura de banda e que utilizam interfaces de voz para alcançar comunidades rurais ou sub-representadas.

Segurança e Privacidade em Chatbots Autônomos: À medida que os chatbots se tornam mais autônomos, a segurança e a privacidade se tornam preocupações ainda maiores. A pesquisa em criptografia, autenticação e anonimização de dados em chatbots autônomos é essencial para proteger os usuários. Um exemplo de aplicação seria um chatbot financeiro que realiza transações automaticamente, protegendo as informações do usuário com criptografia avançada e garantindo a conformidade com regulamentos como o GDPR.

Colaboração Interdisciplinar: O futuro do desenvolvimento de chatbots será moldado pela colaboração interdisciplinar. As áreas de IA, psicologia, linguística, direito e ética precisarão trabalhar juntas para criar sistemas que sejam não apenas tecnicamente avançados, mas também socialmente responsáveis. Uma proposta pode ser desenvolver chatbots educacionais que não apenas ensinem, mas também sejam capazes de adaptar seus métodos pedagógicos com base em princípios psicológicos e educativos, colaborando com educadores e psicólogos.

Recomendações para Futuros Desenvolvedores

Para aqueles que desejam continuar explorando e inovando no campo dos chatbots, algumas recomendações:

- **Explore Novas Ferramentas e Tecnologias:** Mantenha-se atualizado sobre os últimos desenvolvimentos em PLN e IA, e não tenha medo de experimentar novas ferramentas e frameworks.

- **Concentre-se na Experiência do Usuário:** Ao desenvolver chatbots, sempre coloque a experiência do usuário em primeiro lugar. Um chatbot útil e intuitivo será muito mais bem-sucedido.
- **Mantenha a Ética em Mente:** Com o poder dos chatbots vem a responsabilidade. Certifique-se de que seu chatbot é ético, respeita a privacidade do usuário e está em conformidade com as regulamentações.
- **Participe da Comunidade:** Envolver-se com a comunidade de PLN e IA. Contribua com projetos de código aberto, participe de conferências e workshops, e colabore com outros pesquisadores e desenvolvedores.

5.8 Considerações Finais

As considerações éticas discutidas nesta seção são essenciais para o desenvolvimento de chatbots que respeitem os direitos e a dignidade dos usuários. Ao abordar questões de privacidade, vies algorítmico, segurança e impacto social, os desenvolvedores podem garantir que seus chatbots não apenas funcionem de maneira eficaz, mas também contribuam positivamente para a sociedade. Com um foco em práticas éticas, os chatbots têm o potencial de transformar interações humanas de forma significativa e benéfica.

Também revisamos as técnicas essenciais discutidas ao longo do livro e exploramos as tendências emergentes que estão moldando o futuro dos chatbots. À medida que continuamos a avançar no campo do Processamento de Linguagem Natural e Inteligência Artificial, as possibilidades para chatbots se expandem exponencialmente. Com as ferramentas e conhecimentos adquiridos neste livro, você está preparado para enfrentar esses desafios e contribuir para a próxima geração de sistemas de diálogo inteligentes. Além disso, destacamos as oportunidades de inovação e pesquisa futura no campo dos chatbots, apontando as tendências emergentes e os desafios que ainda precisam ser abordados.

Este livro explorou uma ampla gama de técnicas e conceitos essenciais para o desenvolvimento de chatbots modernos e eficazes. À medida que avançamos, a responsabilidade recai sobre os desenvolvedores, pesquisadores e inovadores para continuar explorando, questionando e expandindo os limites do que os chatbots podem alcançar.

O campo está repleto de oportunidades para aqueles que estão dispostos a enfrentar os desafios técnicos e éticos na construção dos chatbots. Com a combinação de técnica e responsabilidade social, o seu uso pode beneficiar a sociedade.

Convidamos os pesquisadores a continuar sua jornada no desenvolvimento de chatbots, contribuindo para a criação de sistemas de diálogo que não apenas resolvam problemas técnicos, mas também ajudem a construir um mundo melhor, explorando novas fronteiras, garantindo que os chatbots do futuro sejam mais capazes, inclusivos e responsáveis.

Sobre os autores

Giseldo da Silva Neo



Figura 5.1: Giseldo Neo

GISELDO DA SILVA NEO é Professor de Informática no Instituto Federal de Alagoas (IFAL) e desenvolve pesquisas na área de IA. Doutorando em Ciência da Computação na Universidade Federal de Campina Grande (UFCG). Possui Mestrado em Modelagem Computacional do Conhecimento (UFAL) e Mestrado em Contabilidade (FUCAPE). Possui MBA em Gestão e Estratégia Empresarial (ESTÁCIO), Especialização em Arquitetura e Engenharia de Software (ESTÁCIO),

MBA em Gestão de Projetos (UNINTER). Graduação em Análise e Desenvolvimento de Sistemas (ESTÁCIO) e Graduação em Processos Gerenciais (UNINTER) e possui nível Técnico em Informática (Escola Técnica Federal de Sergipe).

Alana Viana Borges da Silva Neo



Figura 5.2: Alana Neo

ALANA VIANA BORGES DA SILVA NEO é Professora de Informática no Instituto Federal do Mato Grosso do Sul (IFMS) e desenvolve pesquisas na área de Informática na Educação. Doutoranda em Ciência da Computação na Universidade Federal de Campina Grande (UFCG), Mestra em Modelagem Computacional

do Conhecimento na Universidade Federal de Alagoas (UFAL), Especialista em Estratégias Didáticas para a Educação Básica com Uso de TIC na Universidade Federal de Alagoas (UFAL), Especialista em Desenvolvimento de Software, Especialista em Segurança da Informação, Graduada em Análise e Desenvolvimento de Sistemas e Bacharel em Sistemas de Informação pela Universidade Estácio de Sá (ESTÁCIO) e Licenciatura em Computação pelo Claretiano Centro Universitário.

Contato

Caso deseje entrar em contato com os autores para reportar algum erro, crítica ou sugestão, envie e-mail para giseldo@gmail.com ou acesse o site <https://giseldo.github.io/>

Aviso Legal

As informações fornecidas neste trabalho são apenas para fins educacionais e informativos. Embora todos os esforços tenham sido feitos para garantir a precisão e a integridade, o autor e o editor não fazem declarações ou garantias de qualquer tipo, expressas ou implícitas, em relação à precisão, confiabilidade ou completude do conteúdo.

O autor e o editor não poderão ser responsabilizados por quaisquer danos ou perdas decorrentes do uso deste material. As opiniões expressas são de responsabilidade exclusiva do autor e não refletem necessariamente as opiniões de qualquer instituição ou organização com a qual o autor possa estar vinculado.

Uso da IA Generativa

Algumas partes textuais e algumas imagens foram criadas ou alteradas com várias das IAs Generativas disponíveis no momento da escrita. Porém, todo o texto foi revisado pelos autores e revisores.

Referências Bibliográficas

Sameera A. Abdul-Kader and John Woods. Survey on Chatbot Design Techniques in Speech Conversation Systems. *International Journal of Advanced Computer Science and Applications*, 6(7):72–80, 2015. ISSN 2158107X. doi: 10.14569/ijacsa.2015.060712.

N Akhila et al. Comparative study of bert models and roberta in transformer based question answering. In *2023 3rd International Conference on Intelligent Technologies (CONIT)*, pages 1–5. IEEE, 2023.

Anthropic. Model context protocol: Open standard for ai tooling integration. <https://modelcontextprotocol.io/>, 2024. Acessado em 2025-09-12.

Everton Moschen Bada. Uma proposta para extracao de perguntas e respostas de textos. *XVII Congresso Internacional de Informatica Educativa, TISE*, pages 44–49, 2012.

blipblog. Inteligência artificial, 2024. <https://www.take.net/blog/tecnologia/inteligenciaartificial/>.

Bhriguraj Borah, Dhrubajyoti Pathak, Priyankoo Sarmah, Bidisha Som, and Sukumar Nandi. Survey of textbased chatbot in perspective of recent technologies. In *International conference on computational intelligence, communications, and business analytics*, pages 84–96. Springer, 2018.

Noam Chomsky and David W Lightfoot. *Syntactic structures*. Walter de Gruyter, 2002.

- CrewAI. Crewai — the leading multi-agent platform. <https://www.crewai.com/>, 2025. Acesso em: 12 set. 2025.
- Arandir Cordeiro da Silva and Evandro De Barros Costa. Um Chatterbot aplicado ao Turismo: Um estudo de caso no Litoral Alagoano e na Cidade de Maceió. pages 160–167, 2007.
- Neo Giseldo da Silva, Otávio Monteiro Pereira, and Alana Viana Borges da Silva Neo. Engajando alunos na aula de filosofia com inteligência artificial. In *VII Semana Internacional de Pedagogia*, number 1, pages 1–12, Maceió - Alagoas - Brasil, 2020.
- Alana Viana Borges da Silva Neo, Giseldo da Silva Neo, Olival de Gusmão F., Wanderon R. M. Rodrigues, and José Antão Beltrão Moura. CHATCreator: Um construtor de chatbots de forma visual. *Sánchez, J. (2023) Editor. Nuevas Ideas en Informática Educativa, Volumen 17, p. 418 - 423. Santiago de Chile, 17: 418–523, 2023.*
- Ryan Ribeiro de Azevedo. *Um sistema de dialogo inteligente baseado em logica de descricoes*. PhD thesis, UFCG, 2015.
- Giovanni De Gasperis, Isabella Chiari, and Niva Florio. *AIML knowledge base construction from text corpora*, volume 427. 2013. ISBN 9783642296932. doi: 10.1007/978-3-642-29694-9_12.
- Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186, 2019.
- David Ferrucci. This Is Watson. *Journal of Research and Development*, 56(3):88, 2012.
- JSON-RPC Working Group. Json-rpc 2.0 specification. <https://www.jsonrpc.org/specification>, 2010. Acessado em 2025-09-12.
- Sviatlana Höhn. *Artificial Companion for Second Language Conversation*. 2019. ISBN 9783030155032.

Jacobstein, Murray, Sams, and Sincoff. A multi-agent associate system guide for a virtual collaboration center. *Proceedings of the International Conference on Virtual Worlds and Simulation Conference*, pages 215–220, 1998.

Hans Jonas. *O princípio responsabilidade: ensaio de uma ética para a civilização tecnológica*. Digitaliza Conteúdo, 2006.

Antonio Fernando Lavareda Jacob Junior. *Buti: um Companheiro Virtual baseado em Computacao Afetiva para Auxiliar na Manutencao da Saude Cardiovascular*. PhD thesis, 2008.

Daniel Jurafsky and James H Martin. *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition*, 2023.

Immanuel Kant. *Crítica da razão prática*. Editora Vozes Limitada, 2017.

Immanuel Kant. *Fundamentação da metafísica dos costumes*. Leya, 2023.

Rashid Khan and Anik Das. *Build Better Chatbots*. 2018. ISBN 9781484231104. doi: 10.1007/978-1-4842-3111-1.

Lorenz Cuno Klopfenstein, Saverio Delpriori, Silvia Malatini, and Alessandro Bogliolo. The Rise of Bots: A Survey of Conversational Interfaces, Patterns, and Paradigms. *Proceedings of the 2017 Conference on Designing Interactive Systems*, pages 555–565, 2017. doi: 10.1145/3064663.3064672.

Aliane Loureiro Krassmann, Fabrício Herpich, Álvaro Souza Pereira da Silva, Anita Raquel da Silva, Cristiane de Souza Abreu, Marcelo Augusto Rauh Schmitt, Magda Bercht, and Liane Margarida Rockenbach Tarouco. FastAIML: uma ferramenta para apoiar a geracao de base de conhecimento para chatbots educacionais. *Revista Novas Tecnologias na Educacao (RENOTE)*, 15(2):1–10, 2017. doi: 10.22456/1679-1916.79256.

Helton Kraus and Anita Fernandes. Ivetebyte: Um Chatterbot para Area Imobiliaria Integrando Raciocinio Baseado em Casos. *Revista Iberica de Sistemas e Tecnologias de Informacao*, (1):40–51, 2008.

Rupert Lane, Anthony Hay, Arthur Schwarz, David M. Berry, and Jeff Shrager. ELIZA Reanimated: The world's first chatbot restored on the world's first time

sharing system. pages 1–21, 2025. URL <http://arxiv.org/abs/2501.06707>.

Michelle Denise Leonhardt, Ricardo Neisse, and Liane Margarida Rockenbach Tarouco. MEARA: Um Chatterbot Temático para Uso em Ambiente Educacional, 2003. ISSN 2316-6533. URL <http://www.br-ie.org/pub/index.php/sbie/article/view/238>.

Jiwei Li, Will Monroe, Tianlin Shi, Sébastien Jean, Alan Ritter, and Dan Jurafsky. Adversarial Learning for Neural Dialogue Generation. pages 2157–2169, 2018. doi: 10.18653/v1/d17-1230.

Carlos Eduardo Teixeira Lima Lima. *Um Chatterbot para Criacao e Desenvolvimento de Ontologias com Logica de Descricao*. PhD thesis, 2017.

Rafael Luiz De Macedo and Elvis Fusco. An Intelligent Robotic Engine Using Digital Repository of the DSpace Platform. (c):17–23, 2014.

S. Madhumitha, B. Keerthana, and B. Hemalatha. Interactive Chatbot Using AIML. *International Journal Of Advanced Networking And Applications*, pages 217–223, 2015.

Mangaba AI. Mangaba ai — framework de automação com agentes inteligentes. <https://mangaba-ai.vercel.app/>, 2025. Acesso em: 12 set. 2025.

Francisco Supino Marcondes, Jose Joao Almeida, and Paulo Novais. A Short Survey on Chatbot Technology : Failure in Raising the State of the Art. pages 28–36, 2020. doi: 10.1007/978-3-030-23887-2.

Luís Antonio Marcuschi. *Análise da Conversação*. 1986.

Filomena Maria, Gonçalves Da, Silva Cordeiro, and Rodrigo Lins Rodrigues. Um ambiente virtual de aprendizagem apoiado por um agente virtual: chatterbot. *Encontro Internacional TIC e Educacao 2010 Lisboa Portugal*, 2010. ISSN 00351334 (ISSN).

Michael L Mauldin. CHATTERBOTS, TINYMUDS, and the Turing Test Entering the Loebner Prize Competition. *AAAI*, 94:16–21, 1994. URL <http://www.aaai.org/Papers/AAAI/1994/AAAI94-003.pdf>.

-
- Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
- Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large Language Models: A Survey. *arXiv*, 2025. URL <http://arxiv.org/abs/2503.23037>.
- Adiel Mittmann. *Implementação do chatterbot ELIZA na linguagem multiparadigma Oz*. PhD thesis, Universidade Federal de Santa Catarina, 2021.
- Alana Viana Borges S. Neo, Giseldo da Silva Neo, Ítalo Rodrigo S. Arruda, Jônatas Duarte Vital Leite, Ezequiel Santana S. Oliveira, Thallis Henrique F. da Silva, Olival de Gusmão F. Júnior, Evandro de Barros Costa, and Wanderson R. M. Rodrigues. ECOWÊ: Um chatbot que ensina ecologia. *XXVI Conferência Internacional sobre Informática na Educação (TISE)*, 17(December):518 – 523, 2023a.
- Giseldo S Neo, Evandro C Barros, Alana VBS Neo, J Antão B Moura, Arthur C Correia, and Wanderson RM Rodrigues. Integrando bpmn e aiml para construção de fluxos de diálogo para chatbots. In *Seminário Integrado de Software e Hardware (SEMISH)*, pages 356–367. SBC, 2023b.
- André M. M. Neves and Flávia de Almeida Barros. iAIML: Um mecanismo para tratamento de intencao em chatterbots. *Congresso da Sociedade Brasileira de Computacao*, pages 1032–1041, 2005.
- OpenAI. GPT-4 Technical Report. 4:1–100, 2023. URL <http://arxiv.org/abs/2303.08774>.
- Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.
- A. C. Pickover. *Artificial Intelligence: An Illustrated History: From Medieval Robots to Neural Networks*. Sterling Publishing Co., 2021.
- Jorge Guerra Pires. A conversational artificial intelligence based web application for medical conversations: a prototype for a chatbot. *medRxiv*, pages 2023–12, 2024.

- Alex Fernando Teixeira PRIMO and Luciano Roth COELHO. A chatterbot Cybelle: experiência pioneira no Brasil. *Mídia, textos e contextos. Porto Alegre: EDIPUCRS*, pages 259–276, 2001.
- Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI*, 2019.
- Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2020.
- Sumit Raj. *Building Chatbots with Python*. 2019. ISBN 9781484240953. doi: 10.1007/978-1-4842-4096-0.
- Kiran Ramesh, Surya Ravishankaran, Abhishek Joshi, and K. Chandrasekaran. A Survey of Design Techniques for Conversational Agents. In *International Conference on Information, Communication and Computing Technology*, volume 835, pages 336–350, 2017. ISBN 978-981-13-5991-0. doi: 10.1007/978-981-13-5992-7. URL https://link.springer.com/chapter/10.1007/978-981-10-6544-6_31.
- Sebastian Raschka. *Build a large language model (from scratch)*. Simon and Schuster, 2024.
- Eliéser Ribeiro. *A era da inteligência artificial e a nova economia*. Impressão sob demanda, Brasil, edição padrão edition, April 2025. Capa comum.
- Stuart Russel and Peter Norving. *Inteligência Artificial*. 2013. ISBN 9780136042594.
- Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. pages 2–6, 2019. URL <http://arxiv.org/abs/1910.01108>.
- Daniel Schiff, Aladdin Ayesh, Laura Musikanski, and John C Havens. leee 7010: A new standard for assessing the well-being implications of artificial intelligence. In *2020 IEEE international conference on systems, man, and cybernetics (SMC)*, pages 2746–2753. IEEE, 2020.

-
- Ayesha Shaikh, Geetanjali Phalke, Pranita Pat, Sangita Bhosale, and Jyoti Raghatwan. A Survey On Chatbot Conversational Systems. *International Journal of Engineering Science and Computing*, 6(11):3117–3119, 2016. URL <http://ijesc.org/upload/464758c5f7d1a1cd13085e8a584ec5f3.ASurveyOnChatbotConversationalSystems.pdf>.
- Moolchand Sharma, Shivang Verma, and Lakshay Sahni. Comparative Analysis of Chatbots. *SSRN Electronic Journal*, 2020. doi: 10.2139/ssrn.3563674.
- Heung-yeung Shum, Xiao-dong He, and Di Li. From Eliza to Xiaolce: challenges and opportunities with social chatbots. *Frontiers of Information Technology e Electronic Engineering*, 19(1):10–26, 2018. ISSN 2095-9184. doi: 10.1631/fitee.1700826.
- Abhishek Singh, Karthik Ramasubramanian, and Shrey Shivam. *Building an enterprise chatbot: Work with protected enterprise data using open source frameworks*. Apress, 2019.
- Paula Geralda Barbosa Coelho Torreao. *Project Management Knowledge Learning Environment: Ambiente Inteligente de Aprendizado para Educacao em Gerenciamento de Projetos*. PhD thesis, 2005.
- A M Turing. Computing Machinery and intelligence. *Mind*, 49(8):433–460, 1950. doi: 10.1016/B978-0-12-386980-7.50023-X.
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017. URL <https://arxiv.org/abs/1706.03762>.
- R Wallace. The Elements of AIML Style. *Alice AI Foundation, Inc*, pages 12–77, 2003. URL <https://files.ifi.uzh.ch/cl/hess/classes/seminare/chatbots/style.pdf>.
- Richard S Wallace. Don't Read Me: A.L.I.C.E. and AIML Documentation. pages 1–72, 2000.
- Richard S. Wallace. The anatomy of A.L.I.C.E. *Parsing the Turing Test*. Springer, Dordrecht, 2009.

Joseph Weizenbaum. ELIZA - A Computer Program For the Study of Natural Language Communication Between Man And Machine. *Communications of the ACM*, 9(1), 1966.

Wikipedia. Inteligência em abelhas, 2024a. https://pt.wikipedia.org/wiki/Inteligência_em_abelhas.

Wikipedia. Transformada de fourier, 2024b. <https://pt.wikipedia.org/wiki/TransformadadeFourier>.

Wikipedia. Humano, 2024c. <https://pt.wikipedia.org/wiki/Humano>.

Hiroshi Yamaguchi, Maxim Mozgovoy, and Anna Danielewicz-Betz. A Chatbot Based On AIML Rules Extracted From Twitter Dialogues. *Communication Papers of the 2018 Federated Conference on Computer Science and Information Systems*, 17:37–42, 2018. doi: 10.15439/2018f297.

Liping Zhao, Waad Alhoshan, Alessio Ferrari, Keletso J. Letsholo, Muideen A. Ajagbe, Erol Valeriu Chioasca, and Riza T. Batista-Navarro. Natural Language Processing (NLP) for requirements engineering: A systematic mapping study. *arXiv*, (v), 2020. ISSN 23318422.