



Pipeline RAG Semântico (SKOS + OWL + Reasoning + Azure AI Search)

Este notebook implementa um pipeline completo unificando:

- Extração de dados estruturados (planilha Contoso)
- Construção de um grafo RDF com OWL e SKOS
- Inferência semântica via Reasoning OWL-RL
- Integração RAG com Azure Cognitive Search (PDF)
- Agente inteligente combinando todas as fontes de conhecimento

Cada célula abaixo possui uma explicação detalhada do seu papel no pipeline.

0. Conexão com Azure OpenAI

Nesta célula realizamos a configuração inicial para acessar o modelo Azure OpenAI utilizado no agente.

A etapa inclui:

- Carregamento das variáveis do arquivo `.env`
- Inicialização do cliente AzureOpenAI
- Teste de conexão

Isso garante que todas as etapas seguintes do pipeline RAG (Reasoning + SKOS + PDF + Agente) possam enviar prompts para o modelo configurado.

```
from openai import AzureOpenAI
from dotenv import load_dotenv
import os

load_dotenv()

client = AzureOpenAI(
    api_key=os.getenv("AZURE_OPENAI_KEY"),
    azure_endpoint=os.getenv("AZURE_OPENAI_ENDPOINT"),
    api_version=os.getenv("AZURE_OPENAI_API_VERSION", "2024-12-01-preview")
)

print("🤖 · Azure · OpenAI · conectado!")
```

1. Extração do PDF (Azure Cognitive Search Simulation)

Nesta etapa realizamos a extração completa do conteúdo do PDF:

Objetivo desta célula:

- Carregar o arquivo PDF da análise estratégica da Contoso.
- Extrair todo o texto das páginas.
- Disponibilizar esse conteúdo como fonte não estruturada para o pipeline RAG.

Esse texto será usado posteriormente na função `agente_rag_semantico()`, permitindo que o agente responda perguntas combinando:

-  Conteúdo textual do PDF
-  Conhecimento semântico do grafo SKOS/OWL
-  Inferências feitas pelo reasoning
-  Modelo Azure OpenAI

Assim, o sistema funciona como um verdadeiro **RAG híbrido**, unificando dados estruturados e não estruturados.

```
from PyPDF2 import PdfReader
pdf_path = r"C:\Users\Microsoft\Windows\meu_agente_azure\data\analise_estrategica_contoso.pdf"
reader = PdfReader(pdf_path)
texto_pdf = "\n".join(p.extract_text() for p in reader.pages)
print("Páginas:", len(reader.pages))
```

[57]

... Páginas: 111

Python

2. Preparação das Páginas do PDF para Indexação

Nesta etapa, transformamos cada página do PDF em um documento independente.

Por que isso é importante?

- Azure Cognitive Search (ou qualquer motor de busca) funciona melhor quando cada página é um documento separado.
- Isso permite respostas mais precisas e segmentadas no pipeline RAG.
- Facilita o resgate de trechos por similaridade e contexto.

O que esta célula faz:

- Percorre todas as páginas do PDF.
- Extrai o texto corrigido de forma segura (`p.extract_text()`).
- Cria uma lista de dicionários contendo:
 - `id` da página
 - `número da página`
 - `conteúdo textual`

Esse vetor será mais tarde enviado ao mecanismo de busca para permitir consultas híbridas no agente.

```
docs = [
    {"id": f"page_{i+1}",
     "page": i + 1,
     "text": p.extract_text()} # <-- AQUI ESTÁ A CORREÇÃO
} for i, p in enumerate(reader.pages)]
```

```
len(docs)
```

[58]

... 111

Python

3. Criação do Índice no Azure Cognitive Search (para consultas RAG)

Agora que o PDF foi extraído e dividido em páginas, precisamos criar um índice no **Azure Cognitive Search** para permitir buscas eficientes e contextuais.

💡 Por que isso é essencial?

- O pipeline RAG depende de um mecanismo de busca para recuperar trechos relevantes.
- Cada página do PDF se torna um documento pesquisável.
- A consulta do agente pode buscar insights precisos dentro do conteúdo do relatório Contoso.

📌 O que esta célula faz:

- Conecta ao serviço Azure Cognitive Search.
- Define o nome do índice: `pdf-index`.
- Cria o schema do índice contendo:
 - `id` → identificador único de cada página
 - `page` → número da página (filtrável e ordenável)
 - `text` → conteúdo textual (campo pesquisável)

📈 Resultado:

Um índice totalmente funcional que será usado mais adiante pelo agente para enriquecer respostas com contexto real retirado do PDF.

```
from azure.search.documents.indexes import SearchIndexClient
from azure.search.documents.indexes.models import SearchIndex, SimpleField, SearchableField
from azure.core.credentials import AzureKeyCredential
import os

index_client = SearchIndexClient(
    endpoint=os.getenv("AZURE_SEARCH_ENDPOINT"),
    credential=AzureKeyCredential(os.getenv("AZURE_SEARCH_KEY"))
)

index_name = "pdf-index"

index_schema = SearchIndex(
    name=index_name,
    fields=[
        SimpleField(name="id", type="Edm.String", key=True),
        SimpleField(name="page", type="Edm.Int32", filterable=True, sortable=True),
        SearchableField(name="text", type="Edm.String")
    ]
)

index_client.create_or_update_index(index_schema)
print("✓ Índice criado:", index_name)
```

[59]

... ✓ Índice criado: pdf-index

Python



Imports necessários para criação do índice no Azure Cognitive Search

Nesta célula importamos todas as classes e módulos necessários para:

- Criar o cliente `SearchIndexClient`
- Definir o schema do índice (`SearchIndex`)
- Criar campos pesquisáveis (`SimpleField` e `SearchableField`)
- Conectar com a credencial segura do Azure (`AzureKeyCredential`)

Esses imports serão usados na próxima célula para criar o índice que armazenará as páginas do PDF.

```
from azure.search.documents.indexes import SearchIndexClient
from azure.search.documents.indexes.models import SearchIndex, SimpleField, SearchableField
from azure.core.credentials import AzureKeyCredential
import os
```

Python

🔗 4. Conexão ao Índice Criado no Azure Cognitive Search

Após criar o índice `pdf-index`, precisamos agora estabelecer uma conexão com ele para realizar:

- Upload dos documentos (páginas do PDF)
- Consultas de busca
- Recuperação de trechos relevantes durante o RAG

Nesta célula:

- Criamos um objeto `SearchClient`, responsável por enviar e buscar documentos.
- Conectamos diretamente ao índice `pdf-index` usando as credenciais do Azure.
- Validamos a conexão imprimindo uma mensagem de confirmação.

A partir deste ponto, o pipeline já está pronto para ingerir dados e realizar buscas semânticas no PDF.

```
from azure.search.documents import SearchClient  
  
search = SearchClient(  
    ... endpoint=os.getenv("AZURE_SEARCH_ENDPOINT"),  
    ... index_name="pdf-index",  
    ... credential=AzureKeyCredential(os.getenv("AZURE_SEARCH_KEY"))  
)  
print("🌐 Conectado ao índice pdf-index")
```

[61]

... 🌐 Conectado ao índice pdf-index

Python

5. Upload das Páginas do PDF para o Azure Cognitive Search

Com o índice `pdf-index` criado e a conexão estabelecida, agora enviamos os documentos (cada página do PDF) para o mecanismo de busca.

O que esta célula faz:

- Usa o `SearchClient` para fazer upload das páginas contidas em `docs`.
- Cada documento contém:
 - ID da página
 - Número da página
 - Texto extraído daquela página

Por que isso é importante?

Esse passo alimenta o mecanismo de busca com conteúdo do PDF. É esse material que será recuperado quando o agente fizer buscas via RAG, combinando:

- texto do PDF
- grafo SKOS/OWL
- reasoning
- Azure OpenAI

Após executar esta célula, todo o conteúdo do PDF ficará acessível para consultas semânticas.

% Generate + Code + Markdown

```
response = search.upload_documents(documents=docs)
response
```

[62]

Python

```
... [<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f020ad0>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f021400>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f021390>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f021080>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f021550>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f0216a0>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f0217f0>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f021710>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f021780>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f021860>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f0218d0>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f021940>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f0219b0>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f021a90>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f021a20>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f021b70>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f021b00>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f021be0>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f021c50>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f021cc0>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f021d30>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f021e10>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f021e80>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f021ef0>,
<azure.search.documents._generated.models._models_py3.IndexingResult at 0x22b7f021f60>,
```

6. Validação da Conexão com o Índice (Check de Segurança)

Antes de avançarmos para testes de busca, é importante confirmar que:

- O `SearchClient` está conectado ao endpoint correto
- O índice selecionado é de fato o `pdf-index`
- As credenciais foram carregadas corretamente

Esta célula recria o cliente e imprime o nome do índice interno do objeto, garantindo que todas as operações seguintes (buscas e RAG) estão sendo realizadas no índice correto.

Esse passo simples evita erros silenciosos como:

- consultas indo para um índice errado
- credenciais inválidas
- endpoint incorreto
- indexação perdida

```
search = SearchClient(  
    ... endpoint=os.getenv("AZURE_SEARCH_ENDPOINT"),  
    ... index_name="pdf-index",  
    ... credential=AzureKeyCredential(os.getenv("AZURE_SEARCH_KEY"))  
)  
print(search._index_name)
```

[65]

Python

... pdf-index

```
results = search.search("sul")  
for r in results:  
    ... print("Página:", r["page"])  
    ... print(r["text"][:300])  
    ... print("---")
```

[66]

Python

```
... Página: 4  
REGIÃO: SUL  
Performance da Região:  
A região Sul contribuiu com R$ 696,939.63 para a receita total.  
Isso representa 27.1% do faturamento corporativo.  
Foram realizadas 6497 transações nesta região.  
Produtos com Melhor Performance:  
Science: R$ 17,691.22 (2.5%)  
Southern: R$ 16,665.11 (2.4%)  
Anythi  
---
```

```
Página: 2  
RESUMO EXECUTIVO  
Performance Geral do Período:  
Receita Total: R$ 2,572,148.35  
Volume de Vendas: 23519 unidades  
Ticket Médio: R$ 5,144.30 por transação  
Produtos Ativos: 399 diferentes
```

```
Principais Insights:  
A região Sul lidera em receita, representando o maior potencial de mercado  
Produtos  
---
```

Página: 7



7. Carregamento da Planilha de Vendas (Fonte Estruturada)

Nesta etapa iniciamos a parte **estruturada** do pipeline RAG Semântico.

O objetivo desta célula é:

- Carregar a planilha de vendas da Contoso Retail (`planilha_equipe_dados.xlsx`)
- Transformar os dados brutos em um DataFrame pandas
- Exibir as primeiras linhas para garantir que tudo está correto

💡 Por que isso é importante?

Este dataset será a base para:

- Criar instâncias de vendas no grafo RDF
- Ligar vendas a categorias SKOS
- Definir propriedades OWL
- Permitir reasoning entre categorias, produtos e regiões

A partir daqui, sua ontologia começa a se conectar diretamente com dados reais.

```
import pandas as pd
excel_path = r"C:\Users\Microsoft Windows\meu_agente_azure\data\planilha_equipe_dados.xlsx"
df_data = pd.read_excel(excel_path)
print("✅ Planilha carregada!")
df_data.head()
```

[67] ... ✅ Planilha carregada!

ID	Produto	Categoria	Região	Quantidade Vendida	Receita	Data Venda	Meta KPI	
0	1	Attack	Brinquedos	Sul	56	4077.24	2025-09-15	1684.45
1	2	Decide	Roupas	Norte	98	4795.83	2025-05-07	1787.59
2	3	Media	Roupas	Oeste	61	7472.94	2025-10-12	1856.99
3	4	Moment	Alimentos	Norte	11	7435.40	2025-09-02	1113.71
4	5	Better	Brinquedos	Sul	15	8492.45	2025-02-25	3080.62

8. Construção Completa do Grafo Semântico (OWL + SKOS + RDF)

Nesta etapa criamos todo o grafo semântico que será utilizado pelo agente para realizar reasoning, inferência e navegação hierárquica entre conceitos.

Este é o coração semântico do pipeline.

◆ 1. Definição dos Namespaces e Inicialização do Grafo

Criamos um grafo RDF (`rdflib.Graph`) e associamos todos os namespaces usados:

- `ex:` domínio da Contoso
- `skos:` para taxonomia
- `owl:` para classes e propriedades ontológicas
- `rdfs:` para domínios, ranges e herança
- `dc:` e `foaf:` para padrões W3C

◆ 2. Definição das Classes OWL

As classes modelam os tipos fundamentais:

- `Venda`
- `Produto`
- `Regiao`
- `CategoriaVenda`

Isso torna o grafo compatível com reasoning OWL-RL, permitindo inferências posteriores.

◆ 3. Propriedades com Domínio e Range

Aqui definimos propriedades fundamentais do modelo semântico, como:

- `ex:temProduto`
- `ex:temCategoria`
- `ex:temRegiao`
- `ex:temReceita`
- ...

Cada propriedade recebe:

- `domain` (quem possui essa propriedade)
- `range` (tipos aceitos)
- definição automática como `ObjectProperty` ou `DatatypeProperty`

Isso é essencial para reasoning.

◆ 4. Criação do SKOS ConceptScheme

Criamos:

- `ex:CategoriasScheme`
Com seu rótulo e pertencimento ao domínio Contoso.

◆ 5. Geração Automática dos SKOS Concepts

Com base na planilha real:

- Cada categoria se torna um `skos:Concept`
- Recebe `skos:prefLabel`
- É adicionada ao ConceptScheme

Esta etapa conecta **dados reais** com **taxonomia semântica**.

◆ 6. Hierarquia SKOS Completa (broader/narrower)

Criamos:

- Categoria genérica `Produtos`
- Subcategorias reais (Alimentos, Eletrônicos, Brinquedos, Roupas)

E adicionamos:

- `skos:broader`
- `skos:narrower`

Essa bidirecionalidade permite reasoning e consultas hierárquicas.

◆ 7. Criação de Instâncias de Venda

Para cada linha da planilha:

- Criamos `ex:venda/{ID}`
- Ligamos atributos reais (produto, regiao, receita...)
- Associamos à categoria SKOS correspondente

Estas instâncias serão usadas:

- nas consultas semânticas
- no reasoning
- no agente final

✓ Resultado:

Ao final desta célula, você terá um grafo robusto contendo:

- taxonomia SKOS completa
- classes OWL
- propriedades ontológicas
- instâncias reais de vendas
- hierarquia necessária para reasoning

E tudo isso será usado no pipeline RAG semântico.

```
from rdflib import Graph, URIRef, Literal, RDF, RDFS, Namespace
from rdflib.namespace import FOAF, DC, SKOS, OWL
import pandas as pd

# Carregar planilha
df_data = pd.read_excel(r"C:\Users\Microsoft.Windows\meu_agente_azure\data\planilha_equipe_dados.xlsx")

# Namespace base
EX = Namespace("https://contoso.com/vendas/")

## Criar grafo RDF
g = Graph()
g.bind("ex", EX)
g.bind("foaf", FOAF)
g.bind("dc", DC)
g.bind("skos", SKOS)
g.bind("owl", OWL)
g.bind("rdfs", RDFS)

#####
# 1) CLASSES OWL
#####

g.add((EX.Venda, RDF.type, OWL.Class))
g.add((EX.Produto, RDF.type, OWL.Class))
g.add((EX.Regiao, RDF.type, OWL.Class))
g.add((EX.CategoriaVenda, RDF.type, OWL.Class))

#####
# 2) PROPRIEDADES COM DOMÍNIO E RANGE
#####

properties = {
    ... EX.temProduto: (EX.Venda, RDFS.Literal),
    ... EX.temCategoria: (EX.Venda, SKOS.Concept),
    ... EX.temRegiao: (EX.Venda, RDFS.Literal),
    ... EX.temQuantidadeVendida: (EX.Venda, RDFS.Literal),
    ... EX.temReceita: (EX.Venda, RDFS.Literal),
    ... EX.temDataVenda: (EX.Venda, RDFS.Literal),
    ... EX.temMetaKPI: (EX.Venda, RDFS.Literal),
}

for prop, (domain, range_) in properties.items():
    g.add((prop, RDF.type, OWL.ObjectProperty if range_ != RDFS.Literal else OWL.DatatypeProperty))
    g.add((prop, RDFS.domain, domain))
```

9. Serialização do Grafo Semântico (Exportação para arquivo .TTL)

Após construir todo o grafo RDF contendo:

- Classes OWL
- Propriedades com domínio e range
- Conceitos SKOS
- Hierarquia broader/narrower
- Instâncias de vendas
- Toda a integração semântica entre categorias e dados

Precisamos **exportar esse grafo** para um arquivo reutilizável.

Por que serializar o grafo?

- Permite visualizar e validar o grafo em ferramentas como Protégé, GraphDB e Stardog.
- Garante compatibilidade com outros sistemas semânticos.
- Facilita versão e armazenamento junto ao projeto (GitHub).
- É necessário para a etapa seguinte de Reasoning OWL-RL.
- Permite reuso em APIs, agentes externos e pipelines futuros.

Formato escolhido: Turtle (.ttl)

- Leitura mais amigável
- Amplamente usado em RDF e SKOS
- Ideal para depuração e documentação

O que esta célula faz

- Define o caminho de saída (`knowledge_graph.ttl`)
- Serializa o grafo inteiro no formato Turtle
- Confirma o total de triplas geradas

Esse arquivo é um dos artefatos mais importantes do projeto — é literalmente o **conhecimento estruturado do domínio Contoso**.

```
output_graph = r"C:\Users\Microsoft\Windows\meu_agente_azure\knowledge_graph.ttl"
g.serialize(output_graph, format="turtle")

print("SKU: Grafo SKOS/OWL salvo em:", output_graph)
print("SKU: Total de triplas no grafo salvo:", len(g))
```

Python

🧠 10. Aplicação de Reasoning OWL-RL (Inferência Automática de Conhecimento)

Depois de construir e exportar o grafo RDF, aplicamos **reasoning** OWL-RL, uma técnica essencial para enriquecer o grafo com novos conhecimentos derivados das regras lógicas do OWL.

O reasoning é o que transforma um grafo "estático" em um grafo **inteligente**, capaz de responder perguntas que não estavam explicitamente codificadas.

🔍 O que esta célula faz:

1. Carrega o grafo previamente serializado

A partir do arquivo `knowledge_graph.ttl`, reconstruímos o grafo base.

2. Aplica o reasoner OWL-RL

O `DeductiveClosure(OWLRL_Semantics).expand()`:

- infere novas relações SKOS (ex.: `broader` → `narrower`, `narrower` → `broader`)
- infere tipos de classes
- deduz propriedades baseadas em domínio e range
- descobre categorias superiores mesmo que não estejam explicitamente ligadas
- enriquece o grafo com tripas adicionais úteis para consultas complexas

3. Cria um grafo resultante (`g_reasoned`)

Este grafo contém **triplas explícitas + inferidas**.

💡 Benefícios do reasoning:

- Permite responder perguntas como:
 - "Eletrônicos está dentro de qual categoria geral?"
 - "Quais categorias são filhas de Produtos?"
 - "Quais vendas pertencem implicitamente a Produtos?"
- Permite que a função `agente_rag_semantico()` tenha acesso a relações que não existiam antes da inferência.
- Habilita navegação semântica real (*semantic traversal*).

📌 Nota importante:

Após aplicar o reasoning, definimos:

```
g_skos = g_reasoned
```

Ou seja, **todas as consultas do agente passam a usar o grafo enriquecido**, garantindo resultados inteligentes e consistentes.

```
from rdflib import Graph
from owlrl import DeductiveClosure, OWLRL_Semantics

print("● Criando grafo para reasoning...")
g_reasoned = Graph()
g_reasoned.parse(output_graph, .format='turtle')

print("● Aplicando reasoning-OWL-RL...")
DeductiveClosure(OWLRL_Semantics).expand(g_reasoned)

print("✓ Reasoning aplicado!")
print("■ Total de tripas inferidas:", len(g_reasoned))

# 🔍 -USAR O GRAFO INFERIDO NAS CONSULTAS
g_skos += g_reasoned
```

[71]

Python

```
...
● Criando grafo para reasoning...
● Aplicando reasoning OWL-RL...
✓ Reasoning aplicado!
■ Total de tripas inferidas: 9930
```

[Generate](#) [+ Code](#) [+ Markdown](#)



11. Função de Busca Semântica de Vendas (Reasoning + SKOS)

Esta função é responsável por realizar uma busca **realmente semântica** no grafo de vendas, combinando:

- o conceito SKOS consultado pelo usuário, e
- todos os conceitos **inferidos como narrower** pelo reasoner, mesmo que não estivessem explícitos no grafo original.

💡 Como funciona a busca semântica?

Quando um usuário faz uma pergunta — por exemplo:
“**Quais vendas pertencem à categoria Produtos?**”

O sistema deve retornar:

- vendas ligadas diretamente à categoria, e
- vendas ligadas às categorias filhas, como:
 - Eletrônicos
 - Alimentos
 - Brinquedos
 - Roupas

Mas essas relações indiretas só são possíveis porque:

- o SKOS tem **broader / narrower**
- o reasoner OWL-RL inferiu automaticamente essas conexões

💡 O que esta função faz:

1. Busca vendas diretamente associadas ao conceito

Usa `buscar_vendas_por_categoria(concept_uri)` para achar vendas explícitas.

2. Busca conceitos mais específicos (narrower)

Executa uma SPARQL para descobrir conceitos filhos inferidos:

```
?concept skos:narrower ?child
```

```
# 🔥 .Busca semântica::conceito + conceitos filhos (inferidos)
def vendas_semanticas(concept_uri):
    ...
    # Vendas ligadas diretamente ao conceito
    vendas = buscar_vendas_por_categoria(concept_uri)

    # Procurar "narrower"-inferidos pelo reasoner
    query = f"""
PREFIX skos:<http://www.w3.org/2004/02/skos/core#>
SELECT ?narrow WHERE {{
    <{concept_uri}> .skos:narrower .?narrow .
}}
"""

    for row in g_skos.query(query):
        vendas += buscar_vendas_por_categoria(row.narrow)

    return vendas
```

Python

12. Funções de Navegação Semântica no SKOS (Conceitos e Relações)

Nesta célula definimos as funções fundamentais responsáveis por navegar dentro da estrutura SKOS enriquecida pelo reasoning OWL-RL.

Essas funções são usadas diretamente pelo agente para:

- Identificar conceitos no grafo
- Descobrir relações semânticas
- Navegar entre níveis da hierarquia (broader/narrower)
- Obter contexto semântico antes da análise das vendas e do PDF

Usamos `g_skos`, o grafo após reasoning, garantindo que relações inferidas sejam encontradas.

1. Função `encontrar_conceito(label)`

Esta função:

- procura um SKOS:Concept cujo `skos:prefLabel` corresponda ao texto da pergunta do usuário
- faz busca *case-insensitive*
- retorna a URI do conceito encontrado

Exemplo:

- Pergunta: "Fale sobre produtos"
- Resultado: `ex:Categoria_Produtos`

Isto é crucial para transformar **línguagem natural** em **URI semântica**.

2. Função `conceitos_relacionados(concept_uri)`

Esta função retorna todos os relacionamentos SKOS:

- `broader` (conceito mais geral)
- `narrower` (conceitos específicos)
- `related` (associação contextual, se existir)

A consulta é feita por SPARQL, e retorna tipos e URLs, permitindo que o agente:

- compreenda a hierarquia
- explique a estrutura para o usuário
- navegue automaticamente entre categorias

Resultado

Essas funções permitem ao agente realizar tarefas como:

- "Quais categorias estão abaixo de Produtos?"
- "A que categoria Eletrônicos pertence?"
- "Mostre conceitos relacionados a Alimentos."
- "Quais conexões semânticas existem para Brinquedos?"

Sem essas funções, o agente não conseguiria interpretar o grafo.

```
from rdflib import Graph, Namespace
from rdflib.namespace import SKOS

EX = Namespace("https://contoso.com/vendas/")

# ▲ USE O GRAFO COM REASONING, NÃO o g original
g_skos = g_reasoned

# 🔍 Encontrar conceito SKOS pelo nome
def encontrar_conceito(label):
    query = f"""
    PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
    PREFIX ex: <https://contoso.com/vendas/>
    SELECT ?c WHERE {{
        ?c a skos:Concept ;
            skos:prefLabel ?lbl .
        FILTER(CONTAINS(LCASE(?lbl), LCASE("{label}")))
    }}
    """
    rows = list(g_skos.query(query))
    return rows[0][0] if rows else None

# 🔍 Extrair conceitos relacionados (broader, narrower, related)
def conceitos_relacionados(concept_uri):
    query = f"""
    PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
    SELECT ?rel ?type WHERE {{
        {{<{concept_uri}>} skos:broader ?rel .
        BIND("broader" AS ?type)
    }}
    UNION
    {{<{concept_uri}>} skos:narrower ?rel .
    BIND("narrower" AS ?type)
    }}
    UNION
    {{<{concept_uri}>} skos:related ?rel .
    BIND("related" AS ?type)
    }}
    """
    return [{"concept": str(r.rel), "tipo": r.type} for r in g_skos.query(query)]
```

13. Função de Busca Direta de Vendas por Categoria SKOS

Antes de realizar buscas semânticas (que envolvem reasoning e categorias mais gerais), precisamos de uma função que recupere as vendas diretamente associadas a uma categoria específica.

Esta célula implementa exatamente isso: a busca direta no grafo RDF.

Como funciona esta função?

A função `buscar_vendas_por_categoria()`:

1. Recebe a URI de um conceito SKOS (ex: `ex:Categoria_Eletrônicos`)

2. Executa uma consulta SPARQL para encontrar todas as instâncias:

- que são do tipo `ex:Venda`
- ligadas a essa categoria via `ex:temCategoria`

3. Para cada venda encontrada, extrai:

- produto
- região
- receita

4. Retorna uma lista estruturada de dicionários Python.

Por que isso é importante?

• É a base para o reasoning posterior:

A função `vendas_semanticas()` usa esta função para buscar:

- vendas diretas
- vendas inferidas via `narrower`

• Permite que o agente responda perguntas como:

- “Quais vendas pertencem à categoria Eletrônicos?”
- “Mostre vendas da categoria Alimentos na região Sul.”
- “Qual o desempenho comercial por categoria?”

• O resultado deste método será integrado ao modelo Azure OpenAI para análises e explicações.

Onde ela se encaixa no pipeline?

1. O PDF alimenta o mecanismo RAG

2. O grafo RDF/SKOS/OWL alimenta a camada semântica

3. O reasoning estende o grafo

4. Esta função traz os dados estruturados para o agente

5. A função `vendas_semanticas()` combina reasoning + busca direta

6. O agente entrega respostas inteligentes

Esta célula é um elo essencial do pipeline.

💡 Onde ela se encaixa no pipeline?

1. O PDF alimenta o mecanismo RAG
2. O grafo RDF/SKOS/OWL alimenta a camada semântica
3. O reasoning estende o grafo
- 4. Esta função traz os dados estruturados para o agente**
5. A função `vendas_semanticas()` combina reasoning + busca direta
6. O agente entrega respostas inteligentes

Esta célula é um elo essencial do pipeline.

```
#... -> .Buscar_vendas_por_categoria_SKOS_(direto)
def buscar_vendas_por_categoria(concept_uri):
    query = """
    PREFIX ex:<https://contoso.com/vendas/>
    SELECT ?v .?produto .?regiao .?receita WHERE .{
        ?v a ex:Venda ;
        ex:itemCategoria <{concept_uri}> ;
        ex:itemProduto ?produto ;
        ex:itemRegiao ?regiao ;
        ex:itemReceita ?receita .
    }
    """
    vendas = []
    for r in g_skos.query(query):
        vendas.append({
            "venda": str(r.v),
            "produto": str(r.produto),
            "regiao": str(r.regiao),
            "receita": float(r.receita)
        })
    return vendas
```

14. Mapeamento de Linguagem Natural para Conceitos SKOS

Esta função é responsável por transformar frases comuns escritas pelo usuário (em linguagem natural) em **URIs de conceitos SKOS** presentes no grafo.

Esse passo é essencial para que o agente consiga:

- interpretar perguntas abertas
- identificar categorias automaticamente
- conectar texto humano ao grafo semântico
- usar reasoning baseado na intenção do usuário

Como a função funciona:

1. Consulta todos os conceitos SKOS existentes

A função realiza uma SPARQL que retorna:

- URI do conceito
- Label (prefLabel) associado

2. Normaliza a frase do usuário

Converte tudo para `lowercase` para comparação robusta.

3. Procura labels dentro da frase

Exemplo:

- Frase: "Quais produtos existem na categoria Eletrônicos?"
- Label: "Eletrônicos"
- Retorna: `ex:Categoria_Eletrônicos`

4. Retorna a URI do conceito encontrado

Se nenhum conceito for identificado, retorna `None`.

Importância no pipeline semântico

Essa função é o ponto onde:

- o grafo SKOS
- o reasoning
- as consultas SPARQL
- e o modelo de linguagem

se encontram para construir a resposta final do agente.

Sem ela, o agente não conseguiria associar "o que o usuário escreveu" com "a estrutura da taxonomia".

Exemplos

Pergunta do usuário	Conceito detectado
"Fale sobre eletrônicos"	ex:Categoria_Eletrônicos
"Quais categorias existem em produtos?"	ex:Categoria_Produtos
"Mostre dados sobre alimentos"	ex:Categoria_Alimentos

Isso torna o agente muito mais natural, flexível e inteligente.

```
def encontrar_conceito_na_frase(frase):
    ... # pegar todos os conceitos SKOS existentes
    query = """
    PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
    SELECT ?c ?lbl WHERE {
        ...?c a skos:Concept ;
           skos:prefLabel ?lbl .
    }
    """

    conceitos = [(str(r.c), str(r.lbl)) for r in g_skos.query(query)]
    ...

    frase_lower = frase.lower()

    # tentar reconhecer o label dentro da frase
    for uri, label in conceitos:
        if label.lower() in frase_lower:
            return uri
    ...
    return None
```

[75]

Python

```
def buscar_pdf(q):
    results = search.search(q)
    return "\n".join([r["text"][:400] for r in results])
```

[76]

Python

15. Agente RAG Semântico (SKOS + OWL + Reasoner + Azure Search + LLM)

Esta é a função central do projeto:
o **agente RAG Semântico**, responsável por combinar:

-  **Ontologias** (OWL Classes e propriedades)
-  **Taxonomia SKOS**
-  **Reasoning OWL-RL para inferência automática**
-  **Azure Cognitive Search (PDF)**
-  **Modelo LLM da Azure OpenAI**
-  **Linguagem natural escrita pelo usuário**

Tudo isso em um único pipeline inteligente que interpreta intenção, navega no grafo, recupera conhecimento e sintetiza análises estratégicas.

O QUE ESTA FUNÇÃO FAZ

A função executa todas as etapas essenciais do pipeline RAG Semântico:

1. Interpretação da pergunta

Usa:

- `encontrar_conceito_na_frase()`
Para encontrar qual conceito SKOS está implicitamente presente no texto.

Exemplos:

Pergunta	Conceito detectado
"Fale sobre produtos"	Categoria_Produtos
"Insights sobre eletrônicos"	Categoria_Eletrônicos
"Quais dados existem para Alimentos?"	Categoria_Alimentos

Se nenhum conceito for encontrado, o agente continua, mas com aviso.

2. Recuperação de conceitos relacionados (grafo SKOS)

Usa `conceitos_relacionados()` para extrair:

- broader
- narrower
- related

Incluindo relações inferidas automaticamente pelo reasoner OWL-RL.

3. Recuperação semântica de vendas (OWL + SKOS + Reasoning)

Usa `vendas_semanticas()` para:

- encontrar vendas associadas diretamente ao conceito
- encontrar vendas associadas a todos os conceitos **narrower inferidos**
- consolidar dados

Exemplo:

Pergunta: "Produtos"

→ retorna vendas de:

- Eletrônicos
- Alimentos
- Roupas
- Brinquedos

Mesmo sem o usuário mencionar esses termos.

4. Recuperação de contexto no PDF (RAG)

Usa `buscar_pdf()` que consulta seu índice no Azure Cognitive Search.

O agente combina:

- insights do grafo
 - inferências do reasoning
 - contexto real do relatório PDF

■ 5. Gera um MEGA prompt estruturado

Com:

- Conceito SKOS
- Conceitos relacionados
- Vendas inferidas
- Contexto do PDF
- Instruções de resposta

O prompt é montado para gerar análises estratégicas completas.

■ 6. Chama Azure OpenAI para gerar a resposta final

O modelo converte conhecimento estruturado do grafo + insights do PDF em:

- análise coerente
- insights de negócio
- explicação causal
- narrativa clara e contextualizada

💡 Por que este agente é poderoso?

Ele une:

Camada	Tecnologia
Semântica formal	OWL + SKOS
Inferência	OWL-RL Reasoning
Recuperação	Azure Cognitive Search
Geração	Azure OpenAI
Interpretação	Linguagem natural + heurística inteligente
API	FastAPI + ngrok

O resultado é um agente capaz de:

- responder perguntas de negócio complexas
- navegar em hierarquias semânticas
- explicar por que inferiu algo
- utilizar documentos externos
- gerar análises estratégicas
- operar via API no Foundry

Tudo isso sem treinar nenhum modelo.

```

def agente_rag_semantico(pergunta):
    # =====
    # CASO ESPECIAL - usuário quer saber inferências do reasoning
    # =====
    pergunta_lower = pergunta.lower()

    if "inferiu" in pergunta_lower or "inferidas" in pergunta_lower or "reasoning" in pergunta_lower:
        inferidos = []

        # Buscar todas as categorias narrow de Produtos
        q = """
        PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
        SELECT ?child WHERE {
            <https://contoso.com/vendas/Categoria_Produtos> skos:narrower ?child .
        }
        """
        for row in g_skos.query(q):
            inferidos.append(str(row.child))

        lista = "\n".join(["f"- '{uri}' for uri in inferidos])

        return f"""

### 🔍 Conceitos inferidos automaticamente pelo Reasoner (OWL-RL)

O reasoner aplicou regras SKOS e inferiu automaticamente que a categoria **Produtos** possui as seguintes subcategorias:

{lista}

Essas inferências não existiam explicitamente no grafo original – foram deduzidas por:

- SKOS.broader → geração automática de SKOS.narrower
- Regras OWL-RL que completam hierarquias
- Propagação semântica por inferência lógica

Isso permite perguntas como:
- “Quais vendas estão ligadas a Produtos?”
- “Mostre toda a hierarquia de produtos”
- “Quais categorias pertencem a Produtos?”

E permite buscas semânticas que vão além das triplas originais.

"""

        # =====
        # 1) Buscar conceito SKOS correspondente à pergunta
        # =====
        try:
            conceito = encontrar_conceito_na_frase(pergunta)
        except NameError:
            return "✗ ERRO: A função encontrar_conceito_na_frase() não foi definida. Execute a célula dessa função primeiro."

        if not conceito:
            conceito_info = "✗ Nenhum conceito SKOS encontrado."
            relacionados_info = "-"
            vendas_info = "-"

        else:
            conceito_info = f"✓ Conceito encontrado: '{conceito}'"

```

```

    vendas_info = "-"

else:
    conceito_info = f"✓ Conceito encontrado: '{conceito}'"

# Conceitos relacionados (broader / narrower / related)
try:
    relacionados = conceitos_relacionados(conceito)
    if relacionados:
        relacionados_info = "\n".join(
            [f"- {r['tipo']} → {r['concept']}" for r in relacionados]
        )
    else:
        relacionados_info = "Nenhum conceito relacionado encontrado."
except Exception as e:
    relacionados_info = f"⚠ Erro ao consultar conceitos relacionados: {e}"

# Vendas inferidas semanticamente via reasoning
try:
    vendas = vendas_semanticas(conceito)
    if vendas:
        vendas_info = "\n".join([
            f"- Produto **{v['produto']}** | Região: **{v['regiao']}** | Receita: **R${v['receita']}**"
            for v in vendas
        ])
    else:
        vendas_info = "Nenhuma venda inferida pelo conceito (mesmo após reasoning)."
except Exception as e:
    vendas_info = f"⚠ Erro ao consultar vendas semânticas: {e}"

# =====
# 2) Buscar no Azure Search (PDF)
# =====
try:
    pdf_context = buscar_pdf(pergunta)
    if not pdf_context:
        pdf_context = "Nenhum trecho relevante encontrado no PDF."
except Exception as e:
    pdf_context = f"⚠ Erro ao consultar o PDF: {e}"

# =====
# 3) Criar o prompt para o modelo
# =====
prompt = """"
Você é um agente de inteligência semântica com **OWL + SKOS + Reasoning + RAG**.

### 🌐 Pergunta do usuário:
{pergunta}

---

## 💡 1. Conceito SKOS identificado
{conceito_info}

---

## 💡 2. Conceitos relacionados (hierarquia SKOS)
{relacionados_info}

---

## 💡 3. Vendas inferidas via Reasoning (broader/narrower)
{vendas_info}
"""

```

```
---
```

```
## 🌐 2. Conceitos relacionados (hierarquia SKOS)
{relacionados_info}
```

```
---
```

```
## 📈 3. Vendas inferidas via Reasoning (broader/narrower)
{vendas_info}
```

```
---
```

```
## 💡 4. Contexto do PDF (Azure Cognitive Search)
{pdf_context}
```

```
---
```

```
### 🎯 Tarefa
```

```
Combine todas as fontes (SKOS + OWL + Reasoner + PDF) para gerar uma resposta:
```

```
- clara
- analitica
- com insights estratégicos
- explicando **por que** esses resultados foram inferidos
```

```
Responda agora:
```

```
"""
# =====
# 4) Chamar Azure OpenAI
# =====
try:
    resp = client.chat.completions.create(
        model=os.getenv("AZURE_OPENAI_DEPLOYMENT_NAME"),
        messages=[{"role": "user", "content": prompt}],
        max_tokens=900
    )
    return resp.choices[0].message.content

except Exception as e:
    return f"X Erro ao consultar Azure OpenAI: {e}"
"""
```