sbbd:01

Chapter 1

Geração com Recuperação Aumentada (RAG) em Grafos de Conhecimento

Otávio Calaça Xavier, Anderson da Silva Soares

Abstract

This chapter addresses the integration of Knowledge Graphs with Retrieval-Augmented Generation (RAG) technology applied to Large Language Models (LLMs), exploring how this combination can enhance the natural language processing capabilities of AI systems. It discussed the fundamental structures of knowledge graphs, detailing the functioning and advantages of RAG technology, and presenting practical examples using tools such as LangChain and the Neo4j database. Moreover, advanced RAG techniques are explored, emphasizing the importance of effective pre- and post-processing strategies to maximize the relevance and accuracy of responses generated by modern AI systems.

Resumo

Este capítulo aborda a integração de Grafos de Conhecimento com a tecnologia de Geração com Recuperação Aumentada (RAG) aplicada a Modelos de Linguagem de Grande Escala (LLMs), explorando como essa combinação pode enriquecer a capacidade de processamento de linguagem natural dos sistemas de IA. Discutiu-se as estruturas fundamentais dos grafos de conhecimento, detalhando o funcionamento e as vantagens da tecnologia RAG, e apresentando exemplos práticos de uso de ferramentas como LangChain e o banco de dados Neo4j. Além disso, são exploradas técnicas avançadas de RAG, enfatizando a importância de estratégias eficazes de pré e pós-processamento para maximizar a relevância e a precisão das respostas geradas pelos sistemas de IA modernos.

1.1. Introdução aos Grafos de Conhecimento e Geração com Recuperação Aumentada (RAG)

Grafos de conhecimento são representações estruturadas que facilitam o armazenamento e a manipulação de informações complexas, onde entidades são mapeadas como nós e as relações entre elas como arestas [Hogan et al. 2021]. Essas estruturas são fundamentais

para modelar dados de forma que preservam suas interconexões semânticas, permitindo que sistemas computacionais interpretem as relações e propriedades das entidades de maneira contextualizada [Zou 2020]. O exemplo da Figura 1.1 apresenta a estrutura de um grafo de conhecimento simples. Nessa estrutura é possível observar dois tipos de nós: Pessoa e Interesse e dois tipos de arestas: INTERESSADA_EM e AMIGO_DE. Tanto as arestas quanto os nós podem conter propriedades.

Figure 1.1. Estrutura do grafo de conhecimento simples

Em um grafo de conhecimento, entidades são tipicamente objetos ou conceitos com existência física ou abstrata, tais como pessoas, locais, ou eventos, enquanto relações descrevem como essas entidades interagem ou estão conectadas [Suchanek et al. 2007]. As relações são explicitamente definidas, frequentemente como vértices dirigidos que ligam um par de nós, o que introduz uma dimensão de direcionalidade e semântica, crucial para inferências e análises subsequentes.

A aplicação de grafos de conhecimento na inteligência artificial (IA) oferece uma forma robusta de incorporar conhecimento contextual em sistemas automáticos, permitindo um raciocínio mais aprofundado e decisões baseadas em uma rica rede de informações interligadas [Paulheim 2017]. Grafos de conhecimento provam ser particularmente valiosos para enriquecer a inteligência artificial com capacidades de raciocínio complexo e adaptativo sobre vastos domínios de conhecimento [Zou 2020].

Utilizando grafos de conhecimento, sistemas de IA podem alcançar um entendimento contextual que é crítico para tarefas que requerem não apenas análise de dados isolados, mas uma interpretação holística das interações e relações. Por exemplo, na área de recomendações personalizadas, grafos de conhecimento permitem mapear preferências do usuário a produtos ou conteúdos de maneira dinâmica e interconectada, levando a recomendações mais precisas e personalizadas [Nickel et al. 2016].

A integração de grafos de conhecimento em modelos de aprendizado de máquina e deep learning proporciona uma base para a inclusão de conhecimento estruturado e semântico, o que é particularmente benéfico em tarefas de compreensão e geração de linguagem [Bordes et al. 2013]. Esta capacidade de incorporar e processar relações complexas dentro de um framework de IA permite que os sistemas não apenas respondam a consultas com maior precisão, mas também interajam de maneira mais natural e intuitiva com os usuários.

Modelos de Linguagem de Grande Escala (LLMs) como GPT-4[1], BERT[2] e Llama 3[3], são desenvolvidos para entender e gerar texto humano de forma que possam realizar tarefas variadas de processamento de linguagem natural, desde tradução até geração de conteúdo. Esses modelos são treinados em vastos conjuntos de dados textuais, absorvendo padrões linguísticos e nuances contextuais [Rogers et al. 2021]. No entanto, por serem primariamente treinados com dados históricos, eles frequentemente carecem de acesso a informações atualizadas ou específicas durante a geração de respostas.

A técnica de Geração com Recuperação Aumentada (RAG) surge como uma solução para a limitação dos LLMs em lidar com informações dinâmicas e específicas. Essencialmente, RAG é um método híbrido que combina as capacidades de geração de texto dos LLMs com sistemas de recuperação de informação. Isso permite que os modelos não apenas gerem texto baseados em seu treinamento prévio, mas também busquem e incorporem informações atualizadas de bases de dados externas durante a geração de texto [Lewis et al. 2020].

O funcionamento do RAG pode ser dividido em duas etapas principais: recuperação e geração. Primeiro, quando uma pergunta (*query*) é recebida, o componente de recuperação busca nos bancos de dados externos para encontrar as informações mais relevantes e atualizadas. Essas informações são geralmente estruturadas em formatos como grafos de conhecimento ou tabelas, que permitem rápida localização e extração de dados. Em seguida, esses dados são fornecidos ao modelo de linguagem, que integra as informações recuperadas com seu conhecimento pré-existente para gerar uma resposta coerente e contextualizada [Lewis et al. 2020].

A técnica RAG evoluiu rapidamente, apresentando paradigmas distintos como o *Naive RAG*, *Advanced RAG* e *Modular RAG* [Gao et al. 2023]. Cada um desses modelos traz melhorias incrementais sobre os seus predecessores, adaptando e refinando a interação entre os módulos de recuperação e geração para melhorar a qualidade e a relevância das respostas fornecidas pelos LLMs.

O Advanced RAG, em particular, aprimora significativamente o processo de recuperação ao introduzir estratégias de pré-recuperação e pós-recuperação, que refinam tanto a indexação quanto a incorporação de informações recuperadas antes da geração de texto. Essas inovações permitem que o RAG lide com tarefas mais complexas e conhecimento intensivo, superando assim as limitações do *Naive RAG* em contextos desafiadores [Gao et al. 2023].

Já o *Modular RAG* representa a mais recente inovação dentro deste campo, oferecendo uma abordagem altamente adaptável que permite a substituição e reconfiguração de módulos para enfrentar desafios específicos. Esta versatilidade torna o *Modular RAG* particularmente eficaz em ambientes dinâmicos e variados, proporcionando uma melhoria substancial sobre as abordagens anteriores [Gao et al. 2023].

[1]GPT é sigla para *Generative Pre-trained Transformer*. Mais detalhes sobre GPT-4 em: https://openai.com/index/gpt-4/

[2]BERT é sigla para *Bidirectional Encoder Representations from Transformers*[Devlin et al. 2018] e foi um dos primeiros modelos de linguagem baseados em *transformers*.

[3]Llama é sigla para *Large Language Model Meta AI*. Mais detalhes sobre Llama 3 em: https://llama.meta.com/

Integrando técnicas de RAG, os LLMs podem não apenas responder perguntas com maior precisão, mas também ajustar suas respostas com base em informações que são dinamicamente recuperadas e integradas. Isso não só aumenta a precisão das respostas geradas como também amplia significativamente a utilidade dos LLMs em aplicações do mundo real, marcando um avanço notável na interação entre recuperação de informações e geração de linguagem [Gao et al. 2023].

1.1.1. Introdução ao *framework* LangChain

O LangChain é um *framework* de código aberto em Python desenvolvida para a criação de aplicações que utilizam grandes modelos de linguagem (LLMs). Ele facilita todo o ciclo de vida de aplicativos baseados em LLM, desde o desenvolvimento até a produção e o lançamento [Pandya and Holia 2023].

Com o *framework* LangChain[4] é possível integrar LLMs com sistemas de recuperação de informações, facilitando o uso de dados estruturados durante a geração de texto. Desenvolvido para otimizar a interação entre LLMs e bases de dados, o LangChain suporta uma variedade de tecnologias de armazenamento de dados, permitindo consultas dinâmicas e a incorporação de informações externas em tempo real.

O LangChain apresenta uma arquitetura modular, que suporta a flexibilidade na configuração de conexões entre diversos LLMs e vários sistemas de armazenamento de dados, como bancos de dados SQL, NoSQL e sistemas baseados em grafos como o Neo4j. O *framework* oferece uma API que abstrai complexidades de consulta, permitindo que desenvolvedores e pesquisadores concentrem-se na lógica de aplicação sem a necessidade de detalhar as especificidades das linguagens de consulta de banco de dados.

No contexto de RAG, o LangChain facilita a execução de consultas complexas a partir de *prompts* de linguagem natural, traduzindo essas entradas para consultas estruturadas que são executadas em bases de dados externas. A informação recuperada é então sintetizada e incorporada pelo LLM para gerar respostas que são contextualmente relevantes. Esta capacidade é essencial para aplicações em que a atualidade e precisão dos dados são críticas, como em assistentes virtuais inteligentes e sistemas de suporte a decisões.

O LangChain suporta uma variedade de funcionalidades como *caching* de resultados, pré-processamento de dados e otimização de consultas, o que pode melhorar significativamente a eficiência das operações de recuperação de dados em aplicações de RAG. Além disso, o *framework* permite ajustes em tempo real dos parâmetros de consulta, o que aumenta a flexibilidade e adaptabilidade dos sistemas baseados em LLMs a mudanças em requisitos ou atualização constante dos dados.

Apesar de suas vantagens, a implementação de LangChain em sistemas de RAG apresenta desafios técnicos, incluindo a necessidade de garantir a segurança dos dados durante a transferência entre diferentes sistemas e a manutenção de latências baixas em consultas de alta complexidade. Não é escopo desde estudo focar no aprimoramento das capacidades de processamento paralelo do LangChain ou na otimização de algoritmos de indexação e recuperação para melhorar o desempenho e a escalabilidade de aplicações de

[4]Site oficial do LangChain: https://www.langchain.com/

RAG.

1.1.2. Introdução ao banco de dados Neo4j

Neo4j[5] é um banco de dados baseado em grafos, desenvolvido para facilitar consultas eficientes e a gestão de relações complexas entre dados. Diferentemente dos bancos de dados relacionais, que armazenam dados em tabelas, o Neo4j organiza dados em nós, arestas e propriedades, proporcionando uma representação mais intuitiva de redes de dados. Esta estrutura permite consultas que eficientemente percorrem grandes redes, explorando relações diretamente entre os objetos de interesse [Robinson et al. 2015].

A modelagem baseada em grafos do Neo4j oferece vantagens significativas, especialmente para aplicações que requerem intensa interação entre elementos conectados, como redes sociais, sistemas de recomendação e infraestruturas de segurança cibernética. As operações, como busca de caminhos mínimos, são otimizadas para serem rápidas e eficientes, o que não seria possível com a mesma eficiência em modelos relacionais ou hierárquicos devido ao custo associado a junções complexas e operações recursivas [Angles and Gutierrez 2008].

Neo4j utiliza a linguagem de consulta Cypher, base para o padrão ISO recentemente publicado GQL[6] (*Graph Query Language*), especificamente desenhada para facilitar a manipulação e recuperação de dados em grafos. Cypher é conhecida por sua expressividade e eficiência, permitindo que os usuários formulem consultas complexas de forma mais intuitiva. Com sintaxe declarativa, facilita significativamente o processo de descrever padrões nos grafos, extrair informações e modificar dados de grafos sem a necessidade de scripts complexos [Angles and Gutierrez 2008].

A flexibilidade do Neo4j para integrar-se com várias tecnologias modernas de análise de dados e inteligência artificial é uma de suas fortes vantagens. Ele pode ser utilizado junto com plataformas de processamento de dados *frameworks* de aprendizagem de máquina, permitindo a construção de sistemas inteligentes que adaptam e aprendem com a estrutura dos dados em grafos. Essa integração é crucial para desenvolver sistemas que não apenas reagem em tempo real, mas também evoluem com as mudanças nas relações de dados [Miller 2013].

Neo4j utiliza o modelo de grafos rotulados com propriedades (*labeled property graphs*), que enriquece os nós e arestas com rótulos e atributos, facilitando a representação detalhada de informações e a modelagem de relações complexas. Cada nó no Neo4j pode ter um ou mais rótulos, que definem seu tipo ou categoria, enquanto as arestas, que representam os relacionamentos, podem conter propriedades que detalham as características do relacionamento, como data, peso, direção ou qualquer outro metadado relevante. Esse modelo é altamente flexível e adaptável, adequado para representar estruturas de dados complexas, como os grafos de conhecimento [Angles et al. 2017].

Em contraste com os grafos rotulados com propriedades, outras tecnologias como o *Resource Description Framework* (RDF) e a *Web Ontology Language* (OWL) utilizam o conceito de triplas para armazenar dados. Uma tripla, no contexto de RDF, consiste em

[5] Site oficial do Neo4j: https://neo4j.com/
[6] ISO/IEC 39075:2024: Graph Query Language - GQL

um sujeito, um predicado e um objeto, e é usada para representar informações semânticas na web. OWL, por sua vez, utiliza RDF para criar ontologias que definem sistemas complexos de entidades e suas inter-relações com regras, normas e lógica, oferecendo uma camada adicional de formalismo e expressividade [Allemang and Hendler 2011].

Uma ontologia é um conjunto formal de conceitos dentro de um domínio e as relações entre esses conceitos. Ontologias desempenham um papel crucial na organização e na inferência de conhecimento em sistemas baseados em grafos. Ao definir um conjunto rigoroso de conceitos e categorias que descrevem um domínio específico, as ontologias permitem a aplicação de raciocínio lógico para deduzir novas informações e relações a partir dos dados existentes. Os grafos de conhecimento, como aqueles suportados pelo Neo4j, podem descrever ontologias, de forma a serem utilizadas para enriquecer a base de dados com metadados semânticos, melhorando a precisão das consultas e a relevância dos resultados [Allemang and Hendler 2011].

1.1.3. *Embeddings* e busca vetorial

Embeddings, ou representações vetoriais, são fundamentais no campo do processamento de linguagem natural (PLN). Algoritmos como Word2Vec [Mikolov et al. 2013], GloVe [Pennington et al. 2014], e BERT [Devlin et al. 2018] aprendem a codificar palavras e frases em vetores que capturam relações semânticas e sintáticas. Esses modelos são treinados em vastos corpora de texto, aprendendo a representar linguisticamente entidades como vetores em espaços dimensionais que refletem a proximidade semântica entre os conceitos. Modelos de linguagem mais recentes como Llama e GPT-4 também podem ser utilizados para geração de *embeddings*.

A busca vetorial utiliza medidas de similaridade, como a distância euclidiana ou a similaridade do cosseno, para identificar itens semanticamente próximos a partir de *embeddings*. No PLN, isso é aplicado em sistemas de recomendação e motores de busca para identificar conteúdo relacionado à consulta do usuário, mesmo sem correspondência direta de palavras-chave, demonstrando uma recuperação de informação mais flexível e contextualizada [Manning 2008].

Em RAG, *embeddings* facilitam a recuperação de conteúdos relevantes que informam a geração de texto subsequente. Ao converter consultas e conteúdos da base de dados em vetores de *embeddings*, sistemas de recuperação podem rapidamente comparar o a representação do texto informado na consulta com porções de texto disponíveis na base de dados. Assim, é possível identificar os segmentos mais pertinentes, enriquecendo a resposta gerada pela LLM com dados contextuais [Lewis et al. 2020].

A integração de técnicas de busca vetorial e *embeddings* em LLMs, como no RAG, melhora significativamente a capacidade dos modelos de responder perguntas complexas e realizar tarefas específicas de domínio. Por exemplo, em um cenário de assistente virtual, *embeddings* podem ajudar a refinar as respostas do assistente para que elas sejam mais personalizadas e informativas, baseando-se na análise vetorial de documentos de suporte em vez de apenas gerar respostas baseadas em treinamento prévio isolado [Henderson et al. 2017].

1.1.4. Configuração inicial do Neo4j e LangChain

O Neo4j é um banco de dados que pode ser instalado em máquina local, ou em um provedor de infraestrutura em nuvem. Para instalação local, o *download* do instalador pode ser realizado através do site oficial[7]. Para facilitar a configuração, será utilizado o Neo4j Aura[8], uma solução em nuvem gratuita para projetos iniciais que não requer instalação.

Como dito anteriormente, o LangChain é um framework Python. Logo, para sua instalação, pode-se utilizar o gerenciador de pacotes `pip`. Como ele será utilizado para conexão ao Neo4j, outros pacotes também são relevantes para instalação. A Figura 1.2 mostra o comando a ser executado para essa instalação. Visando facilitar a utilização do *framework* LangChain com Neo4j, será utilizado o Google Colab, um ambiente online onde é possível a execução de códigos Python sem a necessidade de instalação do interpretador da linguagem em ambiente local. Para isso, foi criado um arquivo *notebook*[9] com todos os códigos a serem executados e mais detalhes.

```
pip install langchain langchain-community neo4j
```

Figure 1.2. Instalação do LangChain e bibliotecas correlatas

Uma vez criado o banco de dados no Neo4j, será disponibilizada uma URI e senha. O valor padrão tanto para o banco de dados quanto para o usuário é `neo4j`. De posse dessas quatro informações (url, usuário, senha e banco de dados) é possível realizar a conexão ao Neo4j a partir do LangChain, conforme Figura 1.3.

```
1  from langchain_community.graphs import Neo4jGraph
2
3  NEO4J_URI = "<URI PARA SEU BANCO NEO4J>"
4  NEO4J_USERNAME = "neo4j"
5  NEO4J_PASSWORD = "<SENHA GERADA>"
6  NEO4J_DATABASE = "neo4j"
7
8  kg = Neo4jGraph(
9      url=NEO4J_URI, username=NEO4J_USERNAME,
10     password=NEO4J_PASSWORD, database=NEO4J_DATABASE
11 )
```

Figure 1.3. Conexão ao banco de dados Neo4j utilizando LangChain

[7] Site oficial para download do Neo4j: https://neo4j.com/download/

[8] Site oficial do Neo4j Aura: https://console.neo4j.io/?ref=aura-lp

[9] O *notebook* criado no Google Colab pode ser acessado em: https://colab.research.google.com/drive/1ZTdI5xWlMjjzPwrsx1q6SJxbOvm05H-j?usp=sharing

1.2. Introdução à linguagem Cypher

Cypher é a linguagem de consulta declarativa do Neo4j, projetada especificamente para trabalhar com dados em formato de grafo. Ela permite expressar o que recuperar do grafo, não como recuperar, o que facilita a focar na definição da lógica de negócios sem se preocupar com os detalhes de implementação subjacentes.

Uma consulta Cypher típica envolve especificar padrões em um grafo, que consistem em nós, arestas (relacionamentos) e propriedades. A sintaxe básica inclui:

- **Nós**: Representados por parênteses `()`, similar a um círculo no diagrama de grafo.
- **Relacionamentos**: Representados por setas -> ou <-, conectando nós. Os colchetes com dois pontos `[:]` são utilizados para nomear o relacionamento, por exemplo: `-[:CONHECE]->`.
- **Propriedades**: Valores armazenados nos nós ou relacionamentos, acessados através de chaves `{}`.

Para criar nós, é utilizada a palavra-chave **`CREATE`**. A Figura 1.4 apresenta um exemplo de criação de um nó do tipo `Pessoa` com o atributo `nome` igual a "João". Já a Figura 1.5 apresenta um exemplo mais completo. Neste exemplo, inicialmente é criado um nó do tipo `Pessoa` (`alice`) e outro do tipo `Interesse`. Em seguida é criado um relacionamento `INTERESSADA_EM` entre esses dois nós.

```
1 CREATE (joao:Pessoa {nome:"João"})
2 RETURN joao
```

Figure 1.4. Adicionando um novo nó do tipo Pessoa

```
1 CREATE (alice:Pessoa {nome: "Alice", idade: 30})
2 CREATE (neo4j:Interesse {nome: "Neo4j"})
3 CREATE (alice)-[:INTERESSADA_EM]->(neo4j)
```

Figure 1.5. Adicionando dois nós do tipo pessoa e uma aresta entre eles

Para executar comandos Cypher em um banco de dados Neo4j utilizando o LangChain, utiliza-se o método `query` do objeto `Neo4jGraph`. A Figura 1.6 mostra um exemplo de execução do código Cypher apresentado na Figura 1.5 utilizando o objeto `kg` criado na Figura 1.3.

```
1 kg.query("""
2 CREATE (alice:Pessoa {nome: "Alice", idade: 30})
3 CREATE (neo4j:Interesse {nome: "Neo4j"})
4 CREATE (alice)-[:INTERESSADA_EM]->(neo4j)
5 """)
```

Figure 1.6. Conexão ao banco de dados Neo4j utilizando LangChain

Supondo um banco de dados de grafos com informações sobre pessoas e seus interesses. Uma consulta nesse grafo para encontrar interesses específicos de uma pessoa poderia ser realizada como apresentado na Figura 1.7. No código apresentado:

- **MATCH**: Define o padrão para a consulta. Aqui, é procurado um nó de pessoa (Pessoa) com nome "Alice" que tenha um relacionamento INTERESSADO_EM com qualquer nó de interesse (Interesse).
- **RETURN**: Especifica o que retornar. Neste caso, o nome da pessoa e o nome de seus interesses.

```
1 MATCH (p:Pessoa {nome: "Alice"})-[:INTERESSADA_EM]->(i:Interesse)
2 RETURN p.nome, i.nome
```

Figure 1.7. Consulta Cypher para requisitar todos os interesses de uma pessoa chamada Alice

Também existe a palavra-chave **WHERE** para filtragem dos resultados. No exemplo da Figura 1.8 é utilizada a palavra-chave **WHERE** objetivando o mesmo resultado da consulta realizada na Figura 1.7.

```
1 MATCH (p:Pessoa)-[:INTERESSADA_EM]->(i:Interesse)
2 WHERE p.nome = "Alice"
3 RETURN p.nome, i.nome
```

Figure 1.8. Consulta Cypher para requisitar todos os interesses de uma pessoa chamada Alice utilizando WHERE

Um outro exemplo interessante, apresentado na Figura 1.9, é a consulta de múltiplos nós. Neste exemplo, a busca objetiva encontrar pessoas que compartilham os mesmos interesses.

```
1 MATCH (p1:Pessoa)-[:INTERESSADA_EM]->(i:Interesse)<-[:INTERESSADA_EM]-
2 (p2:Pessoa)
3 RETURN p1.nome, p2.nome, i.nome LIMIT 10
```

Figure 1.9. Consulta Cypher para encontrar pessoas que compartilham os mesmos interesses

Para excluir relacionamentos ou nós, a palavra-chave **DELETE** é utilizada, como observado na Figura 1.10. A modificação de nós ou relacionamentos pode ser realizada utilizando a palavra-chave **MERGE**. No exemplo da Figura 1.11, inicialmente é realizada uma consulta com match para pegar dois nós e em seguida eles são atualizados para a inclusão de um novo relacionamento entre eles.

```
MATCH (p:Pessoa {nome:"João"})-[rel:INTERESSADA_EM]->(i:Interesse)
DELETE rel
```

Figure 1.10. Excluindo um relacionamento utilizando DELETE

```
MATCH (joao:Pessoa {nome:"João"}), (alice:Pessoa {nome:"Alice"})
MERGE (joao)-[rel:AMIGOS_DE]->(alice)
RETURN joao, rel, alice
```

Figure 1.11. Utilizando MERGE para adicionar um relacionamento entre nós existentes

1.3. Construção e Indexação de Grafos de Conhecimento para RAG

A primeira etapa na construção de grafos de conhecimento no Neo4j é a modelagem de dados adequada. Esta fase envolve definir como as entidades e as relações serão representadas no grafo. Os modelos de dados em grafos, diferentemente dos modelos relacionais, são projetados para otimizar a conectividade e a eficiência das consultas relacionais, aproveitando estruturas que refletem diretamente as interações e relações naturais entre os dados [Robinson et al. 2015]. Por exemplo, em um contexto de mídia social, usuários, posts, e interações como "curtidas" e comentários podem ser diretamente mapeados para nós e relações em um grafo.

A extração de dados é o próximo passo crítico, envolvendo tanto fontes estruturadas quanto não estruturadas. Fontes estruturadas incluem bancos de dados relacionais e planilhas, enquanto fontes não estruturadas abrangem textos, imagens, e vídeos. Ferramentas como processadores de linguagem natural podem ser utilizadas para extrair entidades e suas relações de documentos de texto, que são então traduzidas em nós e arestas no Neo4j [Manning 2008]. Essa etapa é crucial para garantir que os dados sejam mapeados corretamente para o modelo de grafo.

Após a extração, os dados são integrados no Neo4j usando uma variedade de técnicas, dependendo da fonte dos dados. Para dados estruturados, ferramentas de ETL (*Extract, Transform, Load*) podem ser empregadas para migrar dados de sistemas tradicionais para o Neo4j. Para dados não estruturados, analizadores especializados e algoritmos de aprendizado de máquina preparam e formatam os dados para inserção no grafo [Angles and Gutierrez 2008]. A combinação da linguagem Cypher com as características do Python e LangChain pode ser muito apropriada para essa etapa.

Uma vez que os dados estão no Neo4j, a indexação e outras otimizações são essenciais para melhorar a performance das consultas. O Neo4j permite a criação de índices em

propriedades de nós e relações, o que pode significativamente acelerar as consultas, especialmente em grafos grandes e complexos. Além disso, o Neo4j suporta procedimentos armazenados que podem realizar operações complexas diretamente no banco de dados, reduzindo a necessidade de processamento externo [Miller 2013].

O suporte para indexação vetorial é particularmente valioso para tarefas que envolvem a comparação de similaridade entre entidades, como na busca de documentos, recomendação de conteúdo, ou agrupamento de itens semelhantes. Por exemplo, no contexto de um banco de dados de filmes, podemos utilizar o campo textual com a sinopse de cada filme para gerar *embeddings* que capturam o conteúdo semântico das descrições. Estes *embeddings* podem então ser indexados e utilizados para encontrar filmes com descrições semelhantes.

1.3.1. Exemplo prático de criação de grafo de conhecimento a partir de dados tabulares

O TMDB 5000 Movie Dataset[10], derivado do The Movie Database (TMDb), é uma compilação abrangente de dados sobre aproximadamente 5.000 filmes, utilizada extensivamente para análise e modelagem em diversos campos de pesquisa, incluindo ciência de dados, aprendizado de máquina e sistemas de recomendação. Este dataset inclui variáveis multidimensionais, tais como orçamento, receita, popularidade, gêneros, e sinopses, proporcionando uma rica fonte de informações para explorar padrões de mercado, comportamento do consumidor e preferências cinematográficas. Além disso, a disponibilidade de detalhes sobre produtoras, países de produção, datas de lançamento, e métricas de votação, entre outros, permite análises profundas de tendências de produção e consumo dentro da indústria cinematográfica.

Os dados TMDB 5000 são dispostos de maneira tabular, tendo alguns campos multivalorados e compostos, em formato JSON, como apresentado na Figura 1.12. A Figura 1.14 apresenta o código para carregar o arquivo CSV do dataset em um DataFrame do Pandas. O primeiro passo para transformar esse dataset em um grafo de conhecimento é criar os nós que representam os filmes (*Movie*). Cada linha do dataset será um nó do grafo. Em seguida, alguns campos multivalorados podem ser extraidos, como Gênero (*genre* e Palavras-Chaves (*keywords*). Visando a busca vetorial, o campo com a sinopse (*overview*) pode ser utilizado para geração dos *embeddings*. O código para execução de todas essas atividades pode ser visto na Figura 1.13.

Figure 1.12. Dados tabulares com campos do tipo JSON no TMDB 5000

	id	title	genres	keywords	budget	revenue	overview	tagline
0	19995	Avatar	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	[{"id": 1463, "name": "culture clash"}, {"id":...	237000000	2787965087	In the 22nd century, a paraplegic Marine is di...	Enter the World of Pandora.
1	285	Pirates of the Caribbean: At World's End	[{"id": 12, "name": "Adventure"}, {"id": 14, "...	[{"id": 270, "name": "ocean"}, {"id": 726, "na...	300000000	961000000	Captain Barbossa, long believed to be dead, ha...	At the end of the world, the adventure begins.
2	206647	Spectre	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	[{"id": 470, "name": "spy"}, {"id": 818, "name...	245000000	880674609	A cryptic message from Bond's past sends him o...	A Plan No One Escapes
3	49026	The Dark Knight Rises	[{"id": 28, "name": "Action"}, {"id": 80, "nam...	[{"id": 849, "name": "dc comics"}, {"id": 853,...	250000000	1084939099	Following the death of District Attorney Harve...	The Legend Ends
4	49529	John Carter	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	[{"id": 818, "name": "based on novel"}, {"id":...	260000000	284139100	John Carter is a war-weary, former military ca...	Lost in our world, found in another.
...	...	...	...	...	...	...	...	...
4798	9367	El Mariachi	[{"id": 28, "name": "Action"}, {"id": 80, "nam...	[{"id": 5616, "name": "united states\u2013mexi...	220000	2040920	El Mariachi just wants to play his guitar and ...	He didn't come looking for trouble, but troubl...
4799	72766	Newlyweds	[{"id": 35, "name": "Comedy"}, {"id": 10749, "...	[]	9000	0	A newlywed couple's honeymoon is upended by th...	A newlywed couple's honeymoon is upended by th...
4800	231617	Signed, Sealed, Delivered	[{"id": 35, "name": "Comedy"}, {"id": 18, "nam...	[{"id": 248, "name": "date"}, {"id": 699, "nam...	0	0	"Signed, Sealed, Delivered" introduces a dedic...	NaN
4801	126186	Shanghai Calling	[]	[]	0	0	When ambitious New York attorney Sam is sent t...	A New Yorker in Shanghai
4802	25975	My Date with Drew	[{"id": 99, "name": "Documentary"}]	[{"id": 1523, "name": "obsession"}, {"id": 224...	0	0	Ever since the second grade when he first saw ...	NaN

4803 rows × 8 columns

[10] Detalhes sobre o TMDB 5000 Movie Dataset podem ser acessados em: https://www.kaggle.com/datasets/tmdb/tmdb-movie-metadata. O dataset dos filmes está disponível em: https://raw.githubusercontent.com/otaviocx/datasets/main/tmdb$_5$000$_m$*ovies.csv*

```
from langchain_community.graphs import Neo4jGraph
from neo4j import GraphDatabase
import json

# Função para carregar dados do CSV para Neo4j
def load_data_to_neo4j(data, graph):
  # Criar nós e relações com base nos dados
  for index, row in data.iterrows():
    # Tratar campos JSON
    genres = json.loads(row['genres'].replace('\'', '"'))
    genre_names = [genre['name'] for genre in genres]
    keywords = json.loads(row['keywords'].replace('\'', '"'))
    keyword_names = [keyword['name'] for keyword in keywords]

    # Criar o nó do filme
    movie_query = """
    MERGE (movie:Movie {title: $title, id: $id})
    SET movie.budget = $budget, movie.revenue = $revenue,
        movie.overview = $overview, movie.tagline = $tagline
    """
    graph.query(movie_query, params={
      "title": row['title'],
      "id": row['id'],
      "budget": row['budget'],
      "revenue": row['revenue'],
      "overview": row['overview'],
      "tagline": row['tagline'],
    })

    # Criar nós de gêneros e relações
    for genre in genre_names:
      genre_query = """
      MATCH (movie:Movie {title: $movie_title})
      MERGE (genre:Genre {name: $genre_name})
      MERGE (movie)-[:HAS_GENRE]->(genre)
      """
      graph.query(genre_query, params={
        "genre_name": genre,
        "movie_title": row['title']
      })

    # Criar nós de palavras chaves e relações
    for keyword in keyword_names:
      keyword_query = """
      MATCH (movie:Movie {title: $movie_title})
      MERGE (keyword:Keyword {name: $keyword})
      MERGE (movie)-[:HAS_KEYWORD]->(keyword)
      """
      graph.query(keyword_query, params={
        "keyword": keyword,
        "movie_title": row['title']
      })

# Carregar dados
load_data_to_neo4j(data, kg)
```

Figure 1.13. Criando Grafo de Conhecimento apartir de dados tabulares do dataset TMDB 5000

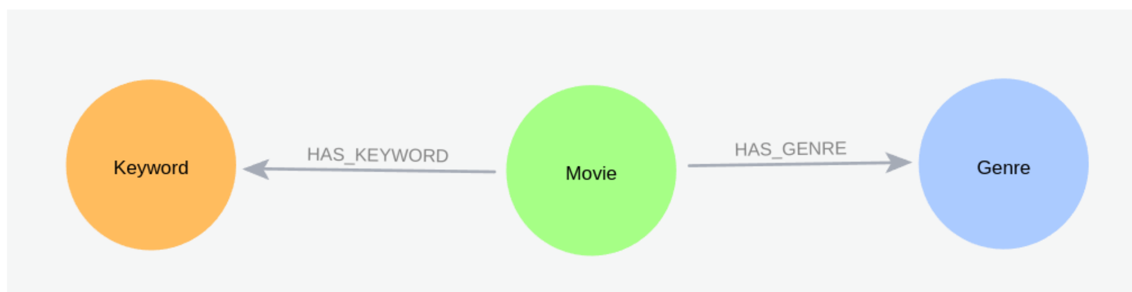
```
import pandas as pd

# Carregando o arquivo CSV para examinar a estrutura
repo = 'https://raw.githubusercontent.com/otaviocx'
file_path = f'{repo}/datasets/main/tmdb_5000_movies.csv'
data = pd.read_csv(file_path)
data
```

Figure 1.14. Carregando o arquivo tabular (CSV) do dataset TMDB 5000

Ao final da execução do código apresentado na Figura 1.13, o grafo criado no Neo4j exibirá a estrutura apresentada na Figura 1.15. O próximo passo é gerar os embeddings e armazená-los nos nós do tipo `Movie`. Para tal, será utilizada a biblioteca GPT4All[11]. Desenvolvida para facilitar a implementação e a utilização de um dos mais avançados modelos de linguagem disponíveis, a GPT4All oferece uma interface simplificada para interagir com o GPT-4, permitindo aos desenvolvedores e pesquisadores incorporar capacidades de compreensão e geração de texto em uma variedade de aplicações. Com essa biblioteca, é possível realizar tarefas complexas de processamento de linguagem natural, como resposta a perguntas, geração de texto, e tradução automática, com relativa facilidade e sem a necessidade de infraestrutura computacional intensiva tipicamente requerida por modelos de grande escala. A GPT4All democratiza o acesso a tecnologias de ponta em PLN, promovendo uma inclusão mais ampla de capacidades de inteligência artificial em aplicações comerciais e acadêmicas, e facilitando a experimentação e inovação no campo da inteligência artificial [Brown et al. 2020].

Figure 1.15. Estrutura do grafo de conhecimento criado com base no dataset TMDB 5000



O GPT4All pode ser carregado a partir do LangChain, conforme apresentado no código da Figura 1.16. Neste código, inicialmente é carregado um modelo de linguagem utilizado para gerar os embeddings do campo `overview` de cada filme no dataset. Ao final da execução desse código, a variável `embeddings` possuirá uma lista de vetores numéricos representando cada um dos filmes. Em seguida, o código da Figura 1.17 é executado, ele criará um novo índice vetorial na propriedade `embedding` dos nós do tipo `Movie`. Por fim, o código da Figura 1.18 adiciona os embbedings aos nós correspondentes.

[11] Site oficial do GPT4All: https://www.nomic.ai/gpt4all

```
from langchain_community.embeddings import GPT4AllEmbeddings

gpt4all_kwargs = {'allow_download': 'True'}
gpt4all_embd = GPT4AllEmbeddings(
    gpt4all_kwargs=gpt4all_kwargs
)

embeddings = gpt4all_embd.embed_documents(data['overview'].values)
embeddings
```

Figure 1.16. Gera os *embeddings* para o campo `overview` dos filmes.

```
CREATE VECTOR INDEX `movie_embedings` IF NOT EXISTS
FOR (m:Movie) ON (m.embedding)
OPTIONS { indexConfig: {
  `vector.dimensions`: 384,
  `vector.similarity_function`: "cosine"
}}
```

Figure 1.17. Adiciona os valores dos embeddings ao campo indexado `embedding` dos nós do tipo `Movie`

```
# Função para carregar embeddings nos nós do Neo4j
def load_embeddings_to_neo4j(data, graph):
  for index, row in data.iterrows():
    # Atualizar o nó do filme com embeddings
    movie_query = """
    MATCH (movie:Movie {id: $id})
    CALL db.create.setNodeVectorProperty(movie, "embedding", $embeddings)
    """
    graph.query(movie_query, params={
      "id": row['id'],
      "embeddings": row['embeddings']
    })

data['embeddings'] = embeddings
load_embeddings_to_neo4j(data, kg)
```

Figure 1.18. Adiciona os valores dos embeddings ao campo indexado `embedding` dos nós do tipo `Movie`

Com os *embeddings* gerados e atribuidos aos respectivos nós, é possível fazer uma consulta vetorial conforme apresentado na Figura 1.19.

```
def neo4j_vector_search(question):
  question_embedding = gpt4all_embd.embed_query(question)

  vector_search_query = """
    CALL db.index.vector.queryNodes(
      "movie_embedings",
      $top_k, $question_embedding
    ) yield node, score
    RETURN score, node.title as title, node.overview AS overview
  """
  similar = kg.query(vector_search_query,
                     params={
                      'question_embedding': question_embedding,
                      'top_k': 10})
  return similar

neo4j_vector_search(
    'Which movies are about love and action?'
)
```

Figure 1.19. Realiza busca vetorial por similaridade no campo `embedding`

1.4. Aplicação de RAG em Grafos de Conhecimento

Ao passo que os `embeddings` foram gerados e a busca vetorial é possivel de ser realizada, a parte de indexação e recuperação da informação está concluída. O próximo passo é a geração, com base na recuperação realizada. Para tal, uma forma simples, e frequentemente utilizada, de implementação de RAG é o complemento do prompt com as informações recuperadas a partir da busca vetorial.

No exemplo da Figura 1.20, estamos carregando um LLM utilizando o GPT4All para gerar respostas com base em seu treino. Nesse código não há a utilização do RAG. Na Figura 1.21 é possível observar um possivel resultado dessa execução. A resposta é inteiramente formada com base no pré-treino do modelo, não há definição de escopo nem é possível uma atualização da lista de filmes conhecidos.

Já o código da Figura 1.22 adiciona uma função para implementação do RAG. Nesta função, inicialmente é utilizada a pergunta original para buscar nós do grafo que possam ser relevantes para a resposta. Em seguida, o prompt (pergunta a ser enviada para o LLM) é composto com os dados obtidos pela busca vetorial. Assim, gerando uma resposta contextualizada com base nos dados que estão no grafo de conhecimento, concluindo a implementação do RAG. O resultado é apresentado na Figura 1.23. Nela é possível observar uma resposta mais direta e correlata com os filmes existentes no dataset.

```python
from langchain.chains import LLMChain
from langchain_community.llms import GPT4All
from langchain_core.prompts import PromptTemplate

template = """
{question}
"""

prompt = PromptTemplate.from_template(template)

llm = GPT4All(
  model="Meta-Llama-3-8B-Instruct.Q4_0.gguf",
  backend="gptj", verbose=True
)

llm_chain = LLMChain(prompt=prompt, llm=llm)

question = "Which movies are about love and action?"

llm_chain.run(question)
```

Figure 1.20. Consulta utilizando LLM sem RAG

```
There are plenty of movies that blend love and action. Here are a few
notable ones:

"Mr. & Mrs. Smith" (2005)
"True Lies" (1994)
"The Princess Bride" (1987)
"Romeo Must Die" (2000)
"The Adjustment Bureau" (2011)
"Kill Bill: Volume 1 & 2" (2003-2004)
```

Figure 1.21. Resultado da execução do LLM sem RAG conforme código da Figura 1.20

```
def print_answer(answer):
  response = ""
  for item in answer:
    response += f"""Score: {item["score"]}
Title: {item["title"]}
Overview: {item["overview"]}

"""
  return response

def ask_with_rag(question):
  retrieved_info = print_answer(neo4j_vector_search(question))
  template = f"""
  Based on the following context:
  {retrieved_info}
  Please answer (be objective, avoid introduction and
  conclusive texts): {{question}}
  """

  prompt = PromptTemplate.from_template(template)

  llm_chain = LLMChain(prompt=prompt, llm=llm)

  llm_chain.run(question)

ask_with_rag("Which movies are about love and action?")
```

Figure 1.22. Consulta utilizando LLM com RAG

```
The Lovers: Epic romance and time travel adventure with love
and action across different periods and continents.

Maleficent: Romance and action with a focus on Maleficent's journey
of betrayal and redemption.

Pirates of the Caribbean: On Stranger Tides: Romance and action
involving Jack Sparrow and a past love.

Spider-Man 2: Romance and action with Peter Parker struggling with his
superhero identity and feelings for Mary Jane Watson.
```

Figure 1.23. Resultado da execução do LLM com RAG conforme código da Figura 1.22

1.5. Técnicas Avançadas de Interligação e Consulta em Grafos de Conhecimento para *Advanced RAG*

Como visto, a tecnologia de Geração com Recuperação Aumentada (RAG), introduzida por [Lewis et al. 2020], é um paradigma inovador no campo do processamento de linguagem natural que combina técnicas de recuperação de informação com geração de linguagem baseada em modelos de linguagem. A abordagem avançada, conhecida como

Advanced RAG, procura ampliar este conceito integrando métodos mais sofisticados de recuperação e contextualização dos dados recuperados antes da geração de respostas. Isso pode inclui o uso de índices vetoriais construídos a partir de *embeddings* densos, que permitem consultas semânticas mais profundas e precisas, e a aplicação de mecanismos de atenção que direcionam o modelo de geração para considerar os segmentos mais relevantes do conteúdo recuperado. Essas inovações visam melhorar a qualidade e a relevância das respostas geradas, tornando o RAG uma ferramenta ainda mais poderosa para aplicações que vão desde assistentes virtuais até sistemas avançados de suporte à decisão.

O *Advanced RAG* incorpora técnicas de aprendizado de máquina para ajustar dinamicamente os métodos de recuperação baseados em feedback do usuário ou em análises de eficácia em tempo real. Por exemplo, estratégias de re-ranking baseadas em aprendizado de reforço podem ser utilizadas para ajustar a ordem dos nós recuperados conforme a interação do sistema com os usuários, melhorando assim a capacidade de adaptação e personalização do sistema. Este nível de personalização e interatividade abre novas possibilidades para a criação de sistemas de resposta a perguntas e de geração de conteúdo que são sensíveis ao contexto específico de suas aplicações, oferecendo respostas que são não apenas corretas, mas também contextualmente apropriadas e informativas.

Para implementar o *Advanced RAG*, necessariamente são necessarias etapas de pré e pós-processamento. Limpeza de dados, concatenação de atributos dos nós, filtragem com base da estrutura do grafo são exemplos de atividades que podem ser realizadas na etapa de pre-processamento. Já na etapa de pós-processamento, pode-se incluir calculo dos embeddings da resposta para comparação com os embeddings dos nós e da pergunta, limpeza da resposta pare remoção de texto desnecessário, entre outros.

Um exemplo um pouco mais avançado é apresentado na Figura 1.24. Inicialmente é gerado um template de prompt mais complexo, cujo foco é a construção de consultas diretamentas ao banco de dados Neo4j. Para tal, é utilizado o schema (estrutura) do banco de dados. A Figura 1.25 apresenta um exemplo de utilização dessa abordagem. Nota-se que a consulta retornada funciona perfeitamente no banco de dados. A Figura 1.26 já apresenta a execução de uma pergunta em linguagem natural diretamente no banco de dados. Observe que a consulta envolve 3 tipos de nós e 2 tipos de relacionamento. Essa execução direta deve ser realizada com cautela. Aconselha-se a adção de outros mecanismos de pós-processamento para evitar falhas de segurança.

```python
def generate_cypher(question):
  kg.refresh_schema()

  CYPHER_GENERATION_TEMPLATE = f"""Task:Generate Cypher statement to
  query a graph database.

  Instructions:
  Use only the provided relationship types and properties in the
  schema. Do not use any other relationship types or properties
  that are not provided.

  Schema:
  {kg.schema}
  Note: Do not include any explanations or apologies in
  your responses. Do not respond to any questions that
  might ask anything else than for you to construct a Cypher
  statement.
  Do not include any text except the generated Cypher statement.

  Examples: Here are a few examples of generated Cypher
  statements for particular questions:

  # Top 10 movies with biggest loss
  MATCH (m:Movie)
  WHERE m.budget > 0 AND m.revenue > 0
  WITH m, (m.budget - m.revenue) AS loss
  WHERE loss > 0
  RETURN m.title, m.budget, m.revenue, loss
  ORDER BY loss DESC
  LIMIT 10

  The question is:
  {{question}}"""

  prompt = PromptTemplate.from_template(CYPHER_GENERATION_TEMPLATE)

  llm_chain = LLMChain(prompt=prompt, llm=llm)

  llm_chain.run(question)
```

Figure 1.24. Função para geração de consultas Cypher a partir de linguagem natural

```python
generate_cypher("What are the Top 3 most expensive Action movies?")

# Saida retornada:
# MATCH (m:Movie)-[:HAS_GENRE]->(:Genre {name: "Action"})
# RETURN m.title, m.budget
# ORDER BY m.budget DESC
# LIMIT 3
```

Figure 1.25. Geração de consultas Cypher a partir de linguagem natural

```
kg.query(generate_cypher("""What are the Top 3 cheaper Adventure movies
which have the keyword ocean?"""))

# Saida Esperada:
# [{'m.title': 'Waterworld', 'm.budget': 175000000},
# {'m.title': "Pirates of the Caribbean: At World's End",
#   'm.budget': 300000000}]

# Consulta:
# MATCH (m:Movie)-[:HAS_GENRE]->(:Genre {name: "Adventure"})
# MATCH (m)-[:HAS_KEYWORD]->(:Keyword {name: "ocean"})
# RETURN m.title, m.budget
# ORDER BY m.budget ASC
# LIMIT 3

```

Figure 1.26. Consultando diretamente no banco de dados com linguagem natural

1.6. Conclusão

Este capítulo explorou conceitos fundamentais e aplicações práticas dos Grafos de Conhecimento e da tecnologia de Geração com Recuperação Aumentada (RAG) no contexto de Modelos de Linguagem de Grande Escala (LLMs). Iniciou-se com uma introdução aos Grafos de Conhecimento, destacando como essas estruturas facilitam a representação semântica e interconectada de informações, crucial para tarefas de IA que exigem um entendimento profundo das relações e contextos.

Em seguida foi detalhado como a tecnologia RAG melhora significativamente a funcionalidade dos LLMs, permitindo que esses modelos não só gerem textos com base em seu treinamento prévio, mas também busquem e integrem informações em tempo real durante a geração de respostas. Essa capacidade de atualização dinâmica é vital para aplicações que necessitam de informações precisas e contextualmente relevantes.

Adicionalmente, discutimos o uso prático do framework LangChain e do banco de dados Neo4j para implementar essas tecnologias. LangChain serve como uma ferramenta facilitadora para a integração de LLMs com bases de dados, enquanto Neo4j permite a manipulação eficiente e intuitiva de grafos de conhecimento, utilizando sua poderosa linguagem de consulta, Cypher.

O capítulo também abordou técnicas avançadas como o Advanced RAG, que introduz métodos refinados de pré e pós-recuperação para melhorar a precisão e a relevância das interações de LLMs. Estas técnicas representam a vanguarda do processamento de linguagem natural, combinando recuperação de informação com geração de linguagem de maneira inovadora.

1.6.1. Tópicos para Estudo Posterior

Para futuros estudos, seria benéfico explorar as seguintes áreas:

- Desenvolvimento de Modelos RAG Personalizados: Investigar como modelos personalizados de RAG podem ser desenvolvidos para atender necessidades específicas

de diferentes domínios de aplicação.

- Otimização de Performance em Grafos de Conhecimento: Estudar métodos para melhorar a eficiência de consultas em grafos de conhecimento, especialmente em bases de dados de grande escala.
- Segurança e Privacidade em RAG: Examinar as implicações de segurança e privacidade ao integrar RAG com LLMs, especialmente quando manipulando dados sensíveis.
- Integração de Novas Fontes de Dados em Grafos: Avaliar técnicas para integrar continuamente novas fontes de dados aos grafos de conhecimento, mantendo a consistência e a precisão das informações.
- Avaliação Comparativa de Frameworks e Ferramentas: Realizar estudos comparativos entre diferentes ferramentas e frameworks utilizados na implementação de RAG e grafos de conhecimento, identificando suas forças e limitações.

Aprofundar nestes tópicos permitirá não apenas uma melhor compreensão das tecnologias discutidas, mas também impulsionará inovações que podem transformar a maneira como interagimos e utilizamos sistemas baseados em IA e LLMs.

1.7. Sobre os autores

Otávio Calaça Xavier

Professor nas instituições UFG e IFG, cursando atualmente doutorado em Ciência da Computação, com foco em *Graph Neural Networks* e *Retrieval-Augmented Generation (RAG)*. Possui mestrado em Ciência da Computação pela UFG, obtido em 2011, com ênfase em Web Semântica e *Linked Data*. Com 19 anos de experiência em Desenvolvimento de Aplicações Web e Administração de Bancos de Dados, além de 15 anos em Arquitetura de Software e Liderança de Equipes. Desde 2007, atuou como palestrante em mais de 100 eventos de tecnologia em cidades como Goiânia, Brasília, São Paulo, Foz do Iguaçu, Rio de Janeiro e Porto Alegre. Leciona há mais de dez anos em cursos de graduação, pós-graduação e capacitação. Atua também como consultor em Aprendizado de Máquina, Ciência de Dados, Engenharia e Arquitetura de Software. Otávio é o autor que irá ministrar o mini-curso.

Anderson da Silva Soares

Professor do Instituto de Informática da Universidade Federal de Goiás, onde é membro permanente dos programas de mestrado e doutorado em Ciência da Computação. Foi vice-coordenador do programa de doutorado entre 2015 e 2016 e atuou como editor associado do *Journal of Computer Science* de 2015 a 2018. É um renomado pesquisador nas áreas de aprendizado de máquina, deep learning e otimização com heurísticas. Fundador do laboratório Deep Learning Brasil, Anderson também é presidente da comissão de criação e atual

coordenador do Bacharelado em Inteligência Artificial na UFG. Representante brasileiro na *Global Partnership on Artificial Intelligence (GPAI)*, realizou captações significativas de mais de 200 milhões de reais em P&D, proporcionando bolsas para alunos em projetos com empresas como Data-H, Copel Distribuição, iFood, entre outros. Foi fundador e diretor-geral do Centro de Excelência em Inteligência Artificial de Goiás (unidade Embrapii), onde atualmente atua como coordenador científico. Várias de suas iniciativas de P&D geraram startups *spin-offs*. Como atividade de extensão, coordenou a Olimpíada Brasileira de Robótica em Goiás e no Distrito Federal, e continua atuando como voluntário na organização. Além disso, é entusiasta e mantenedor do núcleo de robótica Pequi Mecânico, focado no nível universitário.

References

[Allemang and Hendler 2011] Allemang, D. and Hendler, J. (2011). *Semantic web for the working ontologist: effective modeling in RDFS and OWL*. Elsevier.

[Angles et al. 2017] Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., and Vrgoč, D. (2017). Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)*, 50(5):1–40.

[Angles and Gutierrez 2008] Angles, R. and Gutierrez, C. (2008). Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39.

[Bordes et al. 2013] Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., and Yakhnenko, O. (2013). Translating embeddings for modeling multi-relational data. *Advances in neural information processing systems*, 26.

[Brown et al. 2020] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

[Devlin et al. 2018] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

[Gao et al. 2023] Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., and Wang, H. (2023). Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*.

[Henderson et al. 2017] Henderson, M., Al-Rfou, R., Strope, B., Sung, Y.-H., Lukács, L., Guo, R., Kumar, S., Miklos, B., and Kurzweil, R. (2017). Efficient natural language response suggestion for smart reply. *arXiv preprint arXiv:1705.00652*.

[Hogan et al. 2021] Hogan, A., Blomqvist, E., Cochez, M., D'amato, C., Melo, G. D., Gutierrez, C., Kirrane, S., Gayo, J. E. L., Navigli, R., Neumaier, S., Ngomo, A.-C. N., Polleres, A., Rashid, S. M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S., and Zimmermann, A. (2021). Knowledge graphs. *ACM Comput. Surv.*, 54(4).

[Lewis et al. 2020] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.

[Manning 2008] Manning, C. D. (2008). *Introduction to information retrieval*. Syngress Publishing,.

[Mikolov et al. 2013] Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

[Miller 2013] Miller, J. J. (2013). Graph database applications and concepts with neo4j. In *Proceedings of the southern association for information systems conference, Atlanta, GA, USA*, volume 2324, pages 141–147.

[Nickel et al. 2016] Nickel, M., Murphy, K., Tresp, V., and Gabrilovich, E. (2016). A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33.

[Pandya and Holia 2023] Pandya, K. and Holia, M. (2023). Automating customer service using langchain: Building custom open-source gpt chatbot for organizations. *arXiv preprint arXiv:2310.05421*.

[Paulheim 2017] Paulheim, H. (2017). Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic web*, 8(3):489–508.

[Pennington et al. 2014] Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

[Robinson et al. 2015] Robinson, I., Webber, J., and Eifrem, E. (2015). *Graph databases: new opportunities for connected data*. " O'Reilly Media, Inc.".

[Rogers et al. 2021] Rogers, A., Kovaleva, O., and Rumshisky, A. (2021). A primer in bertology: What we know about how bert works. *Transactions of the Association for Computational Linguistics*, 8:842–866.

[Suchanek et al. 2007] Suchanek, F. M., Kasneci, G., and Weikum, G. (2007). Yago: a core of semantic knowledge. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, page 697–706, New York, NY, USA. Association for Computing Machinery.

[Zou 2020] Zou, X. (2020). A survey on application of knowledge graph. In *Journal of Physics: Conference Series*, volume 1487, page 012016. IOP Publishing.