Aplicações IA com LangChain

Asimov Academy

ASIMOV

Conteúdo

01. Aplicações de IA com Langchain

Olá, seja muito bem-vindo ao curso “Aplicações de IA com Langchain” oferecido pela Asimov Academy! Meu nome é Adriano Soares, e estou aqui para guiá-lo através deste fascinante mundo da Inteligência Artificial, especialmente focado no uso de Large Language Models (LLMs) e no framework Langchain.

O que é Langchain?

Langchain é um framework avançado destinado à criação de aplicações de Inteligência Artificial. Ele se destaca por simplificar o processo de desenvolvimento, permitindo que, com menos linhas de código, você possa construir aplicações robustas e inteligentes. Langchain é especialmente útil para trabalhar com dados em texto, utilizando os chamados Modelos de Linguagem de Grande Escala, ou LLMs, como são conhecidos.

Por que Langchain?

A escolha do Langchain como foco deste curso não é por acaso. Este framework oferece uma série de abstrações que facilitam a criação de aplicações complexas, permitindo que você se concentre mais na lógica de negócios e menos nos detalhes técnicos. Com Langchain, você pode esperar uma curva de aprendizado suave, mas com um potencial poderoso para aplicações práticas.

Estrutura do Curso

Este curso está estruturado para levá-lo desde os conceitos básicos até os mais avançados dentro do universo do Langchain e das aplicações de IA:

1. **Models**: Aprenderemos como acessar e interagir com modelos de linguagem.
2. **Prompt Templates**: Criaremos prompts eficazes para extrair respostas precisas dos modelos.
3. **Output Parsers**: Padronizaremos as respostas para manter a consistência das saídas.
4. **Memory**: Adicionaremos memória às interações, permitindo que o modelo lembre de conversas anteriores.
5. **Chains e Routers**: Exploraremos como encadear prompts e criar cadeias de roteamento para processos mais complexos.
6. **Retrieval Augmented Generation (RAG)**: Introduziremos técnicas avançadas para aumentar o poder dos modelos de LLM.

O que você ganhará com este curso?

Ao final deste curso, você terá uma compreensão sólida de como utilizar o Langchain para criar aplicações de IA inovadoras e valiosas. Você estará no “Edge do Conhecimento”, aplicando técnicas que estão na fronteira da tecnologia atual.

Espero que você aproveite cada aula e que este curso realmente ajude a transformar sua maneira de interagir com a tecnologia e a informação. Vamos juntos desbravar este novo mundo. Prepare-se para iniciar uma jornada incrível no desenvolvimento de aplicações de IA com Langchain!

Fique à vontade para explorar, perguntar e experimentar. Este é apenas o começo de sua aventura em IA com Langchain. Vamos lá para a primeira aula!

02. Models - Acessando modelos de linguagem

Então, amigos. Vamos começar a entender o nosso LangChain através de sua estrutura mais básica, os Models, a abstração que permite o acesso aos mais diversos modelos de linguagem diferentes. No LangChain, temos duas estruturas de Models, os LLMs e os ChatModels que entenderemos a seguir.

LLMs (Large Language Models)

Os LLMs são, sem dúvida, o coração do LangChain. Eles são estruturas padronizadas para acessar modelos de linguagens, modelos estes que recebem uma entrada de texto e produzem uma saída relevante também em texto. Importante ressaltar que o LangChain não cria seus próprios LLMs, mas oferece uma interface padronizada para interagir com uma variedade de modelos fornecidos por terceiros, como OpenAI, Cohere e Hugging Face.

Como interagir com um LLM?

Vamos começar dando um exemplo de acesso a um modelo LLM da OpenAI, chamado `gpt-3.5-turbo-instruct`:

```
from langchain_openai import OpenAI

llm = OpenAI(model='gpt-3.5-turbo-instruct')
```

Agora, vamos fazer uma chamada básica ao modelo utilizando o método *invoke*:

```
pergunta = 'Conte uma história breve sobre a jornada de aprender a programar'
print(llm.invoke(pergunta))
```

```
"'\\n\\nEra uma vez um jovem chamado Lucas que sempre teve interesse em tecnolog
```

E se quisermos uma resposta em tempo real, como se estivéssemos conversando diretamente com o modelo? Podemos usar o método `stream`:

```
for trecho in llm.stream(pergunta):
    print(trecho, end='')
```

Desta forma, o texto assim que gerado pelo modelo já é enviado para nossa aplicação, criando a mesma usabilidade de quando utilizamos a interface de um ChatGPT, por exemplo.

Chamadas simultâneas

E se precisarmos de respostas para várias perguntas ao mesmo tempo? O LangChain nos permite fazer isso de forma eficiente com o método `batch`:

```
perguntas = ['O que é o céu?', 'O que é a terra?', 'O que são as estrelas?']
respostas = llm.batch(perguntas)
for resposta in respostas:
    print(resposta)
```

```
['\\n\\nO céu é o espaço acima da superfície da Terra, onde se encontram as nuve
```

Podemos perceber que recebemos uma lista de respostas com tamanho três, cada item faz referência a uma das perguntas, tendo a mesma ordem da lista de perguntas original.

ChatModels

Agora, vamos explorar os ChatModels, que são uma evolução dos LLMs, projetados especificamente para conversações. Eles entendem e interpretam a conversa, funcionando com três tipos principais de mensagens: de sistema, de usuário e da IA.

Estrutura de um ChatModel

Veja como é fácil configurar e usar um ChatModel com o LangChain:

```
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage, SystemMessage

chat = ChatOpenAI(model='gpt-3.5-turbo-0125')

mensagens = [
    SystemMessage(content='Você é um assistente que conta piadas.'),
    HumanMessage(content='Quanto é 1 + 1?')
]
resposta = chat.invoke(mensagens)
print(resposta.content)
```

```
Depende, você está pedindo a resposta matemática ou a resposta engraçada?
```

Tipos de Mensagens

No mundo dos ChatModels, temos diferentes tipos de mensagens que podem ser usadas para interagir com o modelo:

- **HumanMessage**: Representa uma mensagem do usuário.
- **AIMessage**: Representa uma resposta do modelo.
- **SystemMessage**: Indica ao modelo como se comportar.
- **FunctionMessage**: Resultado de uma chamada de função específica.
- **ToolMessage**: Resultado de uma chamada de ferramenta específica.

Neste momento focaremos nas três primeiras. **FunctionMessage** e **ToolMessage** serão abordadas em cursos posteriores quando tratamos de *Tools* e *Agents* com LangChain.

Streaming de Mensagens

Assim como com os LLMs, podemos receber respostas em um fluxo contínuo, o que é ideal para simular uma conversa real:

```
for trecho in chat.stream(mensagens):
    print(trecho.content, end='')
```

Como você viu, tanto os LLMs quanto os ChatModels são ferramentas poderosas que nos permitem interagir com a linguagem de maneira sofisticada e eficiente. Nos próximos capítulos, vamos explorar os “Prompt Templates”, que nos ajudarão a moldar essas interações de forma ainda mais eficaz.

03. Models - Conceitos Avançados

No capítulo anterior, exploramos os fundamentos dos modelos de linguagem, como importá-los e instanciá-los. Agora, vamos mergulhar em técnicas mais sofisticadas que ampliarão nossa capacidade de interagir e manipular esses poderosos modelos.

Prompt Few-Shot

Uma técnica essencial que já discutimos em nosso curso de Engenharia de Prompt é o **Prompt Few-Shot**. Essa técnica envolve fornecer ao modelo exemplos específicos de perguntas e respostas para moldar a maneira como ele responde a consultas futuras. Isso é particularmente útil para alinhar as respostas do modelo com o tipo de saída que desejamos.

Vejamos um exemplo prático utilizando o `ChatOpenAI` da LangChain:

```
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage, AIMessage

chat = ChatOpenAI()

mensagens = [
    HumanMessage(content='Quanto é 1 + 1?'),
    AIMessage(content='2'),
    HumanMessage(content='Quanto é 10 * 5?'),
    AIMessage(content='50'),
    HumanMessage(content='Quanto é 10 + 3?'),
]

chat.invoke(mensagens)
```

Neste exemplo, alternamos entre `HumanMessage` e `AIMessage` para treinar o modelo sobre como esperamos que ele responda. Isso é crucial para garantir que o modelo não só entenda a pergunta, mas também responda no formato desejado.

Utilizando Outros Modelos

LangChain não se limita a um único provedor de modelo. Podemos acessar uma variedade de modelos de diferentes provedores, como Hugging Face, Cohere, Gemini, entre outros. Isso é feito de maneira padronizada, o que facilita a integração e o uso de diferentes modelos em nossas aplicações. ### Utilizando Modelos do Hugging Face

Por exemplo, para usar um modelo da Hugging Face:

```
from langchain_community.chat_models.huggingface import ChatHuggingFace
from langchain_community.llms.huggingface_endpoint import HuggingFaceEndpoint
```

```
modelo = 'mistralai/Mixtral-8x7B-Instruct-v0.1'
llm = HuggingFaceEndpoint(repo_id=modelo)
chat = ChatHuggingFace(llm=llm)

mensagens = [
    HumanMessage(content='Quanto é 1 + 1?'),
    AIMessage(content='2'),
    HumanMessage(content='Quanto é 10 * 5?'),
    AIMessage(content='50'),
    HumanMessage(content='Quanto é 10 + 3?'),
]

chat.invoke(mensagens)
```

Aqui, `ChatHuggingFace` utiliza `HuggingFaceEndpoint` para interagir com um modelo específico hospedado na Hugging Face, demonstrando a flexibilidade do LangChain em trabalhar com diferentes fontes de modelos.

Caching

O caching é uma técnica vital para otimizar o desempenho e reduzir custos operacionais, especialmente quando lidamos com modelos que requerem muitos recursos. LangChain oferece suporte para caching tanto em memória quanto usando SQLite, permitindo que resultados de consultas sejam reutilizados sem a necessidade de novas chamadas ao modelo.

Cache em Memória

```
from langchain.cache import InMemoryCache
from langchain.globals import set_llm_cache
from langchain_openai.chat_models import ChatOpenAI

chat = ChatOpenAI(model='gpt-3.5-turbo-0125')
set_llm_cache(InMemoryCache())

mensagens = [
    HumanMessage(content='Você é um assistente engraçado.'),
    HumanMessage(content='Quanto é 1 + 1?')
]

chat.invoke(mensagens)  # Primeira chamada, sem cache
chat.invoke(mensagens)  # Segunda chamada, com cache
```

Cache SQLite

Para aplicações que precisam de persistência de dados entre sessões, o caching via SQLite é uma excelente opção.

```
from langchain.cache import SQLiteCache
from langchain.globals import set_llm_cache

set_llm_cache(SQLiteCache(database_path='arquivos/langchain_cache_db.sqlite'))

chat.invoke(mensagens)  # Primeira chamada, sem cache
chat.invoke(mensagens)  # Segunda chamada, com cache
```

Este capítulo avançado sobre modelos em LangChain mostra como podemos manipular, interagir e otimizar o uso de modelos de linguagem para criar aplicações robustas e eficientes. Espero que essas técnicas enriqueçam sua jornada de aprendizado e abram novas possibilidades em seus projetos de IA. Continue explorando e experimentando!

04. Prompt Templates - Criando prompts para aplicações robustas

Vamos agora entender sobre a forma de criação de prompts com LangChain e como eles podem ser estruturados para criar interações robustas e eficientes com modelos de linguagem.

O que são Prompt Templates?

Prompt Templates são estruturas pré-construídas que facilitam a criação de prompts para interagir com modelos de linguagem. Pense neles como os esqueletos de uma aplicação, onde a estrutura básica está pronta e apenas os detalhes específicos precisam ser adicionados conforme a necessidade.

Um **prompt** para um modelo de linguagem é um conjunto de instruções ou entradas fornecidas por um usuário para guiar a resposta do modelo. Isso ajuda o modelo a entender o contexto e gerar uma saída baseada em linguagem que seja relevante e coerente, como responder a perguntas, completar frases ou participar de uma conversa.

Exemplo Básico de Prompt Template

Vamos começar com um exemplo simples usando o modelo LLM da OpenAI:

```
from langchain_openai.llms import OpenAI
llm = OpenAI()

from langchain.prompts import PromptTemplate

# Criando um Prompt Template
prompt_template = PromptTemplate.from_template('''
Responda a seguinte pergunta do usuário:
{pergunta}
''')

# Usando o Prompt Template
print(prompt_template.format(pergunta='O que é um buraco negro?'))
```

```
Responda a seguinte pergunta do usuário:
O que é um buraco negro?
```

Neste exemplo, `{pergunta}` é uma variável que será substituída pela pergunta real que queremos fazer ao modelo. O método `format` é usado para substituir essa variável pela pergunta específica.

Partial Variables para Valores Default

Podemos também definir variáveis parciais (Partial Variables) que têm valores padrão, mas que podem ser substituídos quando necessário:

```
prompt_template = PromptTemplate.from_template('''
Responda a seguinte pergunta do usuário em até {n_palavras} palavras:
{pergunta}
''', partial_variables={'n_palavras': 10})

# Formatando com a variável parcial
prompt_template.format(pergunta='O que é um buraco negro?')
```

```
'\nResponda a seguinte pergunta do usuário em até 10 palavras:\nO que é um burac
```

Se quiséssemos modificar o valor padrão, seria só informar este valor ao formatar o prompt:

```
prompt_template = PromptTemplate.from_template('''
Responda a seguinte pergunta do usuário em até {n_palavras} palavras:
{pergunta}
''', partial_variables={'n_palavras': 10})

# Formatando com a variável parcial
prompt_template.format(n_palavras=5, pergunta='O que é um buraco negro?')
```

```
'\nResponda a seguinte pergunta do usuário em até 5 palavras:\nO que é um buraco
```

Combinando Prompts (Composing)

Além disso, podemos compor múltiplos Prompt Templates para criar prompts mais complexos:

```
from langchain.prompts import PromptTemplate

template_word_count = PromptTemplate.from_template('Responda a pergunta em até {n_palavras}
↪  palavras.')
template_lingua = PromptTemplate.from_template('Retorne a resposta na {lingua}.')

template_final = (
    template_word_count
    + template_lingua
    + '\nResponda a pergunta seguinte seguindo as instruções: {pergunta}'
)

prompt = template_final.format(n_palavras=10, lingua='inglês', pergunta='O que é uma
↪  estrela?')
llm.invoke(prompt)
```

```
'\n\nA star is a luminous ball of gas.'
```

Prompt Templates para Chat Models

Quando trabalhamos com Chat Models, a estrutura dos prompts muda um pouco para acomodar o formato de mensagens:

```
from langchain.prompts import ChatPromptTemplate

chat_template = ChatPromptTemplate.from_template('Essa é a minha dúvida: {duvida}')
chat_template.format_messages(duvida='Quem sou eu?')
```

```
[HumanMessage(content='Essa é a minha dúvida: Quem sou eu?')]
```

Few-shot Prompting

Few-shot prompting é uma técnica poderosa onde fornecemos ao modelo exemplos de perguntas e respostas para ajudá-lo a entender melhor o contexto antes de fazer uma nova pergunta.

No nosso exemplo, temos um conjunto de perguntas e respostas já na forma que gostaríamos que o modelo raciocinasse. Queremos através dos nossos exemplos ensinar ao modelo uma forma específica de pensar para responder suas perguntas:

```
exemplos = [
    {"pergunta": "Quem viveu mais tempo, Muhammad Ali ou Alan Turing?",
     "resposta":
     """São necessárias perguntas de acompanhamento aqui: Sim.
Pergunta de acompanhamento: Quantos anos Muhammad Ali tinha quando morreu?
Resposta intermediária: Muhammad Ali tinha 74 anos quando morreu.
Pergunta de acompanhamento: Quantos anos Alan Turing tinha quando morreu?
Resposta intermediária: Alan Turing tinha 41 anos quando morreu.
Então a resposta final é: Muhammad Ali
""",
    },
    {"pergunta": "Quando nasceu o fundador do craigslist?",
     "resposta":
"""São necessárias perguntas de acompanhamento aqui: Sim.
Pergunta de acompanhamento: Quem foi o fundador do craigslist?
Resposta intermediária: O craigslist foi fundado por Craig Newmark.
Pergunta de acompanhamento: Quando nasceu Craig Newmark?
Resposta intermediária: Craig Newmark nasceu em 6 de dezembro de 1952.
Então a resposta final é: 6 de dezembro de 1952
""",
    },
    {"pergunta": "Quem foi o avô materno de George Washington?",
     "resposta":
"""São necessárias perguntas de acompanhamento aqui: Sim.
Pergunta de acompanhamento: Quem foi a mãe de George Washington?
Resposta intermediária: A mãe de George Washington foi Mary Ball Washington.
Pergunta de acompanhamento: Quem foi o pai de Mary Ball Washington?
Resposta intermediária: O pai de Mary Ball Washington foi Joseph Ball.
```

```
Então a resposta final é: Joseph Ball
""",
    }
]
```

Few-shot Prompting Utilizando LLMs

```
from langchain.prompts.few_shot import FewShotPromptTemplate
from langchain.prompts.prompt import PromptTemplate
from langchain_openai.llms import OpenAI

llm = OpenAI()

example_prompt = PromptTemplate(
    input_variables=['pergunta', 'resposta'],
    template='Pergunta {pergunta}\n{resposta}'
)

few_shot_template = FewShotPromptTemplate(
    examples=exemplos,
    example_prompt=example_prompt,
    suffix='Pergunta: {input}',
    input_variables=['input']
)

llm.invoke(few_shot_template.format(input='Quem fez mais gols, Romário ou Pelé?'))
```

Few-shot Prompting Utilizando ChatModels

```
from langchain.prompts.few_shot import FewShotChatMessagePromptTemplate
from langchain.prompts import ChatPromptTemplate
from langchain_openai.chat_models import ChatOpenAI

chat = ChatOpenAI()

example_prompt = ChatPromptTemplate.from_messages(
    [('human', '{pergunta}'),
     ('ai', '{resposta}')]
)

few_shot_template = FewShotChatMessagePromptTemplate(
    examples=exemplos,
    example_prompt=example_prompt
)

prompt_template = ChatPromptTemplate.from_messages(
    [
    few_shot_template,
    ('human', '{input}')
    ]
)
```

```
llm.invoke(prompt_template.format(input='Quem fez mais gols, Romário ou Pelé?'))
```

Espero que este capítulo tenha esclarecido o conceito e a aplicação de Prompt Templates em suas interações com modelos de linguagem. Com essas ferramentas, você está bem equipado para criar aplicações de IA robustas e eficientes.

05. Output Parsers - Formatando saídas

O que é um Output Parser?

Quando interagimos com um modelo de linguagem, especialmente em tarefas complexas, frequentemente precisamos que a saída seja mais do que apenas texto corrido. Precisamos que essa saída seja estruturada de maneira que possa ser facilmente interpretada e manipulada por scripts ou outras partes de nossa aplicação. É aqui que os **Output Parsers** entram em cena.

Os **Output Parsers** são responsáveis por pegar a saída bruta de um modelo de linguagem e transformá-la em um formato estruturado, como JSON, listas ou dicionários em Python. Isso é extremamente útil quando usamos modelos de linguagem para gerar dados estruturados a partir de texto não estruturado.

Por que usar Output Parsers?

Imagine que você tem uma review de um cliente sobre um produto e quer extrair informações específicas, como quantos dias levou para ser entregue ou as percepções sobre o valor do produto. Sem um Output Parser, você teria que processar manualmente o texto para extrair essas informações, o que não só é trabalhoso, mas também propenso a erros.

Com um Output Parser, você pode definir um esquema que descreve exatamente o que extrair e como formatar a saída. O modelo de linguagem então usa esse esquema para fornecer uma resposta estruturada que pode ser facilmente manipulada posteriormente.

Por exemplo, ao utilizar um Output Parser, você pode especificar que deseja extrair informações como:

- **Produto**: uma breve descrição do produto.
- **Entrega**: se o cliente ficou satisfeito com a entrega.
- **Satisfação**: a satisfação geral do cliente com a compra.

Dessa forma, a saída do modelo pode ser formatada em um objeto estruturado, como um dicionário ou uma classe Pydantic, que pode ser facilmente utilizado em outras partes da sua aplicação.

Estruturando saídas de chat - StrOutputParser

Mas antes de explorarmos casos mais complexos, vamos começar com o formatador mais simples do LangChain: o **StrOutputParser**. Ele é utilizado para converter saídas do modelo no formato de conversação para formato texto. Essa é uma atividade comum, considerando que a maior parte dos LLMs que utilizamos com LangChain são acessados através dos ChatModels.

Exemplo de uso do StrOutputParser

```
from langchain.prompts import ChatPromptTemplate

chat_template = ChatPromptTemplate.from_messages(
    [
        ('system', 'Você é um assistente engraçado e se chama {nome_assistente}'),
        ('human', '{pergunta}')
    ]
)

# Formatando a mensagem
chat_template.format_messages(nome_assistente='Asimo', pergunta='Qual o seu nome?')
```

A saída será:

```
[SystemMessage(content='Você é um assistente engraçado e se chama Asimo', additi
 HumanMessage(content='Qual o seu nome?', additional_kwargs={}, response_metadat
```

Agora, vamos invocar o modelo e obter a resposta:

```
from langchain_openai.chat_models import ChatOpenAI

chat = ChatOpenAI()
resposta = chat.invoke(prompt)
```

A resposta do modelo será:

```
AIMessage(content='Olá! Meu nome é Asimo. Como posso te ajudar hoje?', ...)
```

Usando StrOutputParser

Para formatar a saída, utilizamos o **StrOutputParser**:

```
from langchain_core.output_parsers import StrOutputParser

output_parser = StrOutputParser()
output_parser.invoke(resposta)
```

A saída será:

```
'Olá! Meu nome é Asimo. Como posso te ajudar hoje?'
```

Dando um spoiler de chains

Podemos encadear as operações utilizando o operador |:

```
chain = chat_template | chat | output_parser
chain.invoke({'nome_assistente': 'Asimo', 'pergunta':'Qual o seu nome?'})
```

A saída será:

```
'Olá! Meu nome é Asimo. Como posso te ajudar hoje?'
```

Isto que criamos aqui é um exemplo bem simples de chains utilizando a sintaxe do LCEL, e estes conceitos serão abordados com mais profundidade nos próximos capítulos da apostila. ## Estruturando saídas mais complexas - Pydantic

O que é Pydantic?

O **Pydantic** é uma biblioteca do Python que facilita a validação de dados e a criação de modelos de dados. Ele permite que os desenvolvedores definam classes que descrevem a estrutura dos dados esperados, incluindo tipos de dados, valores padrão e descrições. Com o Pydantic, é possível garantir que os dados recebidos estejam no formato correto e atendam a critérios específicos, o que ajuda a evitar erros e inconsistências. Além disso, o Pydantic oferece suporte à serialização e desserialização de dados, permitindo que objetos Python sejam facilmente convertidos para formatos como JSON e vice-versa. Essa funcionalidade é especialmente útil em aplicações que interagem com APIs ou que precisam manipular dados estruturados de forma eficiente.

Combinando Pydantic ao método with_structured_output()

Esta é a maneira mais fácil e confiável de obter saídas estruturadas. O método `with_structured_output()` é implementado para modelos que fornecem APIs nativas para estruturar saídas, como chamadas de ferramentas/funções ou modo JSON, e aproveita essas capacidades internamente.

Este método recebe um esquema como entrada, que especifica os nomes, tipos e descrições dos atributos desejados na saída, e este esquema pode (e deve) ser criado com a biblioteca Pydantic. O resultado ao utilizarmos with_structured_output é um objeto similar a um Runnable, exceto que, em vez de gerar strings ou mensagens, produz objetos correspondentes ao esquema fornecido.

Exemplo com Pydantic

```
from typing import Optional
from pydantic import BaseModel, Field

class Piada(BaseModel):
    """Piada para contar ao usuário"""
    introducao: str = Field(description='A introdução da piada')
    punchline: str = Field(description='A conclusão da piada')
    avaliacao: Optional[int] = Field(description='O quão engraçada é a piada de 1 a 10')

llm_estruturada = chat.with_structured_output(Piada)
resposta = llm_estruturada.invoke('Conte uma piada sobre gatinhos')
```

A saída será:

```
Piada(introducao='Por que os gatinhos são tão bons em jogos de tabuleiro?', punc
```

Entendo a classe em Pydantic

Vamos entender agora passo a passo a criação da classe Piada utilizando Pydantic, para que não fique dúvidas!

Primeiro, definimos uma nova classe chamada `Piada`, que herda de `BaseModel`, que é a base para a criação de nossos próprios modelos de dados. Esta herança transforma a classe `Piada` em um modelo de dados Pydantic, permitindo que os dados sejam validados automaticamente.

```
class Piada(BaseModel):
```

A docstring (texto entre aspas triplas abaixo do nome da classe) que acompanha a classe descreve seu propósito, que é representar uma piada a ser contada ao usuário.

```
class Piada(BaseModel):
    """Piada para contar ao usuário"""
```

Dentro da classe, definimos três campos: `introducao`, `punchline` e `avaliacao`. O campo `introducao` é do tipo string (`str`) e é utilizado para armazenar a introdução da piada. Utilizamos `Field` para adicionar uma descrição a esse campo, o que é útil para o modelo de linguagem compreender o funcionamento deste campo. O mesmo se aplica ao campo `punchline`, que também é uma string e representa a conclusão da piada.

```
from typing import Optional
from pydantic import BaseModel, Field

class Piada(BaseModel):
    """Piada para contar ao usuário"""
    introducao: str = Field(description='A introdução da piada')
    punchline: str = Field(description='A conclusão da piada')
```

Por fim, temos o campo `avaliacao`, que é do tipo `Optional[int]`. Isso significa que ele pode ser um número inteiro ou None, tornando-o um campo opcional. Este campo é utilizado para armazenar uma avaliação da piada em uma escala de 1 a 10, e sua descrição fornece contexto sobre o que se espera nesse campo.

```
class Piada(BaseModel):
    """Piada para contar ao usuário"""
    introducao: str = Field(description='A introdução da piada')
    punchline: str = Field(description='A conclusão da piada')
    avaliacao: Optional[int] = Field(description='O quão engraçada é a piada de 1 a 10')
```

Um exemplo mais prático

Digamos que temos a seguinte review de um produto:

> "Este soprador de folhas é bastante incrível. Ele tem quatro configurações: sopro de vela, brisa suave, cidade ventosa e tornado. Chegou em dois dias, bem a tempo para o presente de aniversário da minha esposa. Acho que minha esposa gostou tanto que ficou sem palavras. Até agora, fui o único a usá-lo, e tenho usado em todas as manhãs alternadas para limpar as folhas do nosso gramado. É um pouco mais caro do que os outros sopradores de folhas disponíveis no mercado, mas acho que vale a pena pelas características extras."

E eu quero que o modelo de linguagem processe esta review para estruturá-la no seguinte formato:

```
{
  "presente": true,
  "dias_entrega": 2,
  "percepcao_de_valor": ["um pouco mais caro do que os outros sopradores de folhas disponíveis
  ↪  no mercado"]
}
```

Implementação

```
review_cliente = """Este soprador de folhas é bastante incrível. Ele tem
quatro configurações: sopro de vela, brisa suave, cidade ventosa
e tornado. Chegou em dois dias, bem a tempo para o presente de
aniversário da minha esposa. Acho que minha esposa gostou tanto
que ficou sem palavras. Até agora, fui o único a usá-lo, e tenho
usado em todas as manhãs alternadas para limpar as folhas do
nosso gramado. É um pouco mais caro do que os outros sopradores
de folhas disponíveis no mercado, mas acho que vale a pena pelas
características extras."""

from pydantic import BaseModel, Field

class AvaliacaoReview(BaseModel):
```

```
    """Avalia review do cliente"""
    presente: bool = Field(description='Verdadeiro se foi para presente e False se não foi')
    dias_entrega: int = Field(description='Quantos dias para entrega do produto')
    percepcao_valor: list[str] = Field(description='Extraia qualquer frase sobre o valor ou
↪   preço do produto. Retorne uma lista.')

llm_estruturada = chat.with_structured_output(AvaliacaoReview)
resposta = llm_estruturada.invoke(review_cliente)
```

A saída será:

```
AvaliacaoReview(presente=True, dias_entrega=2, percepcao_valor=['um pouco mais c
```

E com isso, espero que você tenha percebido que os *Output Parsers* são ferramentas poderosas que nos ajudam a tirar o máximo proveito dos modelos de linguagem. Na próximo capítulo falaremos sobre as chains e começaremos a integrar os Output Parser às nossas aplicações!

Desafio de OutputParser

Para o mesmo exemplo visto em aula da review de um cliente, crie uma estrutura para extrair as seguintes informações:

- produto: breve descrição do produto
- entrega: cliente ficou satisfeito com a entrega
- produto: cliente ficou satisfeito com o produto
- atendimento: cliente ficou satisfeito com o atendimento
- satisfacao: satisfação geral do cliente com a compra

```
review_cliente = """Este soprador de folhas é bastante incrível. Ele tem
quatro configurações: sopro de vela, brisa suave, cidade ventosa
e tornado. Chegou em dois dias, bem a tempo para o presente de
aniversário da minha esposa. Acho que minha esposa gostou tanto
que ficou sem palavras. Até agora, fui o único a usá-lo, e tenho
usado em todas as manhãs alternadas para limpar as folhas do
nosso gramado. É um pouco mais caro do que os outros sopradores
de folhas disponíveis no mercado, mas acho que vale a pena pelas
características extras."""
```

06. Chains - Encadeamento com Langchain

Vamos entender agora **Chains** do LangChain, um dos conceitos mais fundamentais e poderosos para construir aplicações robustas e complexas de IA. As Chains são essenciais porque atuam como a coluna vertebral de uma aplicação, conectando diversas estruturas como prompts, chatmodels e output parser de maneira estratégica para resolver problemas maiores e mais complexos. Mostraremos como criar chains utilizando a metodologia mais recente desenvolvida pelo LangChain, chamada LCEL.

O que são Chains?

Chains, ou Cadeias, são sequências etapas que são processadas para realizar tarefas mais complexas que não poderiam ser eficientemente resolvidas por um único prompt. Isso é feito quebrando um problema grande em problemas menores e mais gerenciáveis, que são resolvidos sequencialmente ou em paralelo. ### Por que usar Chains?

Quando solicitamos a um modelo de linguagem que execute várias tarefas em um único Prompt, como traduzir um texto, resumi-lo e identificar sua linguagem simultaneamente, a performance geralmente é prejudicada. As Chains permitem que cada tarefa seja atribuída a um Prompt específico, otimizando a performance e a precisão do modelo.

O que é LCEL?

A LangChain Expression Language, ou LCEL, é uma forma declarativa e intuitiva de compor chains, de maneira fácil e escalável. A LCEL foi criada para facilitar a vida dos desenvolvedores, permitindo que protótipos sejam rapidamente colocados em produção, desde a cadeia mais simples “prompt + LLM” até as cadeias mais complexas (com centenas de etapas).

Criando sua Primeira Chain com LCEL

Vamos começar com um exemplo simples para entender como tudo funciona. Primeiro, precisamos configurar nosso ambiente e importar as bibliotecas necessárias:

```
from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv())

from langchain_openai import ChatOpenAI
model = ChatOpenAI(model='gpt-3.5-turbo-0125')
```

Vamos criar a cadeia mais simples possível, composta de um prompt combinado a um modelo de linguagem. Para isso, importamos o promptTemplate:

```
from langchain_core.prompts import ChatPromptTemplate
prompt = ChatPromptTemplate.from_template('Crie uma frase sobre o seguinte: {assunto}')
```

Agora, vamos compor nossa chain. A beleza da LCEL está na simplicidade com que podemos encadear diferentes componentes:

```
chain = prompt | model
```

Para invocar a chain com um exemplo prático, vamos usar:

```
response = chain.invoke({'assunto': 'gatinhos'})
print(response)
```

```
AIMessage(content='Os gatinhos são seres adoráveis que conseguem conquistar noss
```

Adicionando mais elementos à chain

Suponha que queremos apenas o texto da resposta, sem metadados adicionais. Podemos adicionar um `output_parser` à nossa chain:

```
from langchain_core.output_parsers import StrOutputParser
output_parser = StrOutputParser()

chain = prompt | model | output_parser
print(chain.invoke({'assunto': 'gatinhos'}))
```

```
'"Gatinhos são fofos, brincalhões e cheios de amor para dar."'
```

E veja como é simples ir compondo nossa chain e adicionando novos elementos. Pense que após esta chain, poderíamos adicionar um novo prompt que pega a resposta produzida após o output parser e processa ele novamente, para gerar uma nova resposta e assim por diante.

Atenção: A ordem importa!

É crucial que os componentes da chain sejam organizados na ordem correta para que os dados fluam corretamente de um estágio para o outro. Por exemplo, inverter a ordem resultará em erro, pois cada componente espera um tipo específico de entrada.

```
chain = model | prompt | output_parser #ORDEM INCORRETA!
chain.invoke({'assunto': 'gatinhos'}) #Invoke retornará um erro
```

Entendendo a ordem correta

Para garantir que você não cometa erros, é importante entender o tipo de entrada e saída de cada componente:

- **Prompt**: Recebe um dicionário e retorna um `PromptValue`.
- **Model (ChatModel)**: Recebe uma string, uma lista de mensagens de chat ou um `PromptValue` e retorna uma `ChatMessage`.
- **OutputParser**: Recebe a saída de um LLM ou ChatModel e retorna um tipo de dado específico, dependendo do parser.

Este é um esquema geral dos principais componentes do LangChain e os tipos de entada e saída que eles suportam:

Component	Tipo de Entrada	Tipo de Saída
Prompt	Dicionário	PromptValue
ChatModel	String única, lista de mensagens de chat ou PromptValue	Mensagem de Chat
LLM	String única, lista de mensagens de chat ou PromptValue	String
OutputParser	A saída de um LLM ou ChatModel	Depende do parser
Retriever	String única	Lista de Documentos
Tool	String única ou dicionário, dependendo da ferramenta	Depende da ferramenta

Você pode também verificar os tipos de entrada e saída utilizando os métodos input_schema e output_schema:

```
print(prompt.input_schema.schema_json(indent=4))
```

```
{
    "title": "PromptInput",
    "type": "object",
    "properties": {
        "assunto": {
            "title": "Assunto",
            "type": "string"
```

```
        }
    }
}
```

Chains Clássicas vs. LCEL

Antes da LCEL, criar chains era mais complexo e menos intuitivo. Veja como seria com a abordagem clássica:

```
from langchain.chains.llm import LLMChain
from langchain_openai.chat_models import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate

model = ChatOpenAI(model="gpt-3.5-turbo")
prompt = ChatPromptTemplate.from_template("Crie uma frase sobre o assunto: {assunto}")
output_parser = StrOutputParser()

chain = LLMChain(llm=model, prompt=prompt, output_parser=output_parser)
print(chain.invoke({'assunto': 'gatinhos'}))
```

Como você pode ver, a LCEL simplifica significativamente o processo de criação de chains, tornando seu código mais limpo e fácil de entender.

Praticando com Chains

A melhor maneira de entender e dominar o uso de Chains é praticando. Tente criar suas próprias Chains para diferentes tipos de aplicações. Experimente combinar diferentes tipos de Chains e veja como você pode resolver problemas complexos de maneira eficiente.

07. Criando Chains mais complexas e observando com LangSmith

Agora que sabemos criar chains simples com LCEL, podemos começar a nos aventurar em chains mais complexas e compreender o verdadeiro poder do LangChain. A capacidade de compor chains nos permite criar "chains de chains". Isso mesmo, podemos acoplar pequenos blocos de chains, moldando-os para criar aplicações muito mais complexas e, consequentemente, mais poderosas.

A capacidade de um modelo de linguagem aumenta drasticamente conforme mais específica for nossa chamada, e é isso que buscamos fazer ao quebrar uma aplicação em múltiplas cadeias. Vamos explorar como somar diferentes chains para obter resultados mais elaborados.

Somando chains

Para ilustrar como podemos combinar diferentes chains, vamos criar duas chains distintas: uma que gera uma curiosidade sobre um assunto e outra que cria uma história baseada nessa curiosidade.

Primeiro, definimos a chain que gera uma curiosidade:

```
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_template('Fale uma curiosidade sobre o assunto: {assunto}')
chain_curiosidade = prompt | ChatOpenAI() | StrOutputParser()
```

Em seguida, criamos a chain que gera uma história a partir de um fato curioso:

```
prompt = ChatPromptTemplate.from_template('Crie uma história sobre o seguinte fato curioso:
↪  {assunto}')
chain_historia = prompt | ChatOpenAI() | StrOutputParser()
```

Agora, podemos combinar essas duas chains em uma única chain:

```
chain = chain_curiosidade | chain_historia
```

Ao invocar essa chain com um assunto específico, como "peixes", obtemos um resultado que combina a curiosidade e a história gerada:

```
result = chain.invoke({'assunto': 'peixes'})
print(result)
```

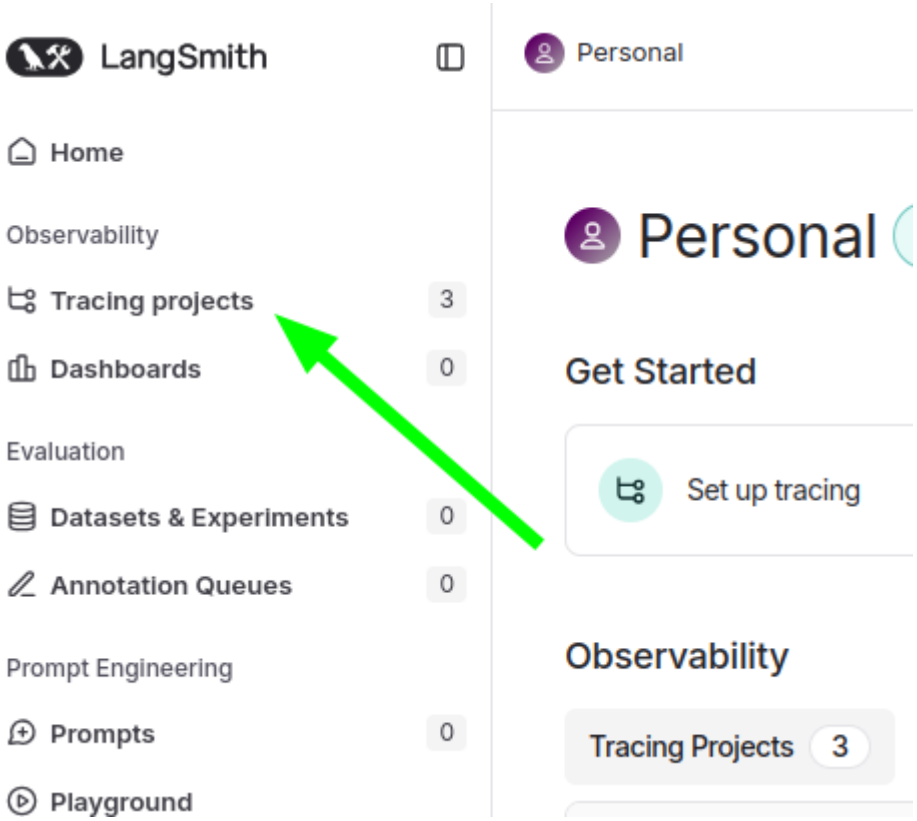
Essa abordagem modular nos permite construir aplicações mais complexas e dinâmicas, aproveitando o poder dos modelos de linguagem de forma eficaz.

Como entender o que está acontecendo?

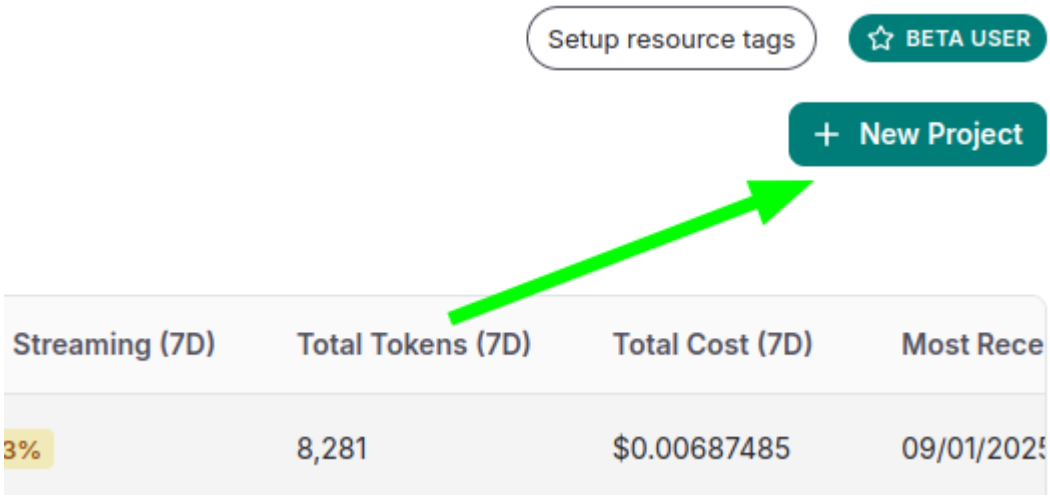
Para entender melhor o que está acontecendo em nossas chains e como elas interagem, podemos utilizar o **LangSmith**. O LangSmith é uma ferramenta projetada para ajudar desenvolvedores a monitorar, depurar e otimizar suas aplicações construídas com LangChain. Ele fornece uma interface visual que permite visualizar o fluxo de dados entre diferentes componentes, facilitando a identificação de problemas e a compreensão do comportamento das chains.

Com o LangSmith, você pode acompanhar as entradas e saídas de cada parte da sua aplicação, permitindo que você veja como as informações estão sendo processadas e transformadas ao longo do caminho. Isso é especialmente útil quando se trabalha com chains mais complexas, onde a interação entre diferentes componentes pode se tornar difícil de rastrear.

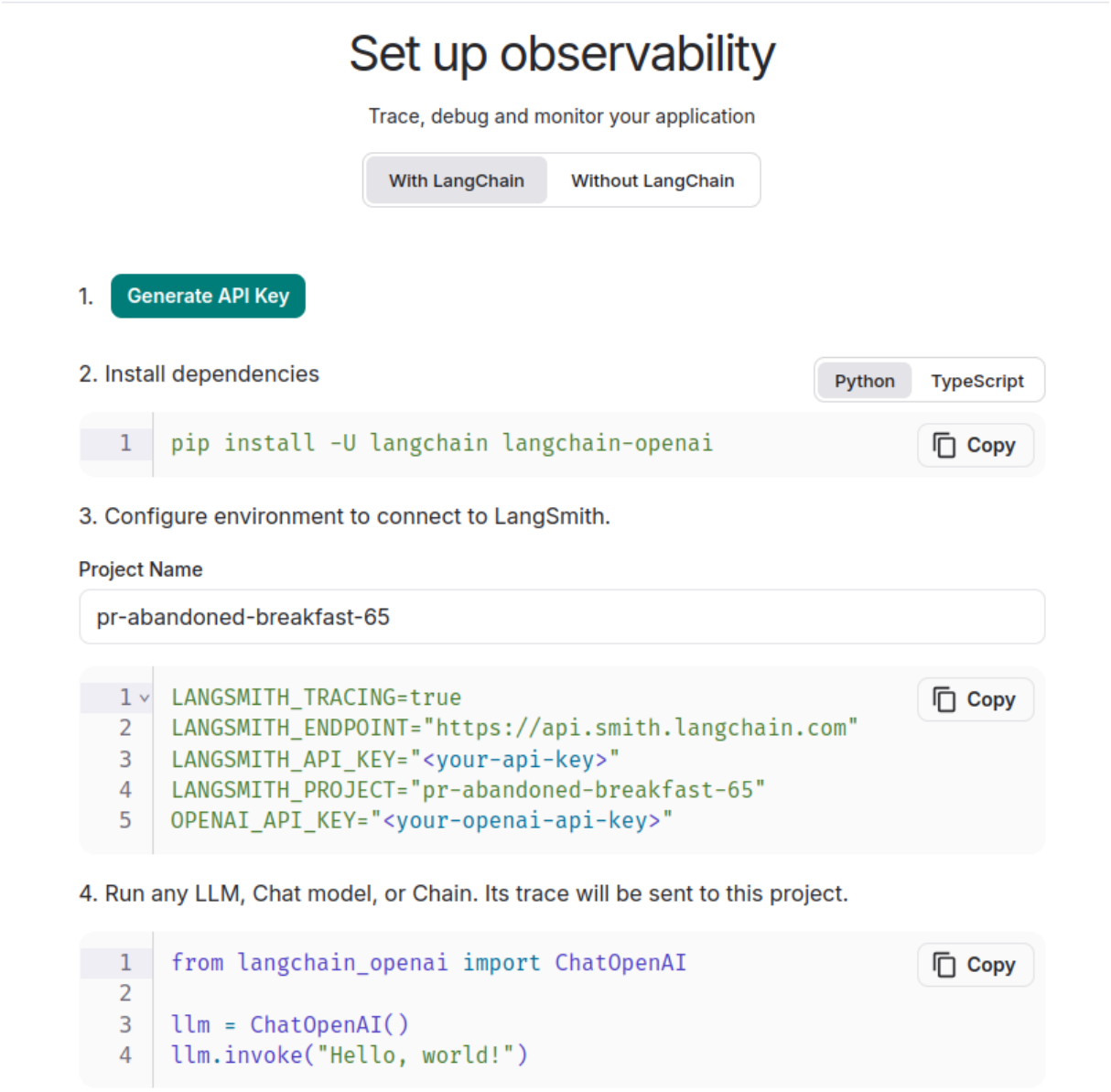
Você pode fazer o login na plataforma através deste link! Depois é só clicar no item “Tracing projects”:



Depois é só clicar em “New Project”:



E seguir os passos informados na plataforma:



Eles consistem em criar uma chave, clicando no botão 'Generate API Key'. Esta chave deve ser colocada em um arquivo .env juntamente ao projeto que você está desenvolvendo. Além dela, outras informações devem ser passadas no arquivo:

```
LANGSMITH_TRACING=true
LANGSMITH_ENDPOINT="https://api.smith.langchain.com"
LANGSMITH_API_KEY="<your-api-key>"
```

```
LANGSMITH_PROJECT="pr-abandoned-breakfast-65"
OPENAI_API_KEY="<your-openai-api-key>"
```

E pronto, é só reiniciar sua aplicação e seu projeto passará a ser observado pelo LangSmith, permitindo uma depuração muito mais rápida e fácil!

08. Runnables - As estruturas básicas de uma Chain

Agora que você já conhece as chains, está na hora de entendê-las com mais profundidade. Para isso, vamos introduzir o conceito de Runnables!

O que são Runnables?

Os **Runnables** são componentes modulares que formam uma Chain dentro do LangChain. Eles são, essencialmente, estruturas que podem ser "rodadas" ou executadas. Quando criamos uma Chain usando LCEL (LangChain Expression Language), estamos montando uma sequência de Runnables que trabalham juntos para processar dados ou realizar tarefas. Essa modularidade permite que você crie aplicações mais dinâmicas e eficientes, aproveitando o poder dos modelos de linguagem de forma estruturada.

Métodos dos Runnables de LCEL

A interface padrão de Runnables em LCEL inclui os seguintes métodos:

- **stream:** transmite de volta fragmentos da resposta.
- **invoke:** chama a cadeia com um input.
- **batch:** chama a cadeia com uma lista de inputs.

Além disso, esses métodos possuem versões assíncronas correspondentes que devem ser usadas com a sintaxe `asyncio await` para permitir concorrência:

- **astream:** transmite de volta fragmentos da resposta de forma assíncrona.
- **ainvoke:** chama a cadeia com um input de forma assíncrona.
- **abatch:** chama a cadeia com uma lista de inputs de forma assíncrona.

Exemplo de criação de uma Chain

Para ilustrar como os Runnables funcionam, vamos criar uma Chain simples que utiliza um modelo de linguagem para gerar frases sobre um assunto específico:

```
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate

model = ChatOpenAI()
prompt = ChatPromptTemplate.from_template("Crie uma frase sobre o assunto: {assunto}")

chain = prompt | model
```

Invoke

O método `invoke` é o método básico para inserir um input na Chain e receber uma resposta. Por exemplo, ao invocar a Chain com o assunto "cachorrinhos", podemos obter uma resposta como esta:

```
response = chain.invoke({'assunto': 'cachorrinhos'})
print(response)
```

A saída será uma mensagem gerada pelo modelo, como:

```
AIMessage(content='Cachorrinhos são seres adoráveis que trazem alegria e amor in
```

Embora o `invoke` possa ser chamado com uma simples string quando existe apenas um input no prompt, a forma mais recomendada é informar especificamente o nome da input através de um dicionário.

Stream

Para receber uma saída conforme ela é gerada pelo modelo, utilizamos o método `stream`. Isso permite que a resposta seja transmitida em tempo real:

```
for stream in chain.stream({'assunto': 'cachorrinhos'}):
    print(stream.content, end='')
```

A saída será exibida à medida que é gerada, proporcionando uma experiência mais interativa.

Batch

O método `batch` é utilizado para fazer múltiplas requisições em paralelo. Isso é especialmente útil quando você precisa lidar com várias entradas ao mesmo tempo:

```
responses = chain.batch([{'assunto': 'cachorrinhos'}, {'assunto': 'gatinhos'}, {'assunto':
↪  'cavalinhos'}])
print(responses)
```

A saída será uma lista de mensagens geradas para cada um dos assuntos fornecidos.

Runnables Assíncronos

Os Runnables também oferecem métodos assíncronos, que permitem que as operações sejam realizadas de forma não-bloqueante. Isso significa que seu programa pode continuar executando outras tarefas enquanto espera que a Chain complete suas operações.

Ainvoke

O `ainvoke` é a versão assíncrona do `invoke`. Ele é usado para fazer chamadas assíncronas à Chain:

```
import asyncio

async def processa_chain(input):
    resposta = await chain.ainvoke(input)
    return resposta

# Criando e esperando tarefas assíncronas
task1 = asyncio.create_task(processa_chain({'assunto': 'gatinho'}))
await task1
```

Runnables Especiais

Existem diversos tipos de Runnables criados pelo LangChain que auxiliam a criar fluxos mais complexos. Entre else, três se destacam por sua funcionalidade e flexibilidade: RunnablePassthrough, RunnableLambda e RunnableParallel. Vamos explorar cada um deles.

RunnablePassthrough

O RunnablePassthrough é um tipo de Runnable que simplesmente passa a entrada recebida como saída, podendo também adicionar chaves adicionais ao resultado. Ele se comporta quase como uma função de identidade, mas é útil em cenários onde você deseja incluir lógica adicional ou manipular a entrada de alguma forma antes de passar adiante.

```
from langchain_core.runnables import RunnablePassthrough

# Criando um RunnablePassthrough
runnable = RunnablePassthrough()

# Invocando o runnable
resultado = runnable.invoke({'mensagem': 'Olá, mundo!'})
print(resultado)  # Saída: {'mensagem': 'Olá, mundo!'}
```

Neste exemplo, o RunnablePassthrough simplesmente retorna o dicionário recebido, permitindo que ele seja usado em um fluxo de trabalho onde a entrada precisa ser preservada.

RunnableLambda

O RunnableLambda permite que você crie um Runnable a partir de uma função Python arbitrária. Isso é útil para encapsular qualquer lógica que você queira aplicar aos dados ou para transformar entradas em saídas de uma maneira específica. Um aspecto interessante do RunnableLambda é que ele pode

ser construído para funcionar de forma síncrona ou assíncrona, dependendo de como a função é definida.

```
from langchain_core.runnables import RunnableLambda

# Definindo uma função simples
def cumprimentar(nome):
    return f'Olá, {nome}!'

# Criando um RunnableLambda a partir da função
runnable_cumprimentar = RunnableLambda(cumprimentar)

# Invocando o RunnableLambda
resultado = runnable_cumprimentar.invoke('Maria')
print(resultado)  # Saída: 'Olá, Maria!'
```

Neste exemplo, o RunnableLambda encapsula a função cumprimentar, permitindo que ela seja chamada como um Runnable, facilitando sua integração em cadeias.

RunnableParallel

O RunnableParallel é um componente poderoso que permite executar múltiplos Runnables simultaneamente. Ele aceita um dicionário de Runnables e fornece a mesma entrada a todos eles, retornando um dicionário com os resultados correspondentes. Isso é especialmente útil quando você precisa realizar várias operações independentes ao mesmo tempo, economizando tempo em processos que podem ser feitos em paralelo.

```
from langchain_core.runnables import RunnableLambda, RunnableParallel

# Definindo algumas funções simples
def adicionar_um(x):
    return x + 1

def multiplicar_por_dois(x):
    return x * 2

# Criando Runnables a partir das funções
runnable_adicionar = RunnableLambda(adicionar_um)
runnable_multiplicar = RunnableLambda(multiplicar_por_dois)

# Criando um RunnableParallel
runnable_paralelo = RunnableParallel(
    adicionar=runnable_adicionar,
    multiplicar=runnable_multiplicar
)

# Invocando o RunnableParallel
resultado = runnable_paralelo.invoke({'x': 3})
print(resultado)  # Saída: {'adicionar': 4, 'multiplicar': 6}
```

Neste exemplo, RunnableParallel executa as duas operações (adicionar_um e multiplicar_por_dois) ao mesmo tempo, retornando os resultados em um dicionário. O uso de RunnableLambda para criar as funções que serão executadas em paralelo mostra como esses componentes trabalham bem juntos.

Dominar os métodos dos Runnables de LCEL é crucial para desenvolver aplicações eficientes e responsivas usando Python e LangChain. Com a capacidade de invocar, transmitir e processar múltiplas entradas em paralelo, você pode criar aplicações mais dinâmicas e interativas.

09 - Roteamento - Trabalhando com seleção de Chains

O roteamento, ou as Chains de roteamento, são uma ferramenta poderosa dentro se uma aplicação de IA que permite direcionar interações de forma inteligente e eficaz. Imagine que você está desenvolvendo uma aplicação que precisa lidar com diferentes tipos de perguntas, cada uma exigindo uma abordagem específica. É aqui que as Chains de roteamento entra em cena! Elas ajudam a decidir para qual Chain especializada uma determinada entrada do usuário deve ser enviada, permitindo que a aplicação responda de maneira mais adequada e eficiente às necessidades do usuário.

Por exemplo, digamos que estamos criando uma aplicação para responder dúvidas de alunos em uma escola de ensino médio. Se um usuário faz uma pergunta sobre física, a Chain de roteamento pode direcionar essa pergunta para uma Chain especializada em responder questões de física. Se a pergunta for sobre história, ela será enviada para uma Chain focada em história. Isso tudo é feito automaticamente pela Chain de roteamento, com base na análise da entrada do usuário.

Criando Estruturas de Roteamento

Para ilustrar como criar uma estrutura de roteamento, vamos considerar uma aplicação que possui três Chains especializadas: matemática, física e história. Primeiro, vamos definir as chains para cada uma dessas áreas do conhecimento:

```
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate

model = ChatOpenAI(model='gpt-4o-mini')

prompt = ChatPromptTemplate.from_template('''Você é um professor de matemática de ensino
↪  fundamental
capaz de dar respostas muito detalhadas e didáticas. Responda a seguinte pergunta de um aluno:
Pergunta: {pergunta}''')
chain_matematica = prompt | model

prompt = ChatPromptTemplate.from_template('''Você é um professor de física de ensino
↪  fundamental
capaz de dar respostas muito detalhadas e didáticas. Responda a seguinte pergunta de um aluno:
Pergunta: {pergunta}''')
chain_fisica = prompt | model

prompt = ChatPromptTemplate.from_template('''Você é um professor de história de ensino
↪  fundamental
capaz de dar respostas muito detalhadas e didáticas. Responda a seguinte pergunta de um aluno:
Pergunta: {pergunta}''')
chain_historia = prompt | model

prompt = ChatPromptTemplate.from_template('''{pergunta}''')
chain_generica = prompt | model
```

Categorizador de Perguntas

Para categorizar as perguntas dos alunos, podemos criar um modelo estruturado que identifica a área de conhecimento da pergunta:

```
from pydantic import BaseModel, Field

prompt = ChatPromptTemplate.from_template('Você deve categorizar a seguinte pergunta:
↪  {pergunta}')

class Categorizador(BaseModel):
    """Categoriza as perguntas de alunos do ensino fundamental"""
    area_conhecimento: str = Field(description='A área de conhecimento da pergunta feita pelo
↪  aluno. \
    Deve ser "física", "matemática" ou "história". Caso não se encaixe em nenhuma delas,
↪  retorne "outra"')

model_estruturado = prompt | model.with_structured_output(Categorizador)
```

Ao invocar o modelo com uma pergunta, ele retornará a categoria correspondente:

```
model_estruturado.invoke({'pergunta': 'Quando foi a independência dos Estados Unidos?'})
```

A saída será:

```
Categorizador(area_conhecimento='história')
```

Criando a Estrutura de Roteamento

Agora, vamos criar a estrutura de roteamento que usará o categorizador para decidir qual chain deve ser invocada:

```
from langchain_core.runnables import RunnablePassthrough

chain = RunnablePassthrough().assign(categoria=model_estruturado)
```

Ao invocar essa chain com uma pergunta, obteremos a categoria da pergunta:

```
chain.invoke({'pergunta': 'Quando foi a independência dos Estados Unidos?'})
```

A saída será um dicionário que inclui a pergunta e sua categoria:

```
{'pergunta': 'Quando foi a independência dos Estados Unidos?',
 'categoria': Categorizador(area_conhecimento='história')}
```

Função de Roteamento

Agora, precisamos definir uma função que irá direcionar a pergunta para a chain apropriada com base na categoria identificada:

```
def route(input):
    if input['categoria'].area_conhecimento == 'matemática':
        return chain_matematica
    if input['categoria'].area_conhecimento == 'física':
        return chain_fisica
    if input['categoria'].area_conhecimento == 'história':
        return chain_historia
    return chain_generica
```

Invocando a Chain de Roteamento

Por fim, podemos combinar a estrutura de roteamento com a chain de categorização:

```
chain = RunnablePassthrough().assign(categoria=model_estruturado) | route
chain.invoke({'pergunta': 'Quando foi a independência dos Estados Unidos?'})
```

A saída será uma resposta detalhada sobre a independência dos Estados Unidos, gerada pela chain de história.

```
AIMessage(content='Claro! A Independência dos Estados Unidos foi declarada no dia 4 de julho
↪  de 1776. Vamos entender melhor o contexto e a importância desse evento!\n\n### Contexto
↪  Histórico\n\n... (continua a resposta)')
```

Exemplo de Pergunta de Matemática

Se invocarmos a chain com uma pergunta de matemática, como:

```
chain.invoke({'pergunta': 'Quanto é 1 + 21?'})
```

A resposta será uma explicação detalhada sobre a soma, gerada pela chain de matemática.

Mas a pergunta está sendo direcionada para a Chain correta?

Esta dúvida deve surgir ao criar uma Chain de roteamento, e é natural. Um bom desenvolvedor de aplicações necessariamente vai verificar se a sua aplicação está executando da maneira desejada, com um grau de confiança aceitável. E para fazer a verificação, podemos utilizar o LangSmith e fazer a observação de quais Chains estão sendo usadas a cada resposta dada pela minnha aplicação!

As cadeias de roteamento são uma ferramenta poderosa para direcionar interações de forma inteligente, permitindo que sua aplicação responda de maneira mais adequada às necessidades dos

usuários. Ao categorizar perguntas e direcioná-las para as chains apropriadas, você pode criar uma experiência de aprendizado mais rica e interativa.

10. Memory - Adicionando memória à conversa com o modelo

O que é a memória em LangChain?

A memória é um conceito fundamental na criação de aplicações voltadas a conversas. No contexto dos modelos de linguagem, como os que usamos no LangChain, a memória não é apenas um recurso técnico; é o coração de uma conversa inteligente e contextualizada. Imagine ter uma conversa onde cada interação é esquecida imediatamente. Frustrante, não é? Para evitar isso, LangChain implementa várias formas de memória, permitindo que o modelo lembre de interações passadas. Isso enriquece a conversa, tornando-a mais relevante e personalizada.

Métodos das classes de memória

LangChain oferece diversas utilidades para adicionar memória a um sistema. Essas utilidades podem ser usadas de forma independente ou integradas em uma chain. Um exemplo simples de implementação de memória é o uso da classe `InMemoryChatMessageHistory`, que armazena as mensagens trocadas durante a conversa.

Exemplo de Uso da Memória

Vamos começar criando uma instância de `InMemoryChatMessageHistory`:

```
from langchain_core.chat_history import InMemoryChatMessageHistory

memory = InMemoryChatMessageHistory()
```

Agora, podemos adicionar mensagens de usuário e do modelo à memória:

```
memory.add_user_message('Olá, modelo!')
memory.add_ai_message('Olá, user')
```

Para visualizar as mensagens armazenadas, podemos acessar o atributo `messages`:

```
memory.messages
```

A saída será uma lista de mensagens, incluindo as interações que acabamos de adicionar:

```
[HumanMessage(content='Olá, modelo!', additional_kwargs={}, response_metadata={}
 AIMessage(content='Olá, user', additional_kwargs={}, response_metadata={})]
```

Criando uma Conversa com Memória

Agora, vamos criar uma chain que utiliza memória para manter o contexto da conversa. Primeiro, definimos um prompt que inclui um espaço reservado para a memória:

```
from langchain_openai.chat_models import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_messages([
    ("system", "Você é um tutor de programação chamado Asimo. Responda as perguntas de forma
↪  didática."),
    ("placeholder", "{memoria}"),
    ("human", "{pergunta}"),
])
chain = prompt | ChatOpenAI()
```

Integrando Memória à Chain

Para integrar a memória à chain, utilizamos a classe `RunnableWithMessageHistory`, que permite associar a chain a um histórico de mensagens:

```
from langchain_core.runnables.history import RunnableWithMessageHistory

store = {}
def get_by_session_id(session_id):
    if session_id not in store:
        store[session_id] = InMemoryChatMessageHistory()
    return store[session_id]

chain_com_memoria = RunnableWithMessageHistory(
    chain,
    get_by_session_id,
    input_messages_key='pergunta',
    history_messages_key='memoria'
)
```

Interagindo com a Chain

Agora podemos invocar a chain com uma pergunta e um ID de sessão:

```
config = {'configurable': {'session_id': 'usuaria_a'}
resposta = chain_com_memoria.invoke({'pergunta': 'O meu nome é Adriano'}, config=config)
print(resposta)
```

A saída será uma resposta personalizada, reconhecendo o nome do usuário:

```
AIMessage(content='Olá, Adriano! Como posso te ajudar hoje?', ...)
```

Se fizermos uma nova pergunta, como:

```
resposta = chain_com_memoria.invoke({'pergunta': 'Qual é o meu nome?'}, config=config)
print(resposta)
```

A resposta será:

```
AIMessage(content='Seu nome é Adriano. Como posso te ajudar, Adriano?', ...)
```

Memória em Diferentes Sessões

Se mudarmos o ID da sessão, a memória não será compartilhada:

```
config = {'configurable': {'session_id': 'usuaria_b'}
resposta = chain_com_memoria.invoke({'pergunta': 'Qual é o meu nome?'}, config=config)
print(resposta)
```

A saída será:

```
AIMessage(content='Desculpe, mas eu não tenho acesso a informações pessoais sobr
```

Isso demonstra como a memória pode ser gerenciada de forma a manter o contexto em sessões diferentes.

A implementação de memória em sistemas de conversação é essencial para criar interações mais ricas e personalizadas. Com as ferramentas que o LangChain oferece, você pode facilmente adicionar memória às suas aplicações, permitindo que o modelo lembre de interações passadas e forneça respostas mais contextuais.

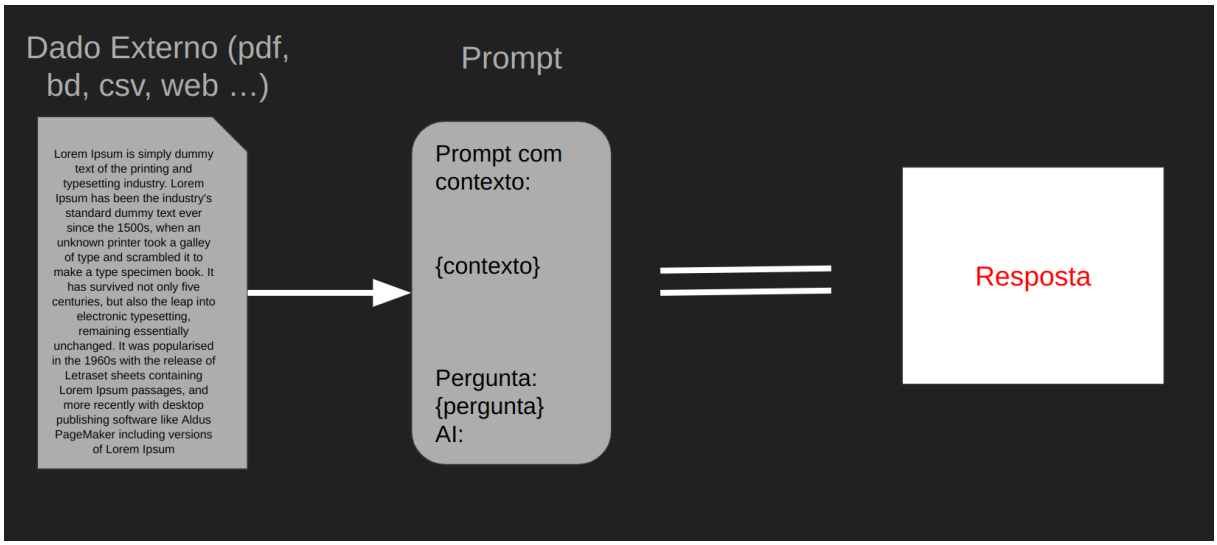
11. RAG - Introdução à técnica Retrieval Augmented Generation

Hoje, vamos nos aprofundar em uma das técnicas mais fascinantes e úteis no mundo da Inteligência Artificial aplicada a linguagem: o **Retrieval Augmented Generation (RAG)**. Esta técnica é um verdadeiro game-changer para criar aplicações personalizadas que interagem de maneira inteligente e específica com os dados relevantes. Vamos entender como isso funciona?

O que é RAG?

RAG, ou **Geração Aumentada por Recuperação de Dados**, é uma técnica que permite enriquecer as respostas de um modelo de linguagem (como o GPT) com informações específicas retiradas de um conjunto de dados externos. Isso é especialmente útil quando queremos que o modelo responda perguntas ou faça declarações sobre informações que não estão contidas em seu treinamento original.

Imagine que você quer construir um chatbot para responder perguntas específicas sobre sua empresa. Um modelo de linguagem padrão não teria acesso às informações específicas da sua empresa, a menos que você as forneça de alguma forma. É aqui que o RAG entra em jogo, permitindo que o modelo acesse e utilize essas informações específicas para gerar respostas relevantes e precisas.

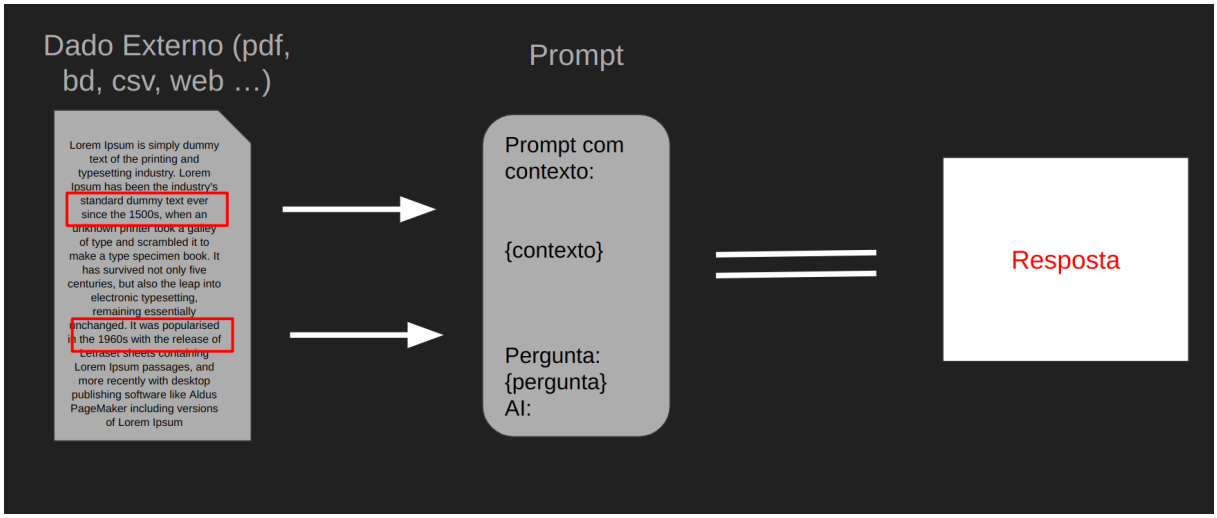


Desafios ao Utilizar RAG

Embora o RAG ofereça uma maneira poderosa de enriquecer as respostas de um modelo de linguagem com informações específicas, existem desafios significativos que precisam ser superados para utilizar essa técnica de forma eficaz.

Um dos principais desafios é a limitação inerente ao tamanho da entrada que os modelos de linguagem podem processar. Modelos como o GPT têm um limite máximo de tokens (palavras ou pedaços de palavras) que podem ser considerados em uma única passagem. Isso significa que não podemos simplesmente alimentar o modelo com uma quantidade ilimitada de informações e esperar que ele processe tudo de uma vez. Fazer isso resultaria em custos proibitivos e, muitas vezes, seria tecnicamente inviável devido às restrições de tamanho de entrada do modelo.

Para contornar essa limitação, é necessário fragmentar o texto original em diversos trechos menores, conhecidos como "chunks". Essa fragmentação deve ser feita de maneira estratégica para garantir que cada chunk contenha informações suficientemente relevantes e contextuais para a consulta do usuário. No entanto, isso introduz outro desafio: a seleção de chunks relevantes.



Durante o processo de retrieval, o modelo deve ser capaz de identificar quais chunks contêm as informações mais pertinentes à pergunta ou ao tópico em questão. Chunks que contêm informações irrelevantes ou enganosas podem confundir o modelo e levar a respostas imprecisas ou fora de contexto. Portanto, a seleção de chunks deve ser feita com extremo cuidado, utilizando algoritmos de busca e comparação eficientes que possam avaliar a relevância do conteúdo de cada chunk em relação à consulta do usuário.

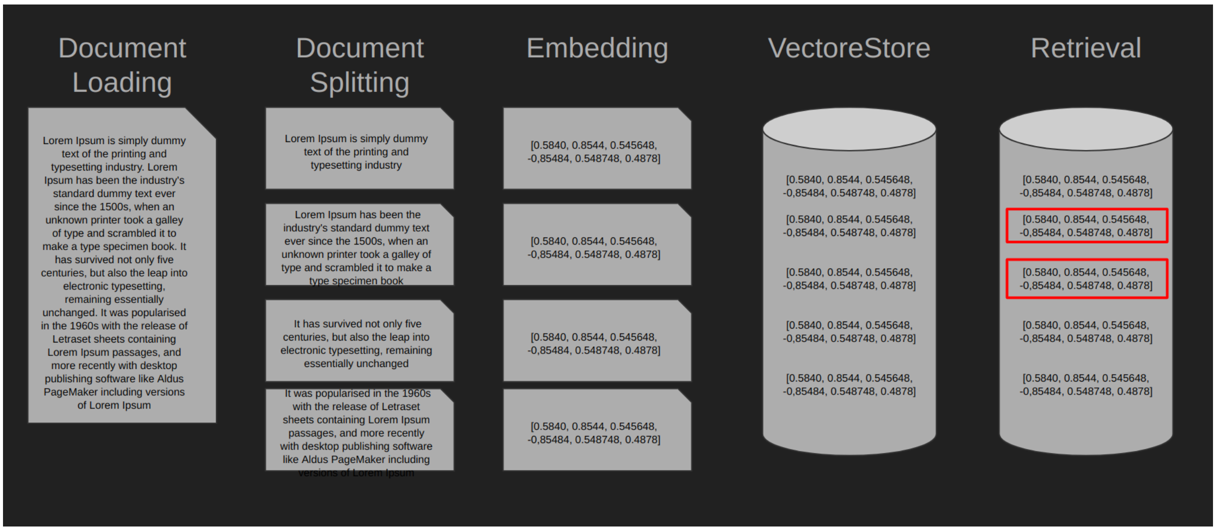
Além disso, a qualidade do embedding dos chunks é crucial, pois vetores numéricos mal construídos podem não capturar a semântica correta do texto, prejudicando a capacidade do modelo de encontrar as informações corretas. Portanto, é essencial utilizar técnicas de embedding avançadas que possam representar de forma precisa o significado e o contexto dos textos.

Em resumo, o sucesso da aplicação do RAG depende de uma série de fatores, incluindo a habilidade de fragmentar o texto de forma eficiente, a precisão dos algoritmos de seleção de chunks e a qualidade dos embeddings gerados. Ao superar esses desafios, podemos desbloquear o potencial completo do RAG para criar aplicações de IA que fornecem respostas ricas e contextualizadas, elevando a interação

entre humanos e máquinas a um novo patamar.

Quebrando RAG em etapas

O processo de RAG pode ser dividido em várias etapas chave: **Document Loading, Text Splitting, Embedding, VectorStores e Retrieval**.



Vamos explorar brevemente cada uma delas:

1. Document Loading

Antes de mais nada, precisamos carregar os documentos que contêm as informações que queremos que nosso modelo de IA utilize. Esses documentos podem estar em diversos formatos e locais, como PDFs, CSVs, bases de dados ou até mesmo páginas web.

2. Text Splitting

Após carregar os documentos, o próximo passo é dividir o texto em pedaços menores. Isso é necessário porque os modelos de linguagem têm limitações quanto ao tamanho do texto que podem processar de uma vez.

3. Embedding

O processo de embedding transforma os pedaços de texto em vetores numéricos. Esses vetores representam semanticamente o conteúdo do texto, permitindo que o modelo de IA realize comparações

e buscas de maneira eficiente.

4. VectorStores

Após criar os vetores, armazenamos eles em uma estrutura chamada VectorStore. Isso facilita a recuperação rápida de informações relevantes durante o processo de consulta.

5. Retrieval

Finalmente, o processo de retrieval utiliza os vetores para encontrar os pedaços de texto mais relevantes para uma dada consulta. Isso é feito comparando o vetor da consulta com os vetores armazenados na VectorStore.

Com o RAG, podemos superar as limitações dos modelos de linguagem padrão, fornecendo-lhes acesso a informações específicas e relevantes. Isso permite a criação de aplicações verdadeiramente personalizadas e poderosas.

RAG ou Fine Tuning? Qual devo utilizar?

Muitos alunos inseridos nas técnicas mais recentes de aplicação de IA percebem uma semelhança entre os objetivos do RAG (Retrieval-Augmented Generation) e os de outra técnica chamada Fine Tuning. É comum surgirem dúvidas sobre quando utilizar uma e quando utilizar a outra, e por isso criamos este pequeno conteúdo para tentar esclarecer essa dúvida comum.

O que é Fine Tuning?

Fine Tuning é uma técnica que envolve o treinamento de um modelo de linguagem em um conjunto de dados menor e especializado. O objetivo é ajustar os parâmetros do modelo para que ele se torne mais eficaz em tarefas específicas, alinhando-o com as nuances e terminologias de um domínio particular. Essa abordagem é útil quando se deseja que o modelo execute tarefas que exigem um entendimento profundo de um contexto específico.

Comparação entre RAG e Fine Tuning

Objetivo e Abordagem

- **RAG**: O foco do RAG é integrar dados externos a um modelo de linguagem, permitindo que ele acesse informações atualizadas e relevantes durante a geração de respostas. Isso é feito sem alterar o modelo em si, mas sim utilizando um banco de dados dinâmico que complementa as capacidades do modelo.
- **Fine Tuning**: O Fine Tuning, por outro lado, envolve a modificação do próprio modelo, ajustando seus parâmetros com base em um conjunto de dados especializado. Isso permite que o modelo aprenda a responder de maneira mais eficaz a perguntas específicas dentro de um domínio.

Utilização

- **Use RAG quando:**
  - Você precisa de uma solução escalável e segura que integre informações atualizadas e confiáveis.
  - O seu foco é em aplicações que exigem respostas precisas baseadas em dados dinâmicos, como chatbots que precisam acessar informações em tempo real.
  - Você deseja evitar o custo e o tempo associados ao treinamento de um modelo, já que RAG pode ser mais econômico e rápido de implementar.
- **Use Fine Tuning quando:**
  - Você precisa que o modelo execute tarefas específicas que exigem um entendimento profundo de um contexto ou domínio.
  - O seu conjunto de dados é pequeno, mas altamente especializado, e você deseja que o modelo aprenda a responder de acordo com as nuances desse domínio.
  - Você está lidando com tarefas que exigem um estilo ou tom específico, como redação de documentos legais ou atendimento ao cliente.

Custo e Recursos

- **RAG:** Geralmente, RAG é mais econômico e menos intensivo em recursos, pois não requer o treinamento de um modelo do zero. Ele utiliza dados existentes e pode ser implementado rapidamente.
- **Fine Tuning:** O Fine Tuning pode ser mais caro e demorado, pois envolve o treinamento do modelo, o que requer recursos computacionais significativos e tempo para rotular e preparar os dados.

4. **Segurança e Privacidade:**

- **RAG:** Oferece maior segurança e privacidade, pois os dados permanecem em um ambiente seguro e controlado, sem se tornarem parte do modelo.

- **Fine Tuning:** Os dados utilizados para o treinamento se tornam parte do modelo, o que pode expor informações sensíveis se não forem gerenciados adequadamente.

Quando usar cada técnica

- **RAG** é ideal para aplicações que precisam de respostas dinâmicas e atualizadas, onde a segurança e a escalabilidade são prioridades. É uma escolha econômica e rápida para integrar dados externos.
- **Fine Tuning** é mais adequado para tarefas específicas que exigem um entendimento profundo de um domínio, onde o modelo precisa ser ajustado para responder de maneira eficaz a perguntas especializadas.

Ambas as técnicas têm seus próprios benefícios e podem ser utilizadas em conjunto, dependendo das necessidades do seu projeto. Avalie cuidadosamente suas prioridades e recursos disponíveis para escolher a abordagem mais adequada. E se você quer entender mais sobre a técnica de Fine Tuning, temos duas aulas no curso de Explorando a API da OpenAI onde esta é abordada.

12. Document Loaders - Carregando dados com Langchain

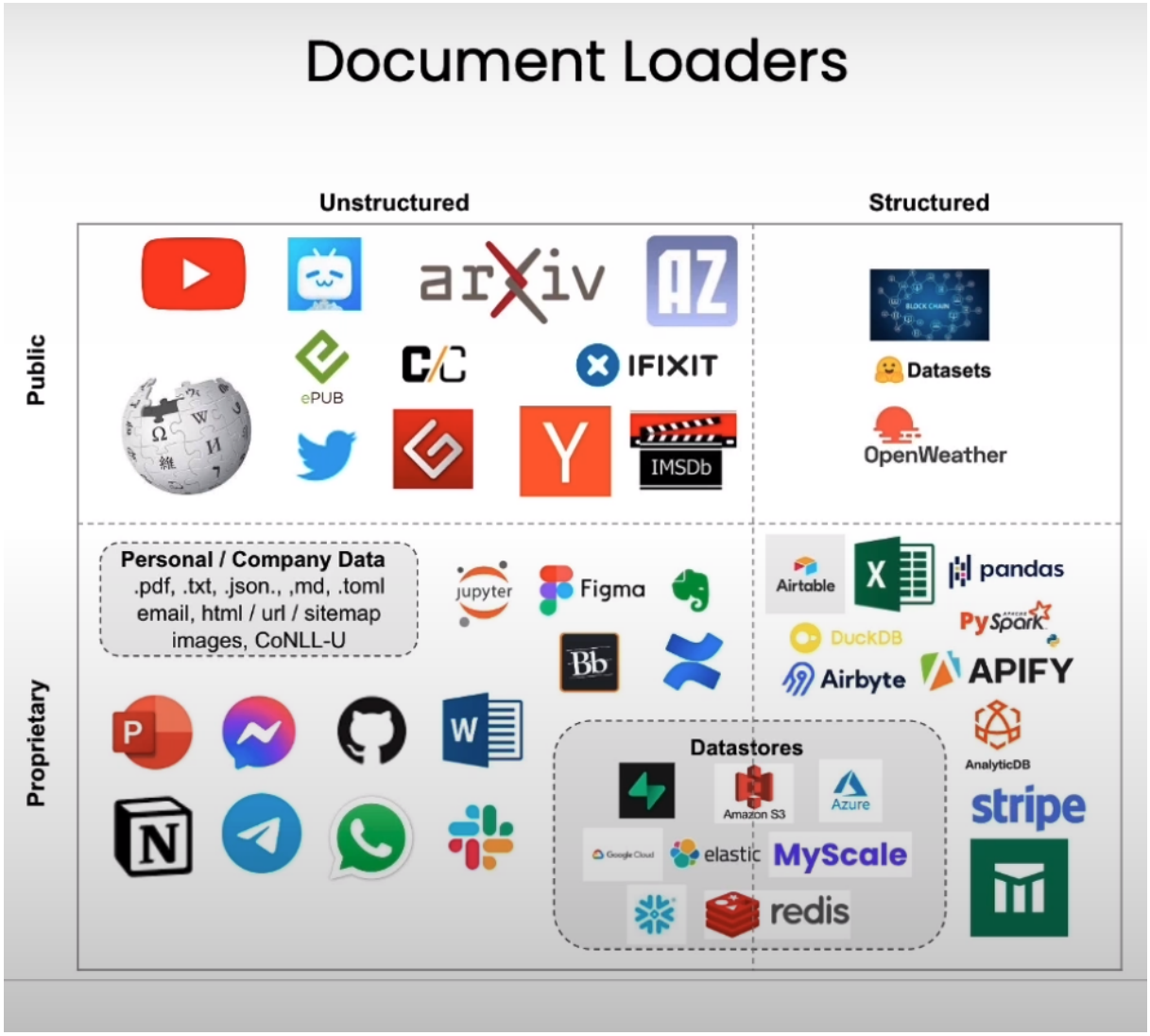
Vamos agora entender RAG com mais profundidade, começando pela sua primeira etapa: Document Loading.

O que são Document Loaders?

Document Loaders são ferramentas incríveis do Langchain que nos permitem carregar dados de diversas fontes, sejam elas arquivos locais ou conteúdos online. Eles são essenciais para alimentar nossas aplicações de IA com os dados necessários para que elas funcionem de maneira eficaz.

Tipos de Documentos

Os documentos podem ser **estruturados** (como tabelas em Excel ou bancos de dados SQL) ou **não estruturados** (como textos livres ou posts de redes sociais). Além disso, podem ser **públicos** (acessíveis via internet) ou **proprietários** (dados privados da sua empresa ou pessoais). Na imagem abaixo, você pode observar alguns dos diversos Document Loaders disponíveis no LangChain.



Carregando Diferentes Tipos de Documentos

PDFs

Vamos começar com algo que todos conhecemos: PDFs. O Langchain utiliza uma biblioteca chamada PyPDF para carregar PDFs. Veja como é simples:

```
from langchain_community.document_loaders.pdf import PyPDFLoader

caminho = 'caminho/para/seu/arquivo.pdf'
loader = PyPDFLoader(caminho)
documentos = loader.load()
```

```
print(f'Total de páginas carregadas: {len(documentos)}')
print(documentos[0].page_content)  # Mostra o conteúdo da primeira página
```

CSVs

E que tal carregar dados de um arquivo CSV? Com o Langchain, isso também é super fácil:

```
from langchain_community.document_loaders.csv_loader import CSVLoader

caminho = 'caminho/para/seu/arquivo.csv'
loader = CSVLoader(caminho)
documentos = loader.load()

print(f'Total de linhas carregadas: {len(documentos)}')
print(documentos[0].page_content)  # Mostra o conteúdo da primeira linha
```

Dados da Internet

YouTube Sim, você pode carregar dados diretamente do YouTube! Isso inclui baixar o áudio de vídeos e transcrevê-los usando o modelo Whisper da OpenAI:

```
from langchain_community.document_loaders.generic import GenericLoader
from langchain_community.document_loaders.blob_loaders.youtube_audio import YoutubeAudioLoader
from langchain.document_loaders.parsers import OpenAIWhisperParser

url = 'https://www.youtube.com/watch?v=seu_video'
save_dir = 'caminho/para/salvar'
loader = GenericLoader(
    YoutubeAudioLoader([url], save_dir),
    OpenAIWhisperParser()
)
docs = loader.load()

print(docs[0].page_content)  # Mostra a transcrição do vídeo
```

Para você conseguir utilizar este código no sitema Windows, é necessário realizar o download ffmpeg e adicioná-lo junto do script que você está rodando:

Web Scraping Carregar dados de páginas web também é possível. Imagine coletar informações diretamente de um artigo de blog:

```
from langchain_community.document_loaders.web_base import WebBaseLoader

url = 'https://seu.site.com/artigo'
loader = WebBaseLoader(url)
documentos = loader.load()

print(documentos[0].page_content)  # Mostra o conteúdo do artigo
```

Notion

E para os fãs do Notion, sim, vocês também podem carregar dados diretamente de páginas do Notion exportadas:

```
from langchain_community.document_loaders.notion import NotionDirectoryLoader

caminho = 'caminho/para/diretório/notion'
loader = NotionDirectoryLoader(caminho)
documentos = loader.load()

print(documentos[0].page_content)  # Mostra o conteúdo da primeira página do Notion
```

Documents de LangChain

Ao carregarmos dados utilizando um document loader, o LangChain armazena a informação em estruturas chamadas de *document*. Esta estrutura apresenta dois atributos principais: **page_content**, que armazena o conteúdo no formato de texto; **metadata**, que armazena informaçoes referente a origem e característica do documento:

```
from langchain_community.document_loaders.pdf import PyPDFLoader

caminho = 'caminho/para/seu/arquivo.pdf'
loader = PyPDFLoader(caminho)
documentos = loader.load()

print(f'Total de páginas carregadas: {len(documentos)}')

documento = documentos[0] # Acessando o primeiro documento, pois o LangChain retorna sempre
↪  uma lista de documentos
```

Para visualizar o conteúdo do documento:

```
print(documento.page_content)
```

Para visualizar os metadados:

```
print(documento.metadata)
```

Passos para Criar um Document Loader Customizado

O LangChain é flexível a ponto de permitir criar novos Document Loaders de forma personalizada, se adequando às suas necessidade de uso. Para isso, basta seguir os seguintes passos:

1. **Importar as Bibliotecas Necessárias:** Você precisará importar as classes e métodos do LangChain que serão utilizados na criação do seu loader.

```
from langchain.document_loaders.base import BaseLoader
from langchain.schema import Document
```

2. **Definir a Classe do Loader:** Crie uma nova classe que herda de BaseLoader. Essa classe deve implementar o método load(). Neste caso, criamos um loader que cria um documento para cada linha de texto de um arquivo.

```
class MyCustomLoader(BaseLoader):
    def __init__(self, source):
        self.source = source  # Fonte de dados, como um arquivo ou URL

    def load(self):
        # Lógica para carregar os dados da fonte
        documents = []
        # Exemplo: Carregar dados de um arquivo de texto
        with open(self.source, 'r') as file:
            content = file.read()
            # Criar um documento com o conteúdo e metadados
            documents.append(Document(page_content=content, metadata={"source": self.source}))
        return documents
```

3. **Utilizar o Document Loader Customizado:** Após definir a classe, você pode instanciá-la e utilizá-la para carregar os dados.

```
# Criar uma instância do loader
loader = MyCustomLoader('caminho/para/seu/arquivo.txt')
documentos = loader.load()

# Verificar o conteúdo carregado
for doc in documentos:
    print(doc.page_content)  # Exibir o conteúdo do documento
    print(doc.metadata)  # Exibir os metadados do documento
```

Exemplo Completo

Aqui está um exemplo completo de um Document Loader customizado que carrega dados de um arquivo de texto:

```
from langchain.document_loaders.base import BaseLoader
from langchain.schema import Document

class MyCustomLoader(BaseLoader):
    def __init__(self, source):
        self.source = source  # Fonte de dados, como um arquivo ou URL

    def load(self):
        documents = []
        with open(self.source, 'r') as file:
            content = file.read()
            documents.append(Document(page_content=content, metadata={"source": self.source}))
```

```
        return documents

# Utilizando o Document Loader customizado
loader = MyCustomLoader('caminho/para/seu/arquivo.txt')
documentos = loader.load()

# Exibindo o conteúdo e metadados
for doc in documentos:
    print(doc.page_content)
    print(doc.metadata)
```

Como vocês podem ver, os Document Loaders do Langchain são extremamente poderosos e versáteis, permitindo que você carregue quase qualquer tipo de dado que possa imaginar. Isso abre um mundo de possibilidades para suas aplicações de IA.

13. Text Splitters - Dividindo texto em trechos

Os Text Splitters são uma ferramenta essencial no processo de RAG, eles são os responsáveis por quebrar nossos textos em trechos menores, como veremos neste capítulo.

O que é Text Splitting?

Text Splitting, ou divisão de texto, é o processo de quebrar textos grandes em pedaços menores, chamados de "chunks". Isso é crucial porque os modelos de linguagem têm um limite para o tamanho de texto que podem processar de uma só vez. Se tentarmos alimentar um modelo com um documento muito grande, ele simplesmente não conseguirá processá-lo.

Por que é importante realizar uma boa quebra de dados?

Imagine que você tem um texto longo que precisa ser analisado por um modelo de IA. Se não dividirmos o texto adequadamente, podemos acabar com pedaços de texto que, isoladamente, não fazem sentido, comprometendo a qualidade da informação e, consequentemente, a eficácia da aplicação. Por exemplo:

Texto original: > "O carro novo da Fiat se chama Toro, tem 120 cavalos de potência e o preço sugerido é 135 mil reais."

Divisão inadequada: - "O carro novo da Fiat se" - "chama Toro, tem 120" - "cavalos de potência e" - "o preço sugerido é" - "135 mil reais."

Cada fragmento, isoladamente, perde o contexto e a informação se torna fragmentada e confusa.

Parâmetros de um Text Splitter

Quando configuramos um Text Splitter, dois parâmetros são fundamentais: `chunk_size` e `chunk_overlap`.

- **Chunk Size**: Define o tamanho de cada pedaço de texto (chunk). Um tamanho maior pode manter mais contexto, mas também pode exceder a capacidade de processamento do modelo.
- **Chunk Overlap**: Permite que os chunks compartilhem uma parte do texto do chunk anterior. Isso ajuda a manter o contexto que poderia ser perdido entre um chunk e outro.

Tipos de Text Splitters

Vamos explorar alguns dos splitters mais comuns e como eles podem ser utilizados:

CharacterTextSplitter

Este é o tipo mais básico de splitter, que divide o texto baseado no número de caracteres.

```
from langchain.textsplitters import CharacterTextSplitter

char_splitter = CharacterTextSplitter(
    chunk_size=50,
    chunk_overlap=10,
    separator=' ')
texto = "Aqui vai um exemplo de texto que será dividido. Será que a divisão será boa?"
splits = char_splitter.split_text(texto)
print(splits)
```

RecursiveCharacterTextSplitter

Um splitter mais avançado que permite múltiplos separadores, respeitando uma hierarquia de divisão.

```
from langchain.textsplitters import RecursiveCharacterTextSplitter

rec_splitter = RecursiveCharacterTextSplitter(
    chunk_size=50,
    chunk_overlap=10,
    separators=['.', ' '])
texto = "Aqui vai um exemplo de texto que será dividido. Será que a divisão será boa?"
splits = rec_splitter.split_text(texto)
print(splits)
```

TokenTextSplitter

Este splitter considera tokens em vez de caracteres, o que é especialmente útil dado que muitos modelos de linguagem operam com tokens.

```
from langchain.textsplitters import TokenTextSplitter

token_splitter = TokenTextSplitter(chunk_size=30, chunk_overlap=5)
texto = "Aqui vai um exemplo de texto que será dividido. Será que a divisão será boa?"
splits = token_splitter.split_text(texto)
print(splits)
```

MarkdownHeaderTextSplitter

Ideal para dividir documentos em markdown baseando-se nos cabeçalhos.

```
from langchain.textsplitters import MarkdownHeaderTextSplitter

markdown_text = '''# Título do Markdown de exemplo
## Capítulo 1
Texto capítulo 1 e mais e mais texto.
Continuamos no capítulo 1!
## Capítulo 2
Texto capítulo 2 e mais e mais texto.
Continuamos no capítulo 2!
## Capítulo 3
### Seção 3.1
Texto capítulo 3 e mais e mais texto.
Continuamos no capítulo 3!
'''
header_to_split_on = [
    ('#', 'Header 1'),
    ('##', 'Header 2'),
    ('###', 'Header 3')
]
md_splitter = MarkdownHeaderTextSplitter(headers_to_split_on=header_to_split_on)
splits = md_splitter.split_text(markdown_text)
print(splits)
```

Dominar o Text Splitting é essencial para preparar seus dados para processamento eficiente em modelos de linguagem. Com a escolha certa de um Text Splitter e a configuração adequada dos parâmetros, você pode maximizar a relevância e a qualidade das informações processadas pela sua aplicação de IA.

14. Embeddings - Transformando texto em vetores

Agora vamos entender o conceito de **Embeddings**, o processo de transformar texto em vetores. Vamos aprender como um simples texto pode ser convertido em uma série de números que representam esse texto no espaço vetorial. Isso é crucial para muitas aplicações de IA, especialmente aquelas relacionadas à busca semântica e ao processamento de linguagem natural.

O que são Embeddings?

Os **Embeddings** são representações vetoriais de um pedaço de texto. Imagine que cada palavra ou frase é um ponto em um espaço de muitas dimensões. Esses pontos (ou vetores) podem ser analisados para entender a proximidade ou a semelhança semântica entre diferentes textos.

Por que isso é útil? Bem, ao transformar textos em vetores, podemos realizar operações matemáticas como calcular distâncias entre esses vetores. Isso nos permite fazer coisas como **busca semântica**, onde procuramos por pedaços de texto que são semelhantes entre si no espaço vetorial, ou seja, que estão mais próximos um do outro.

Como os Embeddings são criados?

A classe `Embeddings` do Langchain é projetada para interagir com modelos de embedding de texto de diferentes provedores como OpenAI, Cohere e Hugging Face. Esta classe fornece uma interface padrão para trabalhar com esses modelos, facilitando a integração em nossas aplicações.

Existem dois métodos principais na classe de Embeddings do LangChain:

1. **embed_documents**: Recebe como entrada vários textos e retorna seus embeddings.
2. **embed_query**: Recebe um único texto (geralmente uma pergunta ou consulta) e retorna seu embedding.

Exemplo Prático com OpenAI

Vamos ver como isso funciona na prática com um exemplo usando o modelo `text-embedding-ada-002` da OpenAI.

```
from langchain_openai import OpenAIEmbeddings

# Inicializando o modelo de embeddings
embedding_model = OpenAIEmbeddings(model='text-embedding-ada-002')

# Lista de textos para transformar em embeddings
```

```
textos = [
    'Eu gosto de cachorros',
    'Eu gosto de animais',
    'O tempo está ruim lá fora'
]

# Gerando embeddings para os documentos
embeddings = embedding_model.embed_documents(textos)

# Exibindo o tamanho e os primeiros elementos do primeiro embedding
print(len(embeddings[0]), embeddings[0][:10])
```

Análise de Similaridade

Após obter os embeddings, podemos calcular a similaridade entre eles usando o produto escalar (dot product). Textos semanticamente mais próximos terão um produto escalar maior, indicando maior proximidade no espaço vetorial.

```
import numpy as np

# Calculando a similaridade entre o primeiro e o segundo texto
similarity = np.dot(embeddings[0], embeddings[1])
print(f"Similaridade: {similarity}")
```

Embedding com HuggingFace

Também podemos usar modelos da Hugging Face para obter embeddings. Aqui está um exemplo usando o modelo `all-MiniLM-L6-v2`:

```
from langchain_community.embeddings.huggingface import HuggingFaceBgeEmbeddings

# Inicializando o modelo de embeddings da Hugging Face
hf_embedding_model = HuggingFaceBgeEmbeddings(model_name='all-MiniLM-L6-v2')

# Lista de textos para transformar em embeddings
textos = [
    'Eu gosto de cachorros',
    'Eu gosto de animais',
    'O tempo está ruim lá fora'
]

# Gerando embeddings
hf_embeddings = hf_embedding_model.embed_documents(textos)

# Calculando similaridades
for i in range(len(hf_embeddings)):
    for j in range(len(hf_embeddings)):
        print(round(np.dot(hf_embeddings[i], hf_embeddings[j]), 2), end=' | ')
    print()
```

Embeddings são uma ferramenta poderosa no mundo da IA, especialmente para aplicações que dependem de entender e processar linguagem natural. Ao transformar texto em vetores, abrimos um leque de possibilidades para análise semântica e recuperação de informação baseada no significado do conteúdo.

Nos vemos na próxima aula, onde exploraremos como armazenar e recuperar esses embeddings de forma eficiente.

15. VectorStores - Criando uma base de dados de vetores

Bem-vindos ao capítulo sobre VectorStores, uma parte crucial do nosso curso. Neste capítulo, vamos nos parofundar no mundo das bases de dados de vetores, que são essenciais para armazenar e buscar dados não estruturados de maneira eficiente. Vamos explorar como criar, salvar e utilizar essas bases usando exemplos práticos com as bibliotecas Chroma e FAISS.

O que é uma VectorStore?

Uma VectorStore é uma estrutura de dados especializada em armazenar vetores de embeddings e realizar buscas eficientes nesses vetores. O processo geral envolve converter textos em vetores usando um modelo de embeddings e, em seguida, armazenar esses vetores em uma VectorStore. Quando precisamos buscar informações relacionadas a uma consulta, convertemos essa consulta em um vetor e buscamos os vetores mais semelhantes na VectorStore.

Utilizando Chroma para criar uma VectorStore

A Chroma é uma das VectorStores mais populares dentro do ecossistema LangChain. Ela permite armazenar vetores de forma persistente e realizar buscas eficientes. Vamos ver como utilizá-la passo a passo.

Carregamento de Documentos

Primeiro, precisamos carregar os documentos dos quais extrairemos os textos para criar os embeddings.

```
from langchain_community.document_loaders.pdf import PyPDFLoader

caminho = "arquivos/Explorando o Universo das IAs com Hugging Face.pdf"
loader = PyPDFLoader(caminho)
paginas = loader.load()
```

Divisão de Texto (Text Splitting)

Após carregar os documentos, dividimos o texto em pedaços menores (chunks) que serão convertidos em vetores.

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

recur_split = RecursiveCharacterTextSplitter(
```

```
    chunk_size=500,
    chunk_overlap=50,
    separators=["\n\n", "\n", ".", " ", ""]
)

documents = recur_split.split_documents(paginas)
```

Criando a VectorStore com Chroma

Agora, vamos criar a VectorStore usando a Chroma, especificando onde os vetores serão persistidos.

```
from langchain_openai import OpenAIEmbeddings
from langchain_chroma import Chroma

embeddings_model = OpenAIEmbeddings()
diretorio = 'arquivos/chroma_vectorstore'

vectorstore = Chroma.from_documents(
    documents=documents,
    embedding=embeddings_model,
    persist_directory=diretorio
)
```

Buscando Informações

Para buscar informações relacionadas a uma consulta, primeiro convertemos a consulta em um vetor e depois realizamos uma busca por similaridade.

```
pergunta = 'O que é o Hugging Face?'
docs = vectorstore.similarity_search(pergunta, k=5)

for doc in docs:
    print(doc.page_content)
    print(f'====== {doc.metadata}\n\n')
```

Utilizando FAISS para criar uma VectorStore

FAISS, desenvolvido pelo Facebook AI, é outra opção popular para criar VectorStores eficientes.

Criando a VectorStore com FAISS

O processo é similar ao usado com a Chroma, mas usamos a biblioteca FAISS.

```
from langchain_community.vectorstores.faiss import FAISS

vectorstore = FAISS.from_documents(
```

```
    documents=documents,
    embedding=embeddings_model
)
```

Busca por Similaridade

Assim como com a Chroma, realizamos buscas por similaridade para encontrar os documentos mais relevantes.

```
docs = vectorstore.similarity_search(pergunta, k=5)

for doc in docs:
    print(doc.page_content)
    print(f'====== {doc.metadata}\n\n')
```

Salvando a VectorStore FAISS

```
vectorstore.save_local('arquivos/faiss_bd')
```

Carregando a VectorStore FAISS

```
vectorstore = FAISS.load_local(
    'arquivos/faiss_bd',
    embeddings=embeddings_model,
    allow_dangerous_deserialization=True
)
```

Espero que este capítulo tenha esclarecido como trabalhar com VectorStores e como elas são essenciais para aplicações de IA que envolvem o processamento de grandes volumes de texto. No próximo capítulo, vamos explorar técnicas avançadas de recuperação de informações.

16. Retrieval - Encontrando trechos relevantes

Neste capítulo vamos entender o processo de **Retrieval**, ou seja, a recuperação trechos de texto relevantes de uma base de dados, conhecida como **VectorStore**. Este processo é crucial para a técnica de **Retrieval Augmented Generation (RAG)**, onde o objetivo é enriquecer as respostas de um modelo de linguagem com informações pertinentes extraídas de uma base de dados.

O que é Retrieval?

Retrieval, em português, significa recuperação. No contexto de aplicações de IA, isso envolve buscar e recuperar trechos de texto que são mais relevantes para uma dada consulta ou pergunta feita pelo usuário. Esses trechos são então utilizados para ajudar o modelo de linguagem a fornecer respostas mais informadas e precisas.

Semantic Search

A busca semântica é a forma mais direta de retrieval. Ela compara a semelhança entre a query (pergunta do usuário) e os documentos na VectorStore. Vamos ver um exemplo de como isso é implementado:

```
from langchain_openai import OpenAIEmbeddings
from langchain_community.vectorstores.chroma import Chroma

# Supondo que 'documents' são os documentos já processados e prontos para serem inseridos na
↪  VectorStore
embeddings_model = OpenAIEmbeddings()
vectordb = Chroma.from_documents(documents, embedding=embeddings_model)

pergunta = 'O que é a OpenAI?'
docs = vectordb.similarity_search(pergunta, k=3)
for doc in docs:
    print(doc.page_content)
    print(f'==========={doc.metadata}\n\n')
```

Limitações do Semantic Search

Embora útil, a Semantic Search tem limitações, especialmente quando há trechos muito similares ou duplicados na base de dados. Isso pode levar a redundâncias nas informações recuperadas.

Max Marginal Relevance (MMR)

Para superar as limitações da Semantic Search, podemos usar o MMR, que busca não apenas relevância, mas também diversidade entre os trechos recuperados.

```
docs = vectordb.max_marginal_relevance_search(pergunta, k=3, fetch_k=10)
for doc in docs:
    print(doc.page_content)
    print(f'==========={doc.metadata}\n\n')
```

O parâmetro fetch_k representa uma busca semântica inicial. Ao utilizarmos o valor de dez como no exemplo, estamos fazendo uma busca semântica simples e encontrando os dez trechos mais próximos semanticamente a pergunta do usuário. A diferença do MMR surge na busca posterior que é feita, onde são recuperados entre estes dez trechos iniciais, dados que contenham uma disparidade grande, resultando nos três trechos (k=3) finais selecionados. Assim garantimos que haja uma dispersão e consequentemente uma maior quantidade de informações diferentes no dados recuperados.

Filtragem Avançada

Podemos também aplicar filtros para refinar os resultados de busca, como buscar apenas em fontes específicas ou páginas específicas.

```
docs = vectordb.similarity_search(
    pergunta,
    k=3,
    filter={'$and':
            [{'source': {'$in': ['Explorando o Universo das IAs com Hugging Face.pdf']}},
            {'page': {'$in': [10, 11, 12, 13]}}],
            })
)
for doc in docs:
    print(doc.page_content)
    print(f'==========={doc.metadata}\n\n')
```

LLM Aided Retrieval

Por fim, uma técnica avançada é o LLM Aided Retrieval, que utiliza modelos de linguagem para ajudar na criação de filtros de busca de forma dinâmica.

```
from langchain_openai.llms import OpenAI
from langchain.retrievers.self_query.base import SelfQueryRetriever
from langchain.chains.query_constructor.schema import AttributeInfo


metadata_info = [
    AttributeInfo(
        name='source',
        description='Nome da apostila de onde o texto original foi retirado. Deve ter o valor
↪  de: \
            Explorando o Universo das IAs com Hugging Face.pdf ou Explorando a API da
↪  OpenAI.pdf',
        type='string'
```

```
    ),
    AttributeInfo(
        name='page',
        description='A página da apostila de onde o texto se origina',
        type='integer'
    ),

]
document_description = 'Apostilas de cursos'

llm = OpenAI()
retriever = SelfQueryRetriever.from_llm(llm, vectordb, document_description, metadata_info,
↪  verbose=True)
docs = retriever.get_relevant_documents(pergunta)
for doc in docs:
    print(doc.page_content)
    print(f'==========={doc.metadata}\n\n')
```

Espero que este capítulo tenha esclarecido como o Retrieval funciona e como ele é crucial para enriquecer as respostas de um modelo de IA.

17. RAG - Conversando com os seus dados

Chegamos ao capítulo final do nosso curso “Aplicações de IA com Langchain”, onde vamos juntar todos os elementos que aprendemos sobre RAG em uma aplicação única e robusta. Este capítulo é como a cereja do bolo, onde vamos aplicar tudo que aprendemos para criar uma estrutura de perguntas e respostas interativa. Vamos lá!

Pipeline completo de RAG

Carregamento e dividindo documentos

Primeiro, vamos carregar e dividir os documentos que serão a base do nosso conhecimento. Utilizamos o `PyPDFLoader` para carregar os PDFs e o `RecursiveCharacterTextSplitter` para dividir o texto em pedaços manejáveis.

```
from langchain_community.document_loaders.pdf import PyPDFLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter

caminhos = [
    "arquivos/Explorando o Universo das IAs com Hugging Face.pdf",
    "arquivos/Explorando a API da OpenAI.pdf",
]

paginas = []
for caminho in caminhos:
    loader = PyPDFLoader(caminho)
    paginas.extend(loader.load())

recur_split = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=100,
    separators=["\n\n", "\n", ".", " ", ""]
)

documents = recur_split.split_documents(paginas)
```

Criação da base de dados de vetores

Após a divisão, os documentos são transformados em vetores usando embeddings da OpenAI e armazenados em uma `VectorStore` chamada FAISS.

```
from langchain_vectorstores import FAISS
from langchain_openai.embeddings import OpenAIEmbeddings

vectorstore = FAISS.from_documents(
    documents=documents,
```

```
    embedding=OpenAIEmbeddings()
)
```

Criando Estrutura de Conversa

Agora que temos nossa base de dados de vetores, vamos criar uma estrutura de conversa que utilizará essa base para responder perguntas.

```
from langchain_core.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_template(
    '''Responda as perguntas se baseando no contexto fornecido.

    contexto: {contexto}

    pergunta: {pergunta}''')
```

Configurando o Retriever

Vamos configurar o retriever, que é um Runnable que, quando invocado, realiza uma busca na Vector Store de acordo com os parâmetros definidos em sua criação. Este retriever buscará os documentos relevantes para a pergunta feita pelo usuário:

```
from langchain_core.runnables import RunnableParallel, RunnablePassthrough

retriever = vectorstore.as_retriever(search_type='mmr', search_kwargs={'k': 5, 'fetch_k': 25})
setup = RunnableParallel({
    'pergunta': RunnablePassthrough(),
    'contexto': retriever
})
```

Invocando a Estrutura de Conversa

Agora, podemos invocar a estrutura de conversa com uma pergunta:

```
input = setup.invoke('O que é a OpenAI?')
print(input)
```

A saída será um dicionário contendo a pergunta e os documentos relevantes que foram recuperados:

```
{'pergunta': 'O que é a OpenAI?',
 'contexto': [Document(...), Document(...), ...]}
```

Juntando os Documentos

Para facilitar a resposta, vamos juntar os conteúdos dos documentos recuperados em uma única string:

```
def join_documents(input):
    input['contexto'] = '\n\n'.join([c.page_content for c in input['contexto']])
    return input

setup = RunnableParallel({
    'pergunta': RunnablePassthrough(),
    'contexto': retriever
}) | join_documents
```

Finalizando a Chain

Agora, podemos criar a chain final que combina a recuperação de documentos e o prompt:

```
chain = setup | prompt | ChatOpenAI()
```

Testando a Aplicação

Vamos testar nossa aplicação invocando a chain com uma pergunta:

```
response = chain.invoke('O que é a OpenAI?')
print(response)
```

A saída será uma resposta gerada pelo modelo, que pode incluir informações relevantes sobre a OpenAI, como:

```
AIMessage(content='A OpenAI é a maior desenvolvedora de ferramentas de inteligên
```

Agora você já aprendeu como unir todos os componentes de RAG para criar uma aplicação de conversação baseada em documentos. Isto fecha todas as ferramentas essenciais de LangChain e também encerra nosso curso. Espero que vocês tenham gostado do conteúdo e nos vemos nos próximos cursos e projetos da Asimov Academy!