Aplicações de 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 RouterChains**: 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 - Padronizando a resposta do modelo

Output Parsers são ferramentas essenciais para quem combina modelos de linguagem a programação Python. Vamos entender como esses parsers podem ajudar a transformar respostas textuais em estruturas de dados que facilitam a manipulação e integração em nossas aplicações.

O que são Output Parsers?

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 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.

Exemplo Prático

Vamos a um exemplo prático para ilustrar como isso funciona. Suponha 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."

Queremos que o modelo processe esta review e estruture a informação no seguinte formato JSON:

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

Implementando um Output Parser

Com o uso de Output Parsers, definimos esquemas de resposta (`ResponseSchema`) para cada informação que desejamos extrair. Em seguida, criamos um `StructuredOutputParser` a partir desses esquemas. Isso nos permite instruir o modelo de linguagem sobre como queremos que a resposta seja formatada.

```
from langchain.output_parsers import ResponseSchema, StructuredOutputParser

schema_presente = ResponseSchema(
    name='presente',
    type='bool',
    description='O item foi comprado para algém? True caso verdadeiro \
e False caso falos ou não tenha a infomação.'
)

schema_entrega = ResponseSchema(
    name='dias_entrega',
    type='int',
    description='Quantos dias para a entrega chegar?\
Se a resposta não for encontrada, retorne -1.'
)

schema_valor = ResponseSchema(
    name='percepcao_de_valor',
    type='list',
    description='Extraia qualquer frase sobre o \
valor ou preço do produto. Retorne como uma lista \
de Python.'
)

response_schema = [schema_presente,
                   schema_entrega,
                   schema_valor]
output_parser = StructuredOutputParser.from_response_schemas(response_schema)
schema_formatado = output_parser.get_format_instructions()
print(schema_formatado)
```

```
The output should be a markdown code snippet formatted in the following schema,

\```json
{
    "presente": bool  // O item foi comprado para algém? True caso verdadeiro e
    "dias_entrega": int  // Quantos dias para a entrega chegar?Se a resposta não
    "percepcao_de_valor": list  // Extraia qualquer frase sobre o valor ou preço
}
\```
```

Com esses esquemas, o modelo pode gerar respostas estruturadas que são facilmente parseadas e utilizadas em nossa aplicação. Isso não só economiza tempo como também reduz a probabilidade de erros.

Adicionando Esquema Criado ao meu Prompt

Dado que temos um esquema que representa o formato da saída esperado do modelo, podemo adicioná-lo ao nosso prompt:

```
from langchain.prompts import ChatPromptTemplate

review_template = ChatPromptTemplate.from_template("""
Para o texto a seguir, extraia as seguintes informações:

presente, dias_entrega e percepcao_de_valor

Texto: {review}

{schema}
""", partial_variables={'schema': schema_formatado})

print(review_template.format_messages(review=review_cliente))
```

E depois podemos fazer a chamada ao modelo:

```
from langchain_openai.chat_models import ChatOpenAI

chat = ChatOpenAI()
resposta = chat.invoke(review_template.format_messages(review=review_cliente))
resposta
```

```
AIMessage(content='```json\n{\n\t"presente": true,\n\t"dias_entrega": 2,\n\t"perc
```

Analisando a saída

Podemos perceber que o resultando ainda está no formato de texto, o que é esperado já que o um modelo de linguagem sempre reponde em texto:

```
print(resposta.content)
```

```
\```json
{
    "presente": true,
    "dias_entrega": 2,
    "percepcao_de_valor": ["É um pouco mais caro do que os outros sopradores de
}
\```
```

Convertendo a Saída em um Dicionário de Python

Podemos utilizar o método *parse* do *Output Parser* para converter a saída em um dicionário em Python:

```
resposta_json = output_parser.parse(resposta.content)
resposta_json['presente']
```

```
True
```

Os *Output Parsers* são ferramentas poderosas que nos ajudam a tirar o máximo proveito dos modelos de linguagem, transformando texto não estruturado em dados estruturados prontos para uso.

06. Memory - Adicionando memória à conversa com o modelo

Agora vamos tratar de um conceito fundamental na criação de aplicações voltadas a conversas: a **Memória**. 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.

O que é Memória em LangChain?

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.

Funcionamento das Memórias em LangChain

Para utilizar uma memória em LangChain, basta instanciá-la e ela já estará armazenando todas interações que passarem por ela

```
from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory()
```

Adicionando Valores Manualmente a Memória

Podemos utilizar os métodos *chat_memory.add_user_message* e *chat_memory.add_ai_message* para adicionar manualmente dados a memória. Isto pode ser relevante ao criarmos uma aplicação com Few-Shot Prompting, por exemplo:

```
memory.chat_memory.add_user_message('Oi?')
memory.chat_memory.add_ai_message('Como vai?')
```

Visualizando Valores Salvos na Memória

```
memory.load_memory_variables({})
```

```
{'history': 'Human: Oi?\nAI: Como vai?'}
```

Armazenando Memória no Formato de Chat

Podemos utilizar o parâmetro return_messages=True para armazenar nossa memória no formato de Chat padrão do LangChain:

```
from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory(return_messages=True)
memory.chat_memory.add_user_message('Oi?')
memory.chat_memory.add_ai_message('Como vai?')
memory.load_memory_variables({})
```

```
{'history': [HumanMessage(content='Oi?'), AIMessage(content='Como vai?')]}
```

Integrando Memória com Chains

A beleza do LangChain está em como esses componentes podem ser integrados. Por exemplo, você pode usar qualquer um dos tipos de memória em uma `ConversationChain` para manter o contexto durante uma sessão de chat.

```
from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferMemory
from langchain_openai.chat_models import ChatOpenAI

chat_model = ChatOpenAI()
memory = ConversationBufferMemory()
chain = ConversationChain(
    llm=chat,
    memory=memory,
    verbose=True
    )
chain.predict(input='Olá! Meu nome é Adriano')
```

```
> Entering new ConversationChain chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is ta

Current conversation:

Human: Olá! Meu nome é Adriano
AI:

> Finished chain.

'Olá, Adriano! Como posso ajudá-lo hoje?'
```

```
conversation.predict(input='Qual é o meu nome?')
```

```
> Entering new ConversationChain chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is ta

Current conversation:
Human: Olá! Meu nome é Adriano
AI: Olá, Adriano! Como posso ajudá-lo hoje?
Human: Qual é o meu nome?
AI:

> Finished chain.

'Seu nome é Adriano. Como posso ajudá-lo hoje, Adriano?'
```

Tipos de Memória em LangChain

LangChain oferece diferentes tipos de memórias, cada uma adequada para necessidades específicas. Vamos explorar algumas delas:

ConversationBufferMemory

Este é o tipo mais simples de memória, onde as mensagens são armazenadas sequencialmente. É útil para manter um histórico simples de mensagens.

```
from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory()
```

ConversationBufferWindowMemory

Este tipo de memória armazena uma "janela" das últimas interações, limitando o histórico a um número específico de trocas de mensagens. Isso é útil para manter o contexto relevante sem sobrecarregar o modelo.

```
from langchain.memory import ConversationBufferWindowMemory

memory = ConversationBufferWindowMemory(k=5)  # Armazena as últimas 5 interações
```

ConversationTokenBufferMemory

Diferente do tipo anterior, este se baseia no número de tokens (unidades básicas de texto para o modelo) ao invés de mensagens. Isso ajuda a manter o prompt dentro dos limites aceitáveis para o modelo processar.

```
from langchain.memory import ConversationTokenBufferMemory

memory = ConversationTokenBufferMemory(max_token_limit=250)
```

ConversationSummaryBufferMemory

Quando o histórico se torna muito extenso, essa memória utiliza um modelo de linguagem para resumir o histórico, reduzindo o número de tokens usados e mantendo apenas as informações mais relevantes.

```
from langchain.memory import ConversationSummaryBufferMemory

memory = ConversationSummaryBufferMemory(max_token_limit=250)
```

A memória não é apenas um recurso técnico, mas uma parte vital que traz inteligência e continuidade às interações com modelos de linguagem. Ao escolher o tipo de memória adequado, você pode otimizar a performance e a relevância das respostas do modelo, tornando a experiência do usuário mais rica e envolvente.

07. Chains - Encadeamento de Prompts 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 diversos Prompts de maneira estratégica para resolver problemas maiores e mais complexos.

O que são Chains?

Chains, ou Cadeias, são sequências de Prompts 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.

Tipos de Chains

Vamos explorar alguns tipos de Chains que você pode utilizar para construir suas aplicações:

ConversationChain

A `ConversationChain` é usada para manter uma conversa fluente e contextual com o usuário. Ela pode armazenar memória da conversa para manter o contexto e responder de forma mais coerente.

```
from langchain_openai.chat_models import ChatOpenAI
from langchain.memory import ConversationBufferMemory
from langchain.chains.conversation.base import ConversationChain

chat = ChatOpenAI(model='gpt-3.5-turbo-0125')
memory = ConversationBufferMemory()
chain = ConversationChain(
    llm=chat,
    memory=memory,
    verbose=True
```

```
)

chain.predict(input='Oi')
```

LLMChain

A `LLMChain` é a forma mais básica de Chain, onde um único Prompt é processado. É útil para tarefas diretas onde uma única entrada e saída são necessárias.

```
from langchain_openai.chat_models import ChatOpenAI
from langchain.chains.llm import LLMChain
from langchain.prompts import PromptTemplate


chat = ChatOpenAI(model='gpt-3.5-turbo-0125')
prompt = PromptTemplate.from_template(
    "Qual o melhor nome de empresa para uma empresa que desenvolve o produto: {produto}."
)

chain = LLMChain(llm=chat, prompt=prompt)
produto = 'Copos de vidro inquebráveis'
chain.run(produto)
```

`VidroShield`

SimpleSequentialChain

A `SimpleSequentialChain` permite combinar várias Chains em uma sequência, onde a saída de uma Chain se torna a entrada da próxima.

```
from langchain.chains import SimpleSequentialChain
from langchain.prompts import PromptTemplate

prompt = PromptTemplate.from_template(
'''Qual o melhor nome de empresa para
uma empresa que desenvolve o produto: {produto}.
Retorne apenas um nome.'''
)

chain_nome = LLMChain(llm=chat, prompt=prompt)

prompt_descricao = PromptTemplate.from_template(
'''Dado a empresa com o seguinte nome: {nome_empresa}.
Dê uma deswcrição de até 50 palavras das atividades
dessa empresa.'''
)

chain_descricao = LLMChain(llm=chat, prompt=prompt_descricao)
```

```
chain = SimpleSequentialChain(
    chains=[chain_nome, chain_descricao],
    verbose=True
)
produto = 'Copos de vidro inquebráveis'
chain.run(produto)
```

```
> Entering new SimpleSequentialChain chain...
Vidro Resistente Ltda.
A Vidro Resistente Ltda. é uma empresa especializada na fabricação e comercializ

> Finished chain.
'A Vidro Resistente Ltda. é uma empresa especializada na fabricação e comerciali
```

SequentialChain

Similar à `SimpleSequentialChain`, mas mais flexível, permitindo que as Chains não sejam apenas sequenciais, mas também paralelas, dependendo das necessidades da aplicação.

```
from langchain.chains import SequentialChain
from langchain.prompts import PromptTemplate

prompt = PromptTemplate.from_template(
'''Qual o melhor nome de empresa para
uma empresa que desenvolve o produto: {produto}.
Retorne apenas um nome.'''
)

chain_nome = LLMChain(llm=chat, prompt=prompt, output_key='nome_empresa')

prompt_descricao = PromptTemplate.from_template(
'''Dado a empresa com o seguinte nome: {nome_empresa}
e que produz o seguinte produto {produto}.
Dê uma deswcrição de até 50 palavras das atividades
dessa empresa.'''
)

chain_descricao = LLMChain(llm=chat, prompt=prompt_descricao, output_key='descricao_empresa')

prompt_traducao = PromptTemplate.from_template(
'''Crie um nome em inglês para a empresa de nome {nome_empresa}
que possui a seguinte descrição {descricao_empresa}'''
)

chain_traducao = LLMChain(llm=chat, prompt=prompt_traducao, output_key='nome_ingles', )

chain = SequentialChain(
    chains=[chain_nome, chain_descricao, chain_traducao],
```

```
    input_variables=['produto'],
    output_variables=['nome_empresa', 'descricao_empresa', 'nome_ingles'],
    verbose=True
)
produto = 'Copos de vidro inquebráveis'
resposta = chain.invoke({'produto': produto})
resposta
```

```
{'produto': 'Copos de vidro inquebráveis', 'nome_empresa': 'Vidro Resistente Inc
```

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.

08. RouterChains - Cadeias de Roteamento

Vamos falar agora de um outro tipo muito especial de chain do LangChain, as RouterChains, ou Cadeias de Roteamento, uma ferramenta incrível para direcionar as interações de forma inteligente e eficaz.

O que são RouterChains?

Imagine que você está desenvolvendo uma aplicação de IA que precisa lidar com diferentes tipos de perguntas, cada uma exigindo uma abordagem específica. Aqui entram as RouterChains! Elas são estruturas que ajudam a decidir para qual cadeia especializada uma determinada entrada do usuário deve ser enviada. Isso permite que a aplicação responda de maneira mais adequada e eficiente às necessidades do usuário.

Por exemplo, se um usuário faz uma pergunta sobre física, a RouterChain direciona essa pergunta para uma cadeia especializada em responder questões de física. Se a pergunta for sobre história, ela será enviada para uma cadeia focada em história. Isso tudo é feito automaticamente pela RouterChain, baseando-se na análise da entrada do usuário.

Como Funciona na Prática?

Para ilustrar, vamos considerar uma aplicação que possui três cadeias especializadas: física, matemática e história. A RouterChain inicial analisa a pergunta e decide qual dessas três cadeias é a mais adequada para responder. Isso é feito através de um conjunto de templates e uma configuração específica que vamos explorar agora.

Criando uma Cadeia de Roteamento

Importação das Bibliotecas Necessárias:

```
from langchain_openai import ChatOpenAI
from langchain.prompts import ChatPromptTemplate
from langchain.chains.llm import LLMChain
from langchain.chains.router import MultiPromptChain
from langchain.chains.router.llm_router import LLMRouterChain, RouterOutputParser
from langchain.chains.router.multi_prompt_prompt import MULTI_PROMPT_ROUTER_TEMPLATE
```

Criação dos Templates Especializados:

Cada área tem um template que define como o modelo deve responder às perguntas relevantes.

```
fis_template = ChatPromptTemplate.from_template("""Você é um professor de física muito
↪  inteligente. Você é ótimo em responder perguntas sobre física de forma concisa e fácil de
↪  entender. Quando você não sabe a resposta para uma pergunta, você admite que não sabe.
Aqui está uma pergunta: {input}""")

mat_template = ChatPromptTemplate.from_template("""Você é um matemático muito bom. Você é
↪  ótimo em responder perguntas de matemática. Você é tão bom porque consegue decompor
↪  problemas difíceis em suas partes componentes, responder às partes componentes e depois
↪  juntá-las para responder à pergunta mais ampla.
Aqui está uma pergunta: {input}""")

hist_template = ChatPromptTemplate.from_template("""Você é um historiador muito bom. Você tem
↪  um excelente conhecimento e compreensão de pessoas, eventos e contextos de uma variedade
↪  de períodos históricos. Você tem a capacidade de pensar, refletir, debater, discutir e
↪  avaliar o passado. Você tem respeito pela evidência histórica e a capacidade de usá-la
↪  para apoiar suas explicações e julgamentos.
Aqui está uma pergunta: {input}""")
```

Definição das Informações dos Prompts:

Criamos uma lista de dicionários que contém informações sobre cada área de conhecimento.

```
prompt_infos = [
   {'name': 'Fisica', 'description': 'Ideal para responder perguntas sobre física',
↪  'prompt_template': fis_template},
   {'name': 'Matematica', 'description': 'Ideal para responder perguntas sobre matemática',
↪  'prompt_template': mat_template},
   {'name': 'Historia', 'description': 'Ideal para responder perguntas sobre história',
↪  'prompt_template': hist_template},
]
```

Configuração das Cadeias de Destino:

Para cada área, configuramos uma cadeia que será responsável por processar as perguntas direcionadas a ela.

```
chat = ChatOpenAI(model='gpt-3.5-turbo-0125')
chains_destino = {}
for info in prompt_infos:
   chain = LLMChain(llm=chat, prompt=info['prompt_template'], verbose=True)
   chains_destino[info['name']] = chain
```

Criação da RouterChain:

Utilizamos o template de roteamento para criar a RouterChain, que decidirá para qual cadeia de destino enviar a entrada do usuário.

```
destinos = [f'{p["name"]}: {p["description"]}' for p in prompt_infos]
destinos_str = '\n'.join(destinos)
router_template = MULTI_PROMPT_ROUTER_TEMPLATE.format(
    destinations=destinos_str
)
```

```
router_template = PromptTemplate(
    template=router_template,
    input_variables=['input'],
    output_parser=RouterOutputParser()
)

router_chain = LLMRouterChain.from_llm(chat, router_template, verbose=True)
```

Chain Completa:

Finalmente, integramos tudo em uma MultiPromptChain, que inclui a RouterChain e as cadeias de destino.

```
default_prompt = ChatPromptTemplate.from_template('{input}')
default_chain = LLMChain(llm=chat, prompt=default_prompt, verbose=True)

chain = MultiPromptChain(
    router_chain=router_chain,
    destination_chains=chains_destino,
    default_chain=default_chain,
    verbose=True)
```

Exemplo de Uso

Vamos ver como essa configuração funciona com alguns exemplos práticos:

Pergunta sobre Física:

```
chain.invoke({'input': 'O que é um buraco negro?'})
```

```
> Entering new MultiPromptChain chain...


> Entering new LLMRouterChain chain...

> Finished chain.
Fisica: {'input': 'O que é um buraco negro?'}

> Entering new LLMChain chain...
Prompt after formatting:
Human: Você é um professor de física muito inteligente.
Você é ótimo em responder perguntas sobre física de forma concisa
e fácil de entender.
Quando você não sabe a resposta para uma pergunta, você admite
que não sabe.
```

```
Aqui está uma pergunta: O que é um buraco negro?

> Finished chain.

> Finished chain.
{'input': 'O que é um buraco negro?',
 'text': 'Um buraco negro é uma região do espaço onde a gravidade é tão intensa
```

Pergunta sobre Matemática:

```
chain.invoke({'input': 'O que é uma equação quadrática?'})
```

```
> Entering new MultiPromptChain chain...


> Entering new LLMRouterChain chain...

> Finished chain.
Matematica: {'input': 'O que é uma equação quadrática?'}

> Entering new LLMChain chain...
Prompt after formatting:
Human: Você é um matemático muito bom.
Você é ótimo em responder perguntas de matemática.
Você é tão bom porque consegue decompor
problemas difíceis em suas partes componentes,
responder às partes componentes e depois juntá-las
para responder à pergunta mais ampla.

Aqui está uma pergunta: O que é uma equação quadrática?

> Finished chain.

> Finished chain.
{'input': 'O que é uma equação quadrática?',
 'text': 'Uma equação quadrática é uma equação polinomial de segundo grau, ou se
```

Espero que tenham gostado deste capítulo sobre RouterChains! Com essa ferramenta, vocês podem criar aplicações de IA mais inteligentes e responsivas.

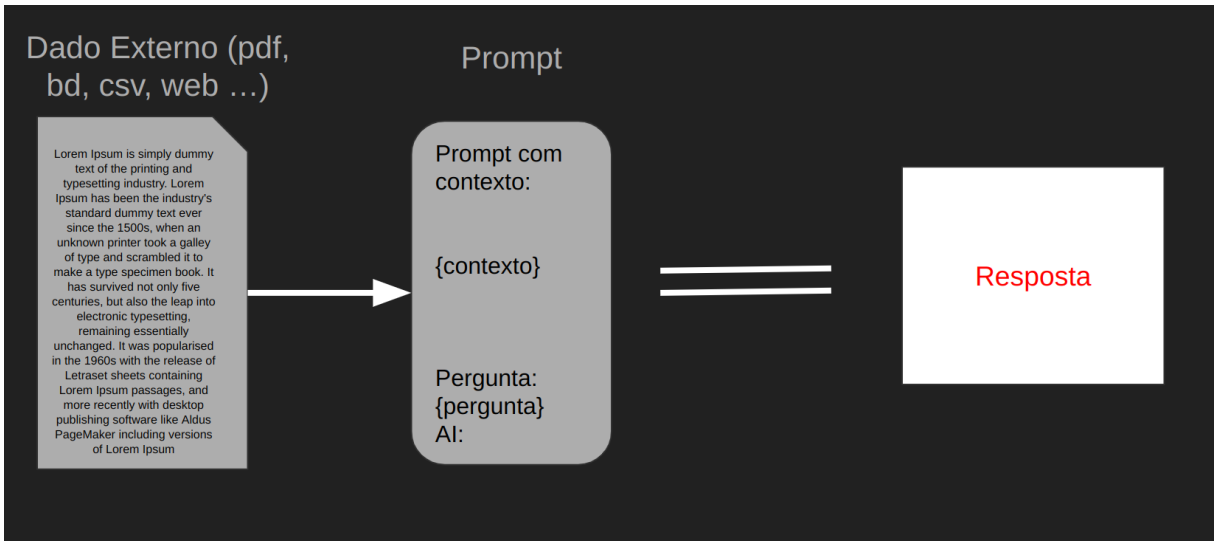
09. RAG - Introdução à técnica Retrieval Augmented Generation

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.

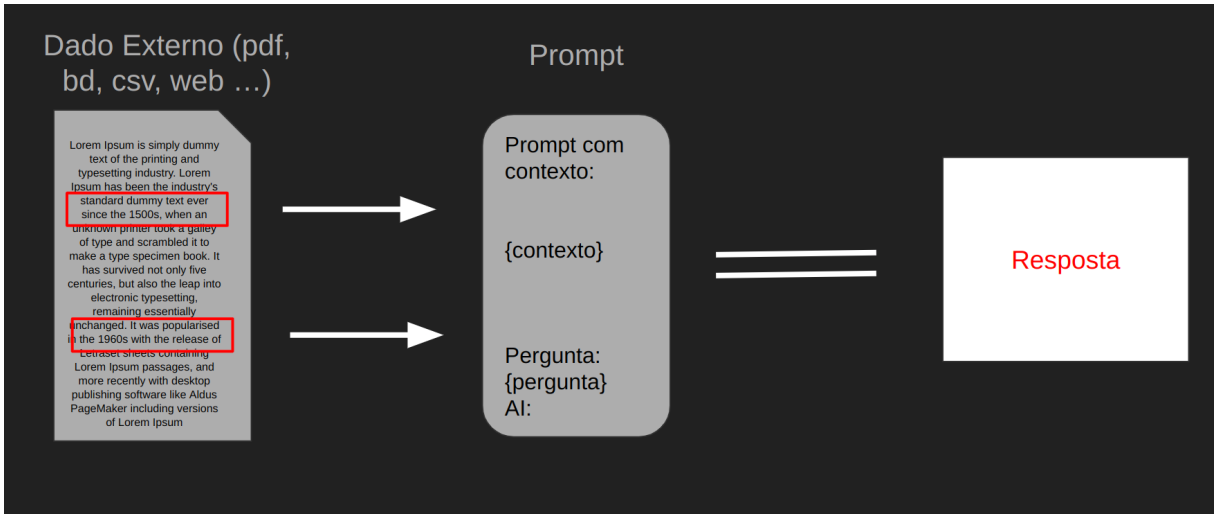


Desafios ao Utilizar RAG

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.

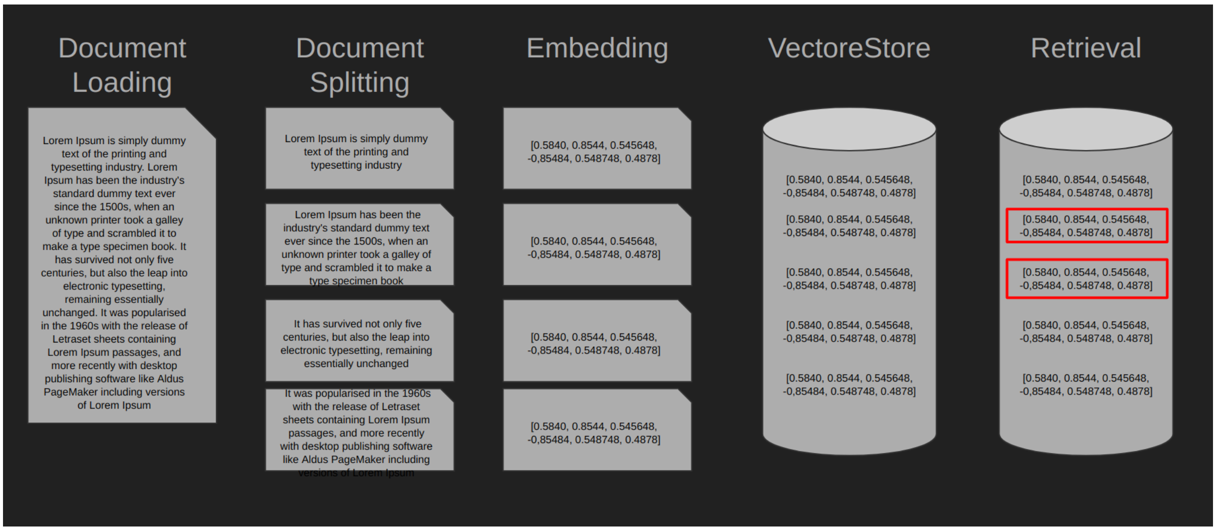
Além disso, a qualidade do embedding dos chunks é crucial, pois vetores numéricos mal construídos podem não capturar a semântica correta do texto, prejudicando a capacidade do modelo de encontrar as informações corretas. Portanto, é essencial utilizar técnicas de embedding avançadas que possam representar 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. Nos próximos capítulos, vamos explorar cada uma dessas etapas em detalhes, preparando você para construir suas próprias aplicações incríveis com Langchain e Python.

10. Document Loaders - Carregando dados com Langchain

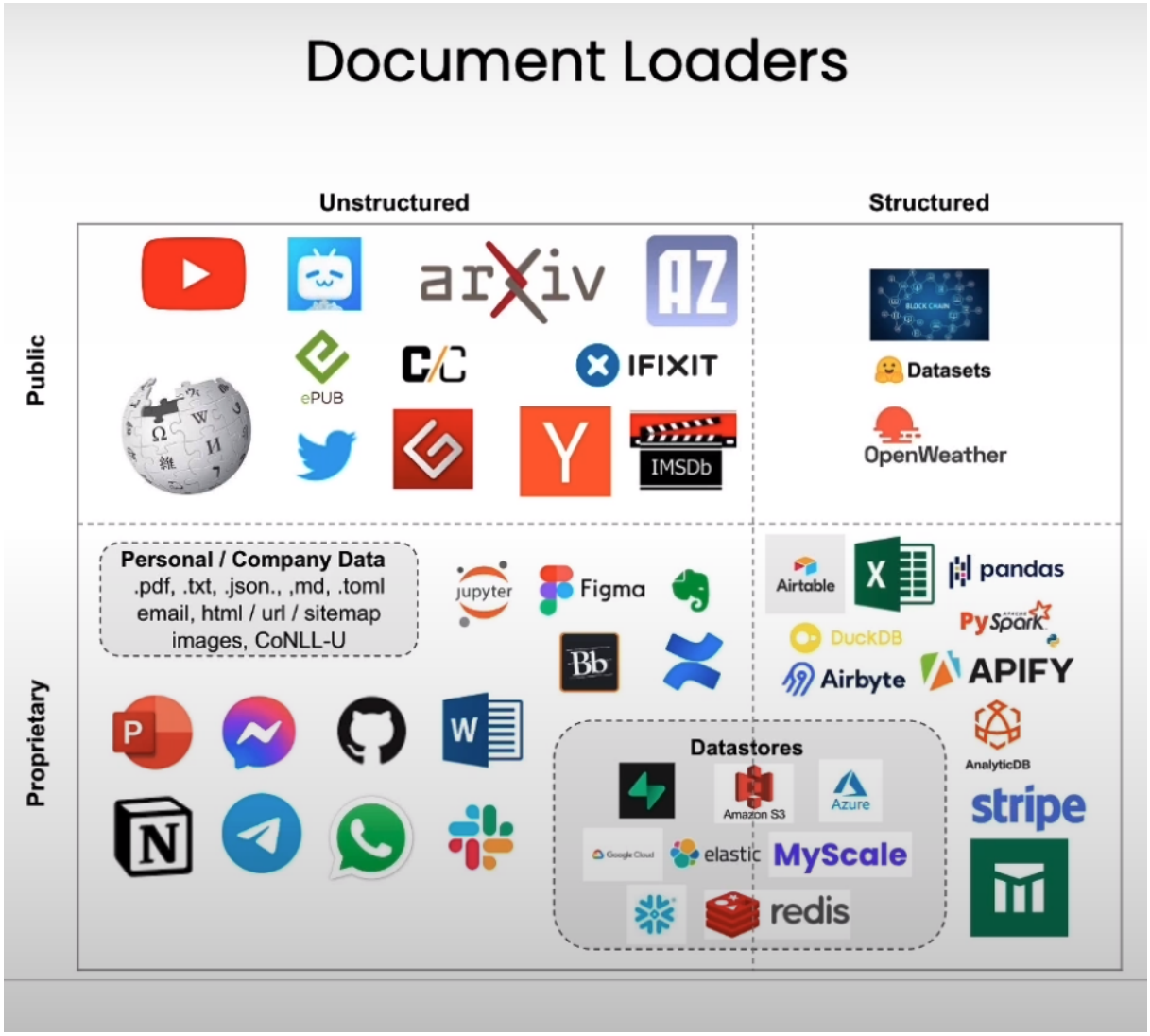
Vamos agora entender RAG com mais profundidade, começando pela sua primeira etapa: Document Loading.

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)
```

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.

11. 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.

12. 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.

13. 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.

14. 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.

15. 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 (Retrieval Augmented Generation) 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!

Processo Completo de RAG

Carregamento e Divisão de 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 Chroma.

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

embeddings_model = OpenAIEmbeddings()
vectordb = Chroma.from_documents(
```

```
    documents=documents,
    embedding=embeddings_model,
    persist_directory='arquivos/chat_retrieval_db'
)
```

Estrutura de Conversa com RAG

Com os documentos carregados e vetorizados, criamos uma estrutura de conversa usando `RetrievalQA`, que permite fazer perguntas e obter respostas baseadas nos documentos.

```
from langchain_openai.chat_models import ChatOpenAI
from langchain.chains.retrieval_qa.base import RetrievalQA

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

chat_chain = RetrievalQA.from_chain_type(
    llm=chat,
    retriever=vectordb.as_retriever(search_type='mmr'),
)
```

Fazendo Perguntas

Agora, podemos fazer perguntas diretamente aos nossos dados. A chain de RAG cuida de buscar os documentos relevantes e passá-los ao modelo de linguagem para gerar uma resposta.

```
pergunta = 'O que é Hugging Face e como faço para acessá-lo?'
resposta = chat_chain.invoke({'query': pergunta})
print(resposta['result'])
```

Customizando o Prompt

Podemos também customizar o prompt usado na chain para ajustar a forma como a pergunta é apresentada ao modelo.

```
from langchain.prompts import PromptTemplate

chain_prompt = PromptTemplate.from_template(
"""Utilize o contexto fornecido para responder a pergunta ao final.
Se você não sabe a resposta, apenas diga que não sabe e não tente inventar a resposta.
Utilize três frases no máximo, mantenha a resposta concisa.

Contexto: {context}

Pergunta: {question}

Resposta:
"""
```

```
)

chat_chain = RetrievalQA.from_chain_type(
    llm=chat,
    retriever=vectordb.as_retriever(search_type='mmr'),
    chain_type_kwargs={'prompt': chain_prompt},
    return_source_documents=True
)
```

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!