Agents de IA com Python e LangChain

Asimov Academy

ASIMOV

Conteúdo

01. Bem-vindos ao curso

Bem-vindos a mais um curso da Asimov!

Este é o curso de Agentes de IA com Python e LangChain, e estamos muito felizes em trazer este conteúdo novo para vocês. Vocês já devem ter percebido que acreditamos muito nas IAs como uma forma revolucionária de lidar com o trabalho e com o acesso à informação. Estamos iniciando uma nova era e quebrando diversos paradigmas. Nessa linha, os agentes entram como componentes principais dessa revolução.

Essas estruturas são capazes de raciocínios similares aos humanos, acesso a informações praticamente ilimitadas, capacidade de agir no mundo e buscar informações em tempo real. Tudo isso de uma forma autônoma, ou seja, após desenvolvido o meu agente, ele será capaz de se adequar às novas tarefas e necessidades que forem surgindo!

Tudo isso parece bom demais, mas eu digo que estamos apenas no início. Recém começamos a explorar as capacidades dessas estruturas e, para ser sincero, ainda não sei do que as aplicações com agentes serão capazes no futuro. Mas uma coisa eu tenho certeza: em um futuro próximo, teremos contato diário com os mais diferentes tipos de agentes! E nada melhor do que começar agora a entender o que são agentes, como construí-los e já adicioná-los às minhas aplicações hoje.

Façamos um futuro que começa agora!

Como o curso está estruturado?

Teremos 4 partes principais no nosso curso:

1) LCEL - LangChain Expression Language

As primeiras aulas são para apresentar o LangChain Expression Language, ou LCEL. Esta é uma forma declarativa e intuitiva de compor chains, de maneira fácil e escalável e será a base para criarmos as chains dos nossos agents!

2) Adição de Funções Externas e Aplicações

Vamos ver neste módulo como criar funções externas que estarão disponíveis para o modelo de linguagem utilizar quando necessário. Essas estruturas aumentarão as capacidades dos modelos, pois poderão gerar informações atualizadas ou atuar no mundo de forma customizada. Neste módulo entenderemos como criar jsons específicos, com boas descrições das funções e dos argumentos, que aumentam as chances do modelo reconhecer e utilizá-las nos momentos corretos.

3) Criação de Tools

As tools, ou ferramentas, são o passo dois das funções. No momento que o modelo consegue reconhecer que existe uma função externa que ele pode utilizar, podemos de fato utilizá-la, executá-la e oferecer as obersvações que a ferramenta gera ao modelo. As tools são os grandes diferenciais dos Agents, quanto melhores as ferramentas, melhor o Agent!

4) Criação de Agents

A última etapa é de finalmente a criação de agents. Juntamos todos os conhecimentos obtidos nos módulos iniciias e entendemos o que é um agent, como construí-los manulamente e como criar agents avançados, como um agent que analisa bancos de dados SQL e agents que analisam dados de DataFrames!

Pré-requisitos para o curso

E é importante ressaltar que será necessário um conhecimento básico de LangChain para compreender este curso. Então, caso você ainda não tenha feito nosso curso introdutório de Aplicações de IA com LangChain, pare agora, assiste ele com calma e depois volte para cá. Caso contrário, então já pode começar agora a explorar o incrível potencial dos agents!

E agora, sem mais delongas, vamos para nosso curso!

02. Criando uma chain com LCEL

Neste capítulo, vamos entender o que é o LangChain Expression Language (LCEL) e como utilizá-lo para criação de chains.

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).

Principais Características da LCEL

- **Suporte a Streaming**: Isso significa que você pode começar a receber resultados assim que o primeiro token é produzido, sem esperar pelo processamento completo.
- **Execução Paralela**: Se sua chain tem etapas que podem ser executadas simultaneamente, a LCEL cuida disso automaticamente.
- **Fallbacks e Retentativas**: Em caso de erros, você pode definir ações alternativas para garantir a confiabilidade de suas chains.
- **Acesso a Resultados Intermediários**: Essencial para depuração, permitindo que você entenda e refine cada etapa da sua chain.

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 nossos corações
↪  com seu charme e fofura.', response_metadata={'token_usage': {'completion_tokens': 28,
↪  'prompt_tokens': 19, 'total_tokens': 47}, 'model_name': 'gpt-3.5-turbo-0125',
↪  'system_fingerprint': None, 'finish_reason': 'stop', 'logprobs': None},
↪  id='run-b9e445f5-5ba9-4aea-9813-eda76880def5-0')
```

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.

Métodos de um Runnable

Agora vamos explorar mais a fundo os métodos dos Runnables de LCEL, que são essenciais para a utilização das chains que agora sabemos criar. Vamos entender como esses métodos funcionam e como você pode utilizá-los para tornar suas aplicações mais dinâmicas e eficientes. Mas primeiro, precisamos entender o que são os tais dos Runnables!

O que são Runnables?

Runnables são componentes modulares que formam uma Chain. Eles são, essencialmente, estruturas que podem ser "rodadas" ou executadas. Quando criamos uma Chain usando LCEL, estamos montando uma sequência de Runnables que trabalham juntos para processar dados ou realizar tarefas.

Métodos Principais dos Runnables

Os Runnables possuem três métodos principais que você precisa conhecer: `invoke`, `stream`, e `batch`. Cada um desses métodos tem uma versão assíncrona, que são `ainvoke`, `astream`, e `abatch`. Vamos detalhar cada um deles:

Invoke O método `invoke` é o mais básico e é usado para inserir uma entrada na Chain e obter uma resposta. É bastante direto e você já deve estar familiarizado com ele de nossas aulas anteriores.

```
# Exemplo de uso do método invoke
chain.invoke({'assunto': 'cachorrinhos'})
```

Stream O método `stream` é usado quando queremos uma interação mais dinâmica, como em uma conversa. Ele permite que a saída seja transmitida de volta conforme é gerada pelo modelo, criando uma experiência mais fluida para o usuário.

```
# Exemplo de uso do método stream
for chunk in chain.stream({'assunto': 'cachorrinho'}):
    print(chunk.content, end='', flush=True)
```

Batch O método `batch` é utilizado para processar múltiplas entradas em paralelo. Isso é especialmente útil quando você precisa lidar com várias requisições ao mesmo tempo, otimizando o tempo de resposta.

```
# Exemplo de uso do método batch
chain.batch([{'assunto': 'cachorrinhos'}, {'assunto': 'gatinho'}, {'assunto': 'cavalinhos'}])
```

Métodos Assíncronos

Os métodos assíncronos são uma extensão dos métodos síncronos 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.

```
# Exemplo de uso do método ainvoke
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
```

Espero que você tenha gostado desta introdução à LCEL e esteja tão empolgado quanto eu para explorar mais sobre o que podemos fazer com essa poderosa ferramenta. No próximo capítulo, vamos explorar como adicionar funções externas à API da OpenAI.

03. Adição de Funções Externas à API da OpenAI

Bem-vindos a mais um capítulo da nossa apostila! Agora vamos explorar um tópico super interessante e essencial para quem quer criar agentes de IA poderosos e versáteis: a adição de funções externas à API da OpenAI. Vamos entender como essa capacidade pode transformar nossos modelos de linguagem em ferramentas ainda mais úteis e inteligentes.

Neste capítulo, nosso objetivo é aprender como adicionar funções externas à API da OpenAI. Vamos ver como isso permite que nossos modelos de linguagem realizem ações no mundo real e obtenham informações atualizadas, superando uma das principais limitações dos modelos de linguagem: o fato de serem treinados em dados antigos.

Por que adicionar ferramentas externas é relevante?

Os modelos de linguagem, como o GPT-3, são treinados em grandes quantidades de dados, mas esses dados têm um limite temporal. Por exemplo, o GPT-3 foi treinado com dados até 2023. Isso significa que ele não tem informações sobre eventos que ocorreram depois disso. Para criar aplicações realmente úteis e interativas, precisamos que nossos modelos possam agir no mundo real e obter informações atualizadas. É aí que entra a adição de funções externas.

A OpenAI foi pioneira em criar modelos que interpretam funções externas, permitindo que frameworks como o LangChain integrem essas capacidades. Vamos começar revisando como adicionar funções diretamente na API da OpenAI, para depois compararmos com a facilidade que o LangChain nos proporciona.

Importações Iniciais

Primeiro, vamos configurar nosso ambiente e importar as bibliotecas necessárias. Vamos utilizar a biblioteca da OpenAI e algumas outras para facilitar nosso trabalho.

```
import json
import openai
from dotenv import load_dotenv, find_dotenv

_ = load_dotenv(find_dotenv())

client = openai.Client()
```

Criando a Função que Será Adicionada ao Modelo

Para adicionar uma função ao modelo, precisamos primeiro definir essa função. Vamos criar uma função simples que retorna a temperatura atual de uma localidade. No nosso exemplo, a função é hardcoded (ou seja, é fictícia e retorna sempre os mesmos valores), mas em um cenário real, você poderia integrar uma API de clima.

```
def obter_temperatura_atual(local, unidade="celsius"):
    if "são paulo" in local.lower():
        return json.dumps(
            {"local": "São Paulo", "temperatura": "32", "unidade": unidade}
            )
    elif "porto alegre" in local.lower():
        return json.dumps(
            {"local": "Porto Alegre", "temperatura": "25", "unidade": unidade}
            )
    else:
        return json.dumps(
            {"local": local, "temperatura": "unknown"}
            )
```

Vamos testar nossa função para ver se ela está funcionando corretamente.

```
obter_temperatura_atual('Porto Alegre')
```

A saída será:

```
'{"local": "Porto Alegre", "temperatura": "25", "unidade": "celsius"}'
```

Criando a Descrição da Função

Agora que temos nossa função, precisamos criar um dicionário que descreva essa função para o modelo de linguagem. Esse dicionário deve ser bem detalhado para que o modelo entenda como e quando usar a função.

```
tools = [
    {
        "type": "function",
        "function": {
            "name": "obter_temperatura_atual",
            "description": "Obtém a temperatura atual em uma dada cidade",
            "parameters": {
                "type": "object",
                "properties": {
                    "local": {
                        "type": "string",
                        "description": "O nome da cidade. Ex: São Paulo",
                    },
                    "unidade": {
```

```
                    "type": "string",
                    "enum": ["celsius", "fahrenheit"]
                },
            },
            "required": ["local"],
        },
    },
}
]
```

Chamando o Modelo com a Nova Ferramenta

Vamos agora fazer uma pergunta ao modelo e ver como ele utiliza a função que adicionamos.

```
mensagens = [
    {'role': 'user', 'content': 'Qual é temperatura em Porto Alegre agora?'}
]

resposta = client.chat.completions.create(
    model='gpt-3.5-turbo-0125',
    messages=mensagens,
    tools=tools,
    tool_choice='auto'
)
resposta
```

A resposta será algo como:

```
ChatCompletion(id='chatcmpl-9S4Dz4a8Om4R33B0IGUIUzDI3Dtcb',
↪  choices=[Choice(finish_reason='tool_calls', index=0, logprobs=None,
↪  message=ChatCompletionMessage(content=None, role='assistant', function_call=None,
↪  tool_calls=[ChatCompletionMessageToolCall(id='call_yOuegMyCjHRpZXkYkWFMmrUL',
↪  function=Function(arguments='{"local":"Porto Alegre"}', name='obter_temperatura_atual'),
↪  type='function')]))], created=1716476451, model='gpt-3.5-turbo-0125',
↪  object='chat.completion', system_fingerprint=None,
↪  usage=CompletionUsage(completion_tokens=22, prompt_tokens=87, total_tokens=109))
```

Podemos perceber que o conteúdo da resposta veio vazio, pois para a pergunta “Qual é a temperatura em Porto Alegre?” ele necessitará chamar a função antes.

```
mensagem = resposta.choices[0].message
mensagem
```

Você pode perceber que a variável content da mensagem está vazia (None), mostrando que o modelo ainda não foi capaz de gerar uma resposta.

```
ChatCompletionMessage(content=None, role='assistant', function_call=None,
↪  tool_calls=[ChatCompletionMessageToolCall(id='call_yOuegMyCjHRpZXkYkWFMmrUL',
↪  function=Function(arguments='{"local":"Porto Alegre"}', name='obter_temperatura_atual'),
↪  type='function')])
```

Vamos extrair o nome da função e os argumentos que o modelo quer utilizar.

```
tools_call = mensagem.tool_calls[0]
print(tools_call.function.name)
print(tools_call.function.arguments)
```

A saída será:

```
obter_temperatura_atual
{"local":"Porto Alegre"}
```

Adicionando Resultado da Função às Mensagens

Vamos agora rodar a função e adicionar o resultado às mensagens.

```
observacao = obter_temperatura_atual(**json.loads(tools_call.function.arguments))
observacao
```

A saída será:

```
'{"local": "Porto Alegre", "temperatura": "25", "unidade": "celsius"}'
```

Vamos adicionar essa observação às mensagens e chamar o modelo novamente.

```
mensagens.append(mensagem)
mensagens.append({
    'tool_call_id': tools_call.id,
    'role': 'tool',
    'name': tools_call.function.name,
    'content': observacao
})
mensagens
```

A lista de mensagens ficará assim:

```
[{'role': 'user', 'content': 'Qual é temperatura em Porto Alegre agora?'},

 ChatCompletionMessage(content=None, role='assistant', function_call=None,
↪  tool_calls=[ChatCompletionMessageToolCall(id='call_yOuegMyCjHRpZXkYkWFMmrUL',
↪  function=Function(arguments='{"local":"Porto Alegre"}', name='obter_temperatura_atual'),
↪  type='function')]),

 {'tool_call_id': 'call_yOuegMyCjHRpZXkYkWFMmrUL',

  'role': 'tool',

  'name': 'obter_temperatura_atual',

  'content': '{"local": "Porto Alegre", "temperatura": "25", "unidade": "celsius"}'}]
```

Agora podemos chamar o modelo novamente.

```
resposta = client.chat.completions.create(
    model='gpt-3.5-turbo-0125',
```

```
    messages=mensagens,
    tools=tools,
    tool_choice='auto'
)
resposta
```

A resposta será:

```
ChatCompletion(id='chatcmpl-9S4IXDcfVOj869V1pS1fzXBToOrPy', choices=[Choice(fini
```

E finalmente, podemos extrair a resposta final.

```
resposta.choices[0].message.content
```

A saída será:

```
'A temperatura em Porto Alegre agora é de 25°C.'
```

E assim criamos um loop onde o modelo não havia capacidade por si só de dar uma resposta a pergunta. Mas ao ter acesso a uma ferramenta externa, ele pode solicitar a execução desta ferramenta com parâmetros definidos por e ele e a partir da observação gerada, devolver uma resposta correta ao usuário.

Explorando Diferentes Perguntas e o Parâmetro `tool_choice`

Por fim, vamos explorar como o parâmetro `tool_choice` afeta o comportamento do modelo. Esse parâmetro pode ser `auto`, `none` ou `um dicionário com a key function`.

Parâmetro “auto”

O parâmetro “auto” permite ao modelo definir automaticamente se é necessária a utilização de uma função ou não.

```
mensagens = [
    {'role': 'user', 'content': 'Qual é temperatura em Porto Alegre agora?'}
]
resposta = client.chat.completions.create(
    model='gpt-3.5-turbo-0125',
    messages=mensagens,
    tools=tools,
    tool_choice='auto'
)
mensagem = resposta.choices[0].message
print('Conteúdo:', mensagem.content)
print('Tools:', mensagem.tool_calls)
```

A saída será:

```
Conteúdo: None

Tools: [ChatCompletionMessageToolCall(id='call_PS1Nw8caOtDo9Q48ygEumxDa', functi
```

```
mensagens = [
    {'role': 'user', 'content': 'Olá'}
]
resposta = client.chat.completions.create(
    model='gpt-3.5-turbo-0125',
    messages=mensagens,
    tools=tools,
    tool_choice='auto'
)
mensagem = resposta.choices[0].message
print('Conteúdo:', mensagem.content)
print('Tools:', mensagem.tool_calls)
```

A saída será:

```
Conteúdo: Olá! Como posso ajudar você hoje?

Tools: None
```

Parâmetro “none”

Com o parâmetro “none”, o modelo não vai utilizar funções.

```
mensagens = [
    {'role': 'user', 'content': 'Qual a temperatura em Porto Alegre?'}
]
resposta = client.chat.completions.create(
    model='gpt-3.5-turbo-0125',
    messages=mensagens,
    tools=tools,
    tool_choice='none'
)
mensagem = resposta.choices[0].message
print('Conteúdo:', mensagem.content)
print('Tools:', mensagem.tool_calls)
```

A saída será:

```
Conteúdo: Por favor, aguarde um momento enquanto verifico a temperatura atual em Porto Alegre.

Tools: None
```

```
mensagens = [
    {'role': 'user', 'content': 'Olá'}
```

```
]
resposta = client.chat.completions.create(
    model='gpt-3.5-turbo-0125',
    messages=mensagens,
    tools=tools,
    tool_choice='none'
)
mensagem = resposta.choices[0].message
print('Conteúdo:', mensagem.content)
print('Tools:', mensagem.tool_calls)
```

A saída será:

```
Conteúdo: Olá! Como posso ajudar você hoje?

Tools: None
```

Parâmetro “dicionário com a key function”

Podemos fazer o modelo rodar obrigatoriamente a função, passando dentro de um dicionário a função que o modelo deve rodar.

```
mensagens = [
    {'role': 'user', 'content': 'Qual é temperatura em Porto Alegre agora?'}
]
resposta = client.chat.completions.create(
    model='gpt-3.5-turbo-0125',
    messages=mensagens,
    tools=tools,
    tool_choice={'function': 'obter_temperatura_atual'}
)
mensagem = resposta.choices[0].message
print('Conteúdo:', mensagem.content)
print('Tools:', mensagem.tool_calls)
```

A saída será:

```
Conteúdo: None

Tools: [ChatCompletionMessageToolCall(id='call_PS1Nw8caOtDo9Q48ygEumxDa',
↪  function=Function(arguments='{"local":"Porto Alegre"}', name='obter_temperatura_atual'),
↪  type='function')]
```

E é isso, pessoal! Agora vocês sabem como adicionar funções externas à API da OpenAI e como isso pode tornar seus modelos de linguagem muito mais poderosos e úteis.

04. Adição de Funções Externas Utilizando LangChain

Olá, pessoal! Bem-vindos de volta :)

Agora que entendemos como adicionar funções externas a modelos da OpenAI, vamos ver como o LangChain nos auxilia a tornar esse processo muito mais fácil e intuitivo. Nosso objetivos, portanto, serão:

- Aprender a usar a biblioteca Pydantic para validação de dados.
- Criar funções externas utilizando LangChain e Pydantic e integrá-las com a API da OpenAI.
- Utilizar o LangChain para adicionar essas funções aos nossos modelos de linguagem.

Introduzindo Pydantic

Como vimos no último capítulo, o trabalho de adicionar funções externas passa muito por sabermos descrever bem a utilidade das nossas funções e seus argumentos. Os modelos de linguagem compreendem apenas texto, então o entendimento de para que as funções que estamos oferecendo servem também deve ser feita por texto. Para que tenhamos funções e argumentos bem definidos, o LangChain passou a utilizar a biblioteca Pydantic, por ser esta a principal ferramenta de validação de estruturas de dados em Python. Agora vamos entender por que e como essa validação pode ser importante para nós.

Como Criamos uma Estrutura de Dados Sem Pydantic

Antes de entendermos como criar estruturas no Pydantic, vamos ver como criar uma estrutura de dados simples em Python sem usar essa biblioteca. Essa será nossa base de comparação e tornará mais evidente a importância de utilizarmos Pydantic:

```
class Pessoa:
    def __init__(self, nome: str, idade: int, peso: float) -> None:
        self.nome = nome
        self.idade = idade
        self.peso = peso

adriano = Pessoa('Adriano', 32, 68)
print(adriano.idade)  # Saída: 32
```

A forma básica em Python é utilizando classes como feito acima. Acabamos de criar uma estrutura para representar uma pessoa, e criamos uma instância dessa estrutura com argumentos referentes ao professor Adriano.

No entanto, sem validação de tipos, podemos acabar com dados inconsistentes:

```
adriano = Pessoa('Adriano', 32, 'ashdbadgvuya')
print(adriano.peso)  # Saída: 'ashdbadgvuya'
```

Neste caso, apesar de ser indicado que peso deve ser um float, ao adicionarmos uma string a estrutura de dados aceitará este sem problemas, o que gerará inconsistências.

Como Criamos uma Estrutura de Dados Usando Pydantic

Agora, vamos ver como o Pydantic pode nos ajudar a criar estruturas de dados mais robustas e com validação automática.

```
from pydantic import BaseModel

class pydPessoa(BaseModel):
    nome: str
    idade: int
    peso: float

adriano = pydPessoa(nome='Adriano', idade=32, peso=68)
print(adriano.nome)  # Saída: 'Adriano'
```

Podemos já ver que a estrutura é mais simples e mais robusta por automaticamente já realizar validação de dados:

```
try:
    adriano = pydPessoa(nome='Adriano', idade=32, peso='asuhdauishg')
except ValidationError as e:
    print(e)
```

Podemos ver que ao tentarmos iniciar uma instância com dados inconsistentes, o Pydantic gerará um erro avisando ao usuário.

Nesting de Classes com Pydantic

Podemos até fazer um “nesting” de classes, onde uma classe de dados recebe outra como input.

```
from typing import List

class pydAsimoTeam(BaseModel):
    funcionarios: List[pydPessoa]

team = pydAsimoTeam(funcionarios=[pydPessoa(nome='Adriano', idade=32, peso=68)])
print(team)
```

Utilizando Pydantic para Criação de Tools da OpenAI

Lembram que para criarmos uma função de temperatura atual que a API da OpenAI reconece precisávamos criar um dicionário cheio de argumentos, que facilmente poderia nos confundir:

```
import json

def obter_temperatura_atual(local, unidade="celsius"):
    if "são paulo" in local.lower():
        return json.dumps({"local": "São Paulo", "temperatura": "32", "unidade": unidade})
    elif "porto alegre" in local.lower():
        return json.dumps({"local": "Porto Alegre", "temperatura": "25", "unidade": unidade})
    else:
        return json.dumps({"local": local, "temperatura": "unknown"})

tools = [
    {
        "type": "function",
        "function": {
            "name": "obter_temperatura_atual",
            "description": "Obtém a temperatura atual em uma dada cidade",
            "parameters": {
                "type": "object",
                "properties": {
                    "local": {
                        "type": "string",
                        "description": "O nome da cidade. Ex: São Paulo",
                    },
                    "unidade": {
                        "type": "string",
                        "enum": ["celsius", "fahrenheit"]
                    },
                },
                "required": ["local"],
            },
        },
    }
]
```

Utilizando Pydantic para Criar a Tool

Vamos agora utilizar Pydantic para criar a mesma função, mas de uma forma mais robusta e com validação de dados.

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

class UnidadeEnum(str, Enum):
    celsius = 'celsius'
    fahrenheit = 'fahrenheit'
```

```
class ObterTemperaturaAtual(BaseModel):
    """Obtém a temperatura atual de uma determinada localidade"""
    local: str = Field(description='O nome da cidade', examples=['São Paulo', 'Porto Alegre'])
    unidade: Optional[UnidadeEnum]
```

Convertendo para uma Função da OpenAI

Agora vamos converter essa classe para uma função que a API da OpenAI consiga entender.

```
from langchain_core.utils.function_calling import convert_to_openai_function

tool_temperatura = convert_to_openai_function(ObterTemperaturaAtual)
print(tool_temperatura)
```

E obtemos o seguinte dicionário descrevendo a ferramenta:

```
{'name': 'ObterTemperaturaAtual',
 'description': 'Obtém a temperatura atual de uma determinada localidade',
 'parameters': {'type': 'object',
  'properties': {'local': {'description': 'O nome da cidade',
    'examples': ['São Paulo', 'Porto Alegre'],
    'type': 'string'},
   'unidade': {'description': 'An enumeration.',
    'enum': ['celsius', 'fahrenheit'],
    'type': 'string'}},
  'required': ['local']}}
```

Adicionando Função Externa Utilizando LangChain

Agora que já sabemos criar funções que os modelos de LLM entendam, podemos passar essas funções para os modelos de linguagem através da biblioteca LangChain.

Utilizando o Parâmetro `functions`

O parâmetro functions permite a adição de ferramentas externas aos chat_models do LangChain:

```
from langchain_openai import ChatOpenAI

chat = ChatOpenAI()

resposta = chat.invoke('Qual é a temperatura de Porto Alegre', functions=[tool_temperatura])
print(resposta)
```

Criando um Novo Componente de `chat_model`

Também é possível vincular essa ferramenta indefinidamente a um chat_model utilizando o método bind:

```
chat_com_func = chat.bind(functions=[tool_temperatura])
resposta = chat_com_func.invoke('Qual é a temperatura de Porto Alegre')
print(resposta)
```

Agora todas as próximas vezes que a instância chat_com_func for chamada, ela terá acesso a ferramenta.

Forçando o Modelo a Chamar uma Função

Da mesma forma que utilizando diretamente a biblioteca da OpenAI podemos forçar o modelo a chamar a função, podemos fazê-lo com LangChain da seguinte forma:

```
resposta = chat.invoke(
    'Qual é a temperatura de Porto Alegre',
    functions=[tool_temperatura],
    function_call={'name': 'ObterTemperaturaAtual'}
)
print(resposta)
```

Quando não passamos o parâmetro function_call, o modelo estará decidindo automaticamente a necessidade de rodar a função (estará utilizando o “auto”, como visto no capítulo anterior).

Adicionando a uma Chain

Podemos adicionar agora este modelo com funções a um prompt e criar uma chain.

```
from langchain.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_messages([
    ('system', 'Você é um assistente amigável chamado Isaac'),
    ('user', '{input}')
])

chain = prompt | chat.bind(functions=[tool_temperatura])

resposta = chain.invoke({'input': 'Olá'})
print(resposta)
```

E é isso, pessoal! Hoje aprendemos como adicionar funções externas utilizando o LangChain e a biblioteca Pydantic para validação de dados. O framework LangChain se mostrou novamente uma grande arma para facilitar nosso trabalho ao criarmos aplicações com modelos de linguagem. Vamos pro próximo capítulo, pois temos muito ainda a conhecer sobre nosso querido LangChain!

Desafio - Criando uma Função para Obter Emails

Após este conteúdo sobre adição de funções, quero propor um desafio para vocês! Em vez de criar uma função para obter a temperatura, que tal desenvolver uma função para obter emails? Essa é uma ótima oportunidade para vocês praticarem e consolidar suas habilidades! ### O que você precisa fazer?

Crie uma função chamada *obter_emails* que tenha os seguintes parâmetros:

- **tipo**: um argumento que define o tipo de emails a serem obtidos. Os valores possíveis são:
  - `'todos'`: para obter todos os emails.
  - `'não lidos'`: para obter apenas os emails que ainda não foram lidos.
  - `'lidos'`: para obter apenas os emails que já foram lidos.
- **quantidade**: um argumento que define quantos emails você deseja obter.

Essa função será o primeiro passo para criar um assistente pessoal que possa ler e responder aos nossos emails. Pense em como você pode estruturar a função e quais dados ela deve retornar. ### Lembre-se!

Nós vimos anteriormente como obter a temperatura atual utilizando LangChain e Pydantic. O processo envolveu a definição de uma classe de dados com Pydantic e a criação de uma função que poderia ser chamada pelo modelo. Aqui estão alguns trechos de código que utilizamos:

```
from pydantic import BaseModel, Field
from typing import Optional
from enum import Enum
from langchain_core.utils.function_calling import convert_to_openai_function

class UnidadeEnum(str, Enum):
    celsius = 'celsius'
    fahrenheit = 'fahrenheit'

class ObterTemperaturaAtual(BaseModel):
    """Obtém a temperatura atual de uma determinada localidade"""
    local: str = Field(description='O nome da cidade', examples=['São Paulo', 'Porto Alegre'])
    unidade: Optional[UnidadeEnum]

tool_temperatura = convert_to_openai_function(ObterTemperaturaAtual)
tool_temperatura
```

Agora, aplique esse conhecimento para criar sua função de obter emails!

05. Tagging - Categorização de Texto Utilizando Funções

Bem-vindos de volta a nossa apostila! No último capítulo, aprendemos como adicionar funções aos modelos utilizando o framework LangChain. Hoje, vamos explorar uma aplicação prática desse conhecimento: a categorização de texto, ou "tagging". Nosso objetivo será:

- Entender o conceito de tagging e sua importância.
- Aprender a criar uma função para análise de sentimento e detecção de língua.
- Implementar um modelo para direcionamento de mensagens para setores específicos.
- Melhorar a precisão do modelo utilizando técnicas de engenharia de prompt.

Por que é importante categorizar textos?

A maior parte da informação disponível hoje está na internet no formato de texto, dentro de websites. Isso pode ser um desafio para analistas de dados, pois esses dados não estão estruturados, ou seja, se apresentam das mais diferentes formas que você possa imaginar. Sorte a nossa que temos modelos de linguagem, pois eles tem uma capacidade incrível para processar esses dados e estruturá-los, possibilitando uma análise em larga escala e de forma eficiente. Tagging é um conceito que auxilia nessa estruturação de dados, transformando dados não estruturados em dados estruturados.

Tagging para Análise de Sentimento e Detecção de Língua

Vamos começar criando uma função que vai extrair o sentimento de uma mensagem (positivo, negativo ou indefinido) e também identificar a língua do texto.

```
from pydantic import BaseModel, Field
from langchain_core.utils.function_calling import convert_to_openai_function

class Sentimento(BaseModel):
    '''Define o sentimento e a língua da mensagem enviada'''
    sentimento: str = Field(description='Sentimento do texto. Deve ser "pos", "neg" ou "nd"
↪  para não definido.')
    lingua: str = Field(description='Língua que o texto foi escrito (deve estar no formato ISO
↪  639-1)')

tool_sentimento = convert_to_openai_function(Sentimento)
tool_sentimento
```

```
{'name': 'Sentimento',
 'description': 'Define o sentimento e a língua da mensagem enviada',
 'parameters': {'type': 'object',
  'properties': {'sentimento': {'description': 'Sentimento do texto. Deve ser "pos", "neg" ou
  ↪  "nd" para não definido.',
    'type': 'string'},
```

```
    'lingua': {'description': 'Língua que o texto foi escrito (deve estar no formato ISO
    ↪  639-1)',
     'type': 'string'}},
   'required': ['sentimento', 'lingua']}}
```

Vamos testar com um texto de exemplo:

```
texto = 'Eu gosto muito de massa aos quatro queijos'
```

Agora, vamos configurar o prompt e o modelo de chat:

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

prompt = ChatPromptTemplate.from_messages([
    ('system', 'Pense com cuidado ao categorizar o texto conforme as instruções'),
    ('user', '{input}')
])
chat = ChatOpenAI()

chain = prompt | chat.bind(functions=[tool_sentimento], function_call={'name': 'Sentimento'})
```

Vamos invocar a chain com o nosso texto:

```
chain.invoke({'input': 'Eu gosto muito de massa aos quatro queijos'})
```

```
AIMessage(content='', additional_kwargs={'function_call': {'arguments':
↪  '{"sentimento":"pos","lingua":"pt"}', 'name': 'Sentimento'}},
↪  response_metadata={'token_usage': {'completion_tokens': 11, 'prompt_tokens': 145,
↪  'total_tokens': 156}, 'model_name': 'gpt-3.5-turbo', 'system_fingerprint': None,
↪  'finish_reason': 'stop', 'logprobs': None},
↪  id='run-e68e3dfc-105c-4cc9-9a2a-f08e9c359f67-0')
```

Vamos testar com outro texto:

```
chain.invoke({'input': 'I dont like this food'})
```

```
AIMessage(content='', additional_kwargs={'function_call': {'arguments':
↪  '{"sentimento":"neg","lingua":"en"}', 'name': 'Sentimento'}},
↪  response_metadata={'token_usage': {'completion_tokens': 11, 'prompt_tokens': 138,
↪  'total_tokens': 149}, 'model_name': 'gpt-3.5-turbo', 'system_fingerprint': None,
↪  'finish_reason': 'stop', 'logprobs': None},
↪  id='run-101f6bd8-b4b9-4e49-b076-f18619b06d14-0')
```

Parseando a Saída para Obtermos Apenas o que Interessa

Podemos utilizar o `JsonOutputFunctionsParser` para obtermos como resultado final da nossa chain apenas o que nos interessa, que é o dicionário com as tags do conteúdo.

```
from langchain.output_parsers.openai_functions import JsonOutputFunctionsParser

chain = (prompt
         | chat.bind(functions=[tool_sentimento], function_call={'name': 'Sentimento'})
```

```
        | JsonOutputFunctionsParser())
chain.invoke({'input': 'Eu gosto muito de massa aos quatro queijos'})
```

```
{'sentimento': 'pos', 'lingua': 'pt'}
```

```
chain.invoke({'input': 'I dont like this food'})
```

```
{'sentimento': 'neg', 'lingua': 'en'}
```

Um Exemplo Mais Interessante

Vamos agora para um exemplo um pouco mais complexo. Digamos que temos um chatbot e queremos fazer um roteamento para os setores de interesse, no nosso caso: atendimento_cliente, duvidas_alunos, vendas, spam. A primeira etapa é criar um modelo que entenda a solicitação do cliente e direcione para o setor certo.

Retiramos as seguintes mensagens de email recebidos pela nossa equipe de atendimento e vamos tentar criar um direcionamento para elas:

```
duvidas = [
    'Bom dia, gostaria de saber se há um certificado final para cada trilha ou se os
    ↪  certificados são somente para os cursos e projetos? Obrigado!',
    'In Etsy, Amazon, eBay, Shopify https://pint77.com Pinterest+SEO +II = high sales
    ↪  results',
    'Boa tarde, estou iniciando hoje e estou perdido. Tenho vários objetivos. Não sei nada
    ↪  programação, exceto que utilizo o Power automate desktop da Microsoft. Quero aprender
    ↪  tudo na plataforma que se relacione ao Trading de criptomoedas. Quero automatizar
    ↪  Tradings, fazer o sistema reconhecer padrões, comprar e vender segundo critérios que
    ↪  eu defina, etc. Também tenho objetivos de aprender o máximo para utilizar em
    ↪  automações no trabalho também, que envolve a área jurídica e trabalho em processos.
    ↪  Como sou fã de eletrônica e tenho cursos na área, também queria aprender o que precisa
    ↪  para automatizacões diversas. Existe algum curso ou trilha que me prepare com base
    ↪  para todas essas áreas ao mesmo tempo e a partir dele eu aprenda isoladamente aquilo
    ↪  que seria exigido para aplicar aos meus projetos?',
    'Bom dia, Havia pedido cancelamento de minha mensalidade no mes 2 e continuaram cobrando.
    ↪  Peço cancelamento da assinatura. Peço por gentileza, para efetivarem o cancelamento da
    ↪  assomatura e pagamento.',
    'Bom dia. Não estou conseguindo tirar os certificados dos cursos que concluí. Por exemplo,
    ↪  já consegui 100% no python starter, porém, não consigo tirar o certificado. Como
    ↪  faço?',
    'Bom dia. Não enconte no site o preço de um curso avulso. SAberiam me informar?'
]
```

Vamos criar um modelo que entenda a solicitação do cliente e direcione para o setor certo:

```
from enum import Enum
from pydantic import BaseModel, Field

class SetorEnum(str, Enum):
    atendimento_cliente = 'atendimento_cliente'
    duvidas_aluno = 'duvidas_aluno'
```

```
    vendas = 'vendas'
    spam = 'spam'

class DirecionaSetorResponsavel(BaseModel):
    """Direciona a dúvida de um cliente ou aluno da escola de programação Asimov para o setor
    ↪  responsável"""
    setor: SetorEnum
```

Vamos converter essa função para o formato da OpenAI:

```
from langchain_core.utils.function_calling import convert_to_openai_function

tool_direcionamento = convert_to_openai_function(DirecionaSetorResponsavel)
tool_direcionamento
```

```
{'name': 'DirecionaSetorResponsavel',
 'description': 'Direciona a dúvida de um cliente ou aluno da escola de programação Asimov
 ↪  para o setor responsável',
 'parameters': {'type': 'object',
  'properties': {'setor': {'description': 'An enumeration.',
    'enum': ['atendimento_cliente', 'duvidas_aluno', 'vendas', 'spam'],
    'type': 'string'}},
  'required': ['setor']}}
```

Vamos configurar o prompt e o modelo de chat:

```
from langchain.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain.output_parsers.openai_functions import JsonOutputFunctionsParser

prompt = ChatPromptTemplate.from_messages([
    ('system', 'Pense com cuidado ao categorizar o texto conforme as instruções'),
    ('user', '{input}')
])
chat = ChatOpenAI()

chain = (prompt
         | chat.bind(functions=[tool_direcionamento], function_call={'name':
         ↪  'DirecionaSetorResponsavel'})
         | JsonOutputFunctionsParser())
```

Vamos testar com uma das dúvidas:

```
duvida = duvidas[5]
resposta = chain.invoke({'input': duvida})
print('Dúvida:', duvida)
print('Resposta:', resposta)
```

```
Dúvida: Bom dia. Não enconte no site o preço de um curso avulso. SAberiam me informar?

Resposta: {'setor': 'atendimento_cliente'}
```

Neste caso, gostaríamos que a dúvida fosse direcionada para vendas. Podemos melhorar nosso prompt para aumentar a chance do modelo responder como gostaríamos:

```
from langchain.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain.output_parsers.openai_functions import JsonOutputFunctionsParser

system_message = '''Pense com cuidado ao categorizar o texto conforme as instruções.
Questões relacionadas a dúvidas de preço, sobre o produto, como funciona devem ser
↪  direcionadas para "vendas".
Questões relacionadas a conta, acesso a plataforma, a cancelamento e renovação de assinatura
↪  devem ser direcionadas para "atendimento_cliente".
Questões relacionadas a dúvidas técnicas de programação, conteúdos da plataforma ou
↪  tecnologias na área da programação devem ser direcionadas para "duvidas_alunos".
Mensagens suspeitas, em outras línguas que não português, contendo links devem ser
↪  direcionadas para "spam".
'''

prompt = ChatPromptTemplate.from_messages([
    ('system', system_message),
    ('user', '{input}')
])
chat = ChatOpenAI()

chain = (prompt
         | chat.bind(functions=[tool_direcionamento], function_call={'name':
         ↪  'DirecionaSetorResponsavel'})
         | JsonOutputFunctionsParser())
```

Vamos testar novamente com a mesma dúvida:

```
duvida = duvidas[5]
resposta = chain.invoke({'input': duvida})
print('Dúvida:', duvida)
print('Resposta:', resposta)
```

```
Dúvida: Bom dia. Não enconte no site o preço de um curso avulso. SAberiam me informar?

Resposta: {'setor': 'vendas'}
```

Vocês podem ver que é um conjunto de tarefas. Tem, claro, a definição da nossa função, que temos que fazer bem, mas também tem muito de prompt, de engenharia de prompt, de forma que fique bem claro para o modelo como ele deve categorizar. Quanto mais informação dermos, melhor será a categorização que ele fará.

Dica de conteúdo

Na nossa aula sobre tagging, discutimos como a categorização de dados (*tagging*) pode ser uma ferramenta poderosa para interpretar e estruturar informações em texto.

Um ótimo exemplo prático dessa aplicação esta no vídeo do Rodrigo, que foi postado no YouTube recentemente. Nele, o Rodrigo mostra como utilizar tagging para categorizar suas compras do cartão de crédito, criando assim facilmente uma ferramenta de análise para finanças pessoais.

É uma aplicação prática exatamente dos conhecimentos que abordamos nesta aula, e sugiro que vocês deem uma conferida!

Graças ao tagging, podemos desenvolver aplicações que entendem o contexto dos dados e os organizam de maneira que façam sentido.

Além disso, existem diversas atividades de categorização que podemos realizar com os modelos de linguagem, assim como o Rodrigo fez. Esses modelos são grandes **ferramentas para estruturar dados não estruturados**, transformando uma vasta gama de informações que, de outra forma, seriam difíceis de processar, em dados acessíveis e prontos para análise.

Essa capacidade de categorizar e interpretar dados é **uma verdadeira revolução na área de análise de dados**. Com o uso de *tagging*, podemos trabalhar cada vez mais com **dados em texto**, o que abre um leque de possibilidades para aplicações em diferentes setores. Seja na análise de sentimentos, no direcionamento de dúvidas em um chatbot ou na organização de feedbacks de clientes, o tagging se mostra uma ferramenta poderosa que pode mudar a forma como lidamos com informações.

Portanto, não deixem de conferir o vídeo do Rodrigo e se inspirar nas diversas maneiras de aplicar o *tagging* em suas próprias soluções.

06. Extraction - Extraindo e Estruturando Informações de Textos

Olá, pessoal! Bem-vindos a mais um capítulo do nosso curso de Agents de IA com Python e LangChain. Hoje vamos explorar mais uma aplicação muito interessante e prática ao utilizarmos funções com modelos de linguagem: a extração de informações de textos. Essa técnica é extremamente útil quando precisamos lidar com grandes volumes de dados textuais e queremos extrair informações específicas de forma estruturada. Nosso objetivo hoje será:

- Entender o conceito de extração de informações de textos.
- Aprender a utilizar funções externas para extrair dados específicos.
- Implementar um exemplo prático de extração de datas e acontecimentos de um texto.
- Explorar a aplicação de extração de informações da web utilizando WebScraping.

Vamos começar com um exemplo simples, extraindo datas e acontecimentos de um texto.

Extraindo de Dados de Textos

Vamos supor que temos o seguinte texto e queremos extrair as datas e os acontecimentos mencionados. Aqui temos um texto que utilizarmos como exemplo:

```
texto = '''A Apple foi fundada em 1 de abril de 1976 por Steve Wozniak, Steve Jobs e Ronald
↪  Wayne com o nome de Apple Computers, na Califórnia. O nome foi escolhido por Jobs após a
↪  visita do pomar de maçãs da fazenda de Robert Friedland, também pelo fato do nome soar bem
↪  e ficar antes da Atari
nas listas telefônicas.

O primeiro protótipo da empresa foi o Apple I que foi demonstrado na Homebrew Computer Club em
↪  1975, as vendas começaram em julho de 1976 com o preço de US$ 666,66, aproximadamente 200
↪  unidades foram vendidas,[21] em 1977 a empresa conseguiu o aporte de Mike Markkula e um
↪  empréstimo do Bank of America.'''
```

Para extrair essas informações, vamos criar uma estrutura de dados que represente um acontecimento com uma data e uma descrição:

```
from pydantic import BaseModel, Field
from typing import List
from langchain_core.utils.function_calling import convert_to_openai_function

class Acontecimento(BaseModel):
    '''Informação sobre um acontecimento'''
    data: str = Field(description='Data do acontecimento no formato YYYY-MM-DD')
    acontecimento: str = Field(description='Acontecimento extraído do texto')

class ListaAcontecimentos(BaseModel):
    """Acontecimentos para extração"""
    acontecimentos: List[Acontecimento] = Field(description='Lista de acontecimentos presentes
↪  no texto informado')
```

```
tool_acontecimentos = convert_to_openai_function(ListaAcontecimentos)
tool_acontecimentos
```

Agora, vamos criar uma chain para extrair essas informações do texto:

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

prompt = ChatPromptTemplate.from_messages([
    ('system', 'Extraia as frases de acontecimentos. Elas devem ser extraídas integralmente'),
    ('user', '{input}')
])
chat = ChatOpenAI()

chain = (prompt
         | chat.bind(functions=[tool_acontecimentos], function_call={'name':
         ↪  'ListaAcontecimentos'}))
```

Vamos invocar a chain com o nosso texto:

```
chain.invoke({'input': texto})
```

O resultado será algo assim:

```
AIMessage(content='', additional_kwargs={'function_call': {'arguments':
↪  '{"acontecimentos":[{"data":"1976-04-01","acontecimento":"A Apple foi fundada por Steve
↪  Wozniak, Steve Jobs e Ronald Wayne."},{"data":"1975-07","acontecimento":"Início das vendas
↪  do Apple I com o preço de US$ 666,66."},{"data":"1977","acontecimento":"A Apple recebeu
↪  aporte de Mike Markkula e um empréstimo do Bank of America."}]}', 'name':
↪  'ListaAcontecimentos'}}, response_metadata={'token_usage': {'completion_tokens': 101,
↪  'prompt_tokens': 325, 'total_tokens': 426}, 'model_name': 'gpt-3.5-turbo',
↪  'system_fingerprint': None, 'finish_reason': 'stop', 'logprobs': None},
↪  id='run-ad3b401a-654f-4d2a-b120-630b0d964429-0')
```

Para tornar a saída mais limpa, podemos utilizar um OutputParser:

```
from langchain.output_parsers.openai_functions import JsonOutputFunctionsParser

chain = (prompt
         | chat.bind(functions=[tool_acontecimentos], function_call={'name':
         ↪  'ListaAcontecimentos'})
         | JsonOutputFunctionsParser())

chain.invoke({'input': texto})
```

O resultado será:

```
{'acontecimentos': [{'data': '1976-04-01',
   'acontecimento': 'A Apple foi fundada em 1 de abril de 1976 por Steve Wozniak, Steve Jobs e
   ↪  Ronald Wayne com o nome de Apple Computers, na Califórnia.'},
  {'data': '1975-07',
   'acontecimento': 'As vendas do Apple I começaram em julho de 1976 com o preço de US$
   ↪  666,66, aproximadamente 200 unidades foram vendidas.'},
```

```
  {'data': '1977',
   'acontecimento': 'Em 1977 a empresa conseguiu o aporte de Mike Markkula e um empréstimo do
   ↪  Bank of America.'}]}
```

Se quisermos uma saída ainda mais limpa, podemos especificar a chave de saída ao utilizarmos o JsonKeyOutputFunctionsParser:

```
from langchain.output_parsers.openai_functions import JsonKeyOutputFunctionsParser

chain = (prompt
         | chat.bind(functions=[tool_acontecimentos], function_call={'name':
         ↪  'ListaAcontecimentos'})
         | JsonKeyOutputFunctionsParser(key_name='acontecimentos'))

chain.invoke({'input': texto})
```

O resultado final será:

```
[{'data': '1976-04-01',

  'acontecimento': 'Apple foi fundada por Steve Wozniak, Steve Jobs e Ronald Wayne.'},

 {'data': '1975',

  'acontecimento': 'Demonstração do primeiro protótipo da empresa, o Apple I, na Homebrew
  ↪  Computer Club.'},

 {'data': '1976-07',

  'acontecimento': 'Início das vendas do Apple I com o preço de US$ 666,66.'},

 {'data': '1977',

  'acontecimento': 'Empresa recebeu aporte de Mike Markkula e empréstimo do Bank of
  ↪  America.'}]
```

Extraindo Informações da Web

A extração de informações também pode ser muito útil quando combinada com técnicas de WebScraping. Vamos ver um exemplo de como extrair informações de uma página web.

Vamos utilizar a página de blog da Asimov Academy como exemplo:

Primeiro, vamos carregar a página utilizando o document_loader do LangChain:

```
from langchain_community.document_loaders.web_base import WebBaseLoader

loader = WebBaseLoader('https://hub.asimov.academy/blog/')
page = loader.load()
page
```

O conteúdo da página estará desformatado, então precisamos definir uma estrutura para os posts do blog:

```
from pydantic import BaseModel, Field
from typing import List
from langchain_core.utils.function_calling import convert_to_openai_function

class BlogPost(BaseModel):
    '''Informações sobre um post de blog'''
    titulo: str = Field(description='O título do post de blog')
    autor: str = Field(description='O autor do post de blog')
    data: str = Field(description='A data de publicação do post de blog')

class BlogSite(BaseModel):
    '''Lista de blog posts de um site'''
    posts: List[BlogPost] = Field(description='Lista de posts de blog do site')

tool_blog = convert_to_openai_function(BlogSite)
tool_blog
```

Agora, vamos criar uma chain para extrair essas informações:

```
from langchain.output_parsers.openai_functions import JsonKeyOutputFunctionsParser
from langchain.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI

prompt = ChatPromptTemplate.from_messages([
    ('system', 'Extraia da página todos os posts de blog com autor e data de publicação'),
    ('user', '{input}')
])
chat = ChatOpenAI()
chain = (prompt
         | chat.bind(functions=[tool_blog], function_call={'name': 'BlogSite'})
         | JsonKeyOutputFunctionsParser(key_name='posts'))
```

Vamos invocar a chain com o conteúdo da página:

```
chain.invoke({'input': page})
```

O resultado será:

```
[{'titulo': 'Python ou Power BI: qual ferramenta é melhor para criar dashboards?',
  'autor': 'Renata Lopes',
  'data': '23-05-24'},
 {'titulo': 'Try e Except em Python - Entenda como lidar com erros',
  'autor': 'Adriano Soares',
  'data': '22-05-24'},
 {'titulo': 'Print em Python: entenda o que é a função print e onde usá-la',
  'autor': 'Juliano Faccioni',
  'data': '21-05-24'},
 {'titulo': 'Exemplos de programas em Python - Aprenda fazendo',
  'autor': 'Adriano Soares',
  'data': '16-05-24'},
 {'titulo': 'Qual linguagem de programação aprender? Guia completo para iniciantes',
```

```
  'autor': 'Renata Lopes',
  'data': '15-05-24'},
```

E com extrema facilidade conseguimos estruturar informações capturadas de páginas web combinando Web Scraping com modelos de linguagem.

E é isso, pessoal! Vimos como é possível extrair informações específicas de textos e páginas web utilizando Python e LangChain. Essas técnicas são extremamente úteis para transformar dados não estruturados em informações organizadas e utilizáveis.

Desafio

Após este conteúdo sobre extração de informações utilizando LLMs, quero propor um desafio para vocês! É um exercício simples e bem pedagógico, mas a ideia é reforçar o aprendizado de vocês e mostrar como as LLMs podem ser utilizadas até nas atividades mais simples do cotidiano.

Vamos aplicar o que aprendemos e extrair dados da seguinte receita que encontrei na internet. Após a extração, vamos criar uma lista de compras. Pensem que este poderia ser muito facilmente o primeiro passo para um agente que faz compras automaticamente ao fornecermos uma receita para ele. A receita que temos é a seguinte:

Receita de Bolo de Cenoura

Massa 1. Em um liquidificador, adicione a cenoura, os ovos e o óleo, depois misture. 2. Acrescente o açúcar e bata novamente por 5 minutos. 3. Em uma tigela ou na batedeira, adicione a farinha de trigo e depois misture novamente. 4. Acrescente o fermento e misture lentamente com uma colher. 5. Asse em um forno preaquecido a 180° C por aproximadamente 40 minutos. Cobertura 6. Despeje em uma tigela a manteiga, o chocolate em pó, o açúcar e o leite, depois misture. 7. Leve a mistura ao fogo e continue misturando até obter uma consistência cremosa, depois despeje a calda por cima do bolo.

O que você precisa fazer?

A partir da receita acima, você deve extrair as seguintes informações:

- **Utensílios de Cozinha**: Liste todos os utensílios que são mencionados na receita (por exemplo, liquidificador, tigela, batedeira).
- **Ingredientes**: Liste todos os ingredientes necessários para preparar o bolo (por exemplo, cenoura, ovos, óleo, açúcar, farinha de trigo, fermento, manteiga, chocolate em pó, leite).
- **Salvar em CSV**: Após extrair as informações, salve os dados de ingredientes em um arquivo CSV e os utensílios em outro.

Dicas para a Extração

- Utilize a estrutura que aprendemos na aula para criar as classes de dados com Pydantic, se necessário.
- Pense em como você pode estruturar seu código para que a extração seja eficiente e organizada.
- Lembre-se de que a categorização e a estruturação de dados são fundamentais para facilitar a análise e utilização posterior do seu agente.

07. Criação de Tools com LangChain

Olá, pessoal! Bem-vindos a mais um capítulo da nossa apostila. Hoje, vamos nos aprofundar em um dos aspectos mais essenciais para a construção de agentes inteligentes: as Tools (ferramentas). As ferramentas são o que realmente diferenciam um Agent, permitindo que ele interaja com o mundo de maneira mais eficiente e eficaz. Nosso objetivo agora será:

- Entender o que são Tools e sua importância para os Agents.
- Aprender a criar Tools utilizando o decorador `@tool`.
- Explorar a criação de Tools com a metaclasse `StructuredTool`.
- Compreender como descrever argumentos de forma clara e precisa.
- Ver exemplos práticos de como chamar e utilizar essas Tools.

O que são tools?

As tools (ferramentas) são componentes essenciais no ecossistema do LangChain, permitindo que agentes, cadeias ou modelos de linguagem (LLMs) interajam com o mundo externo. Elas funcionam como interfaces que possibilitam a execução de ações específicas, facilitando a integração de funcionalidades externas nas aplicações de IA.

Como as Tools se Relacionam com Modelos de Linguagem

As tools são especialmente poderosas quando combinadas com as capacidades de tool calling dos modelos de linguagem. Isso significa que, ao invés de apenas gerar texto, os LLMs podem agora chamar funções externas para realizar ações específicas, como buscar informações, processar dados ou interagir com APIs.

Por exemplo, ao criar uma tool que retorna a temperatura atual de uma localidade, o modelo de linguagem pode ser instruído a chamar essa função quando um usuário faz uma pergunta sobre o clima. Isso não só enriquece a interação, mas também permite que o modelo forneça respostas mais precisas e contextuais.

Criando Tools com o Decorador `@tool`

Vamos começar criando uma ferramenta utilizando o decorador `@tool`. O decorador `@tool` modifica uma função, permitindo que ela seja usada como uma ferramenta pelo LangChain.

```
from langchain.agents import tool
```

```
@tool
def retorna_temperatura_atual(localidade: str):
    '''Faz busca online de temperatura de uma localidade'''
    return '25°C'

retorna_temperatura_atual
```

Quando executamos o código acima, obtemos uma `StructuredTool` com todas as informações necessárias:

```
StructuredTool(name='retorna_temperatura_atual',
↪  description='retorna_temperatura_atual(localidade: str) - Faz busca online de temperatura
↪  de uma localidade', args_schema=<class
↪  'pydantic.v1.main.retorna_temperatura_atualSchema'>, func=<function
↪  retorna_temperatura_atual at 0x000001EE78100D60>)
```

Podemos acessar o nome, a descrição e os argumentos da ferramenta:

```
retorna_temperatura_atual.name
# 'retorna_temperatura_atual'

retorna_temperatura_atual.description
# 'retorna_temperatura_atual(localidade: str) - Faz busca online de temperatura de uma
↪  localidade'

retorna_temperatura_atual.args
# {'localidade': {'title': 'Localidade', 'type': 'string'}}
```

Descrevendo os Argumentos

Para garantir que a ferramenta seja utilizada corretamente, é importante descrever melhor os argumentos. Vamos usar a biblioteca `pydantic` para isso.

```
from langchain.agents import tool
from pydantic import BaseModel, Field

class RetornaTempArgs(BaseModel):
    localidade: str = Field(description='Localidade a ser buscada', examples=['São Paulo',
↪  'Porto Alegre'])

@tool(args_schema=RetornaTempArgs)
def retorna_temperatura_atual(localidade: str):
    '''Faz busca online de temperatura de uma localidade'''
    return '25°C'

retorna_temperatura_atual
```

Agora, a descrição dos argumentos é mais detalhada, o que aumenta as chances do modelo entender como utilizar bem a ferramenta.

```
retorna_temperatura_atual.args
# {'localidade': {'title': 'Localidade', 'description': 'Localidade a ser buscada',
↪  'examples': ['São Paulo', 'Porto Alegre'], 'type': 'string'}}
```

Chamando a Tool

Depois de passar o decorador, a ferramenta ganha o método `invoke`, que permite chamá-la com os argumentos necessários.

```
retorna_temperatura_atual.invoke({'localidade': 'Porto Alegre'})
# '25°C'
```

Criando Tools com `StructuredTool`

Outra forma de criar uma ferramenta é utilizando a metaclasse `StructuredTool` do LangChain. As funcionalidades são bem similares ao uso do decorador.

```
from langchain.tools import StructuredTool
from pydantic import BaseModel, Field

class RetornaTempArgs(BaseModel):
    localidade: str = Field(description='Localidade a ser buscada', examples=['São Paulo',
↪  'Porto Alegre'])

def retorna_temperatura_atual(localidade: str):
    return '25°C'

tool_temp = StructuredTool.from_function(
    func=retorna_temperatura_atual,
    name='ToolTemperatura',
    args_schema=RetornaTempArgs,
    description='Faz busca online de temperatura de uma localidade',
    return_direct=True
)

tool_temp
```

A `StructuredTool` criada possui todas as informações necessárias:

```
StructuredTool(name='ToolTemperatura', description='ToolTemperatura(localidade: str) - Faz
↪  busca online de temperatura de uma localidade', args_schema=<class
↪  '__main__.RetornaTempArgs'>, return_direct=True, func=<function retorna_temperatura_atual
↪  at 0x000001EE79A29440>)
```

Podemos acessar o nome, a descrição e os argumentos da ferramenta:

```
tool_temp.name
# 'ToolTemperatura'

tool_temp.args
```

```
# {'localidade': {'title': 'Localidade', 'description': 'Localidade a ser buscada',
↪  'examples': ['São Paulo', 'Porto Alegre'], 'type': 'string'}}

tool_temp.description
# 'ToolTemperatura(localidade: str) - Faz busca online de temperatura de uma localidade'
```

E também podemos chamar a ferramenta:

```
tool_temp.invoke({'localidade': 'Porto Alegre'})
# '25°C'
```

E é isso aí, pessoal! Agora vocês sabem como criar Tools utilizando o LangChain. Na próxima aula, vamos criar ferramentas reais que buscam informações através de APIs, como a temperatura de uma localidade e buscas no Wikipedia. As coisas estão começando a ficar cada vez mais reais, e logo chegaremos aos Agents. Até o próximo capítulo!

08. Processo Completo para Criação de Tool de Temperatura e do Wikipedia

Olá, pessoal! Bem-vindos a mais um capítulo do nosso curso. Hoje, vamos colocar a mão na massa e criar duas ferramentas (ou "tools") reais e funcionais: uma para buscar a temperatura atual de uma localidade específica e outra para fazer buscas no Wikipedia. Essas ferramentas são ótimos exemplos de como podemos integrar APIs externas e bibliotecas em nossos projetos de IA. Nossos objetivos, portanto, serão:

- Entender como criar uma tool de busca de temperatura utilizando a API da OpenMeteo.
- Aprender a criar uma tool de busca no Wikipedia utilizando a biblioteca `wikipedia`.
- Testar as tools para garantir que estão funcionando corretamente.
- Integrar essas tools em uma chain utilizando LangChain.

No mundo da programação, especialmente quando estamos lidando com agentes de IA, é essencial saber como integrar diferentes APIs e bibliotecas para ampliar as capacidades dos nossos modelos. Hoje, vamos ver como fazer isso de forma prática e direta, criando ferramentas que podem ser usadas em diversos contextos.

Criando uma Tool de Busca de Temperatura

Vamos começar com a criação de uma ferramenta que busca a temperatura atual de uma localidade específica utilizando a API da OpenMeteo.

Passo a Passo para Criar a Função de Temperatura

Primeiro, vamos criar uma função que acessa a API da OpenMeteo e retorna a temperatura de um local. Aqui está o código básico:

```
import requests
import datetime

# URL da API
URL = 'https://api.open-meteo.com/v1/forecast'

# Parâmetros da API
params = {
    'latitude': -30,  # Latitude de Porto Alegre
    'longitude': -50,  # Longitude de Porto Alegre
    'hourly': 'temperature_2m',
    'forecast_days': 1,
}
```

```
# Fazendo a requisição para a API
resposta = requests.get(URL, params=params)
if resposta.status_code == 200:
    resultado = resposta.json()

    # Obtendo a hora atual em UTC
    hora_agora = datetime.datetime.now(datetime.UTC).replace(tzinfo=None)

    # Convertendo a lista de horas para o formato datetime
    lista_horas = [datetime.datetime.fromisoformat(temp_str) for temp_str in
↪   resultado['hourly']['time']]

    # Encontrando o índice da hora mais próxima
    index_mais_prox = min(range(len(lista_horas)), key=lambda x: abs(lista_horas[x] -
↪   hora_agora))

    # Obtendo a temperatura atual
    temp_atual = resultado['hourly']['temperature_2m'][index_mais_prox]
    print(temp_atual)
else:
    raise Exception(f'Request para API {URL} falhou: {resposta.status_code}')
```

Encapsulando em uma Função

Agora que temos o código básico funcionando, vamos encapsulá-lo em uma função para facilitar o uso e utilizar o decorator @tool para gerar uma ferramenta, adicionando um ArgsSchema para definir os argumentos:

```
from langchain.agents import tool
from pydantic import BaseModel, Field

class RetornTempArgs(BaseModel):
    latitude: float = Field(description='Latitude da localidade que buscamos a temperatura')
    longitude: float = Field(description='Longitude da localidade que buscamos a temperatura')

@tool(args_schema=RetornTempArgs)
def retorna_temperatura_atual(latitude: float, longitude: float):
    '''Retorna a temperatura atual para uma dada coordenada'''

    URL = 'https://api.open-meteo.com/v1/forecast'

    params = {
        'latitude': latitude,
        'longitude': longitude,
        'hourly': 'temperature_2m',
        'forecast_days': 1,
    }

    resposta = requests.get(URL, params=params)
    if resposta.status_code == 200:
```

```
        resultado = resposta.json()

        hora_agora = datetime.datetime.now(datetime.UTC).replace(tzinfo=None)
        lista_horas = [datetime.datetime.fromisoformat(temp_str) for temp_str in
↪  resultado['hourly']['time']]
        index_mais_prox = min(range(len(lista_horas)), key=lambda x: abs(lista_horas[x] -
↪  hora_agora))

        temp_atual = resultado['hourly']['temperature_2m'][index_mais_prox]
        return temp_atual
    else:
        raise Exception(f'Request para API {URL} falhou: {resposta.status_code}')
```

Testando a Tool

Vamos testar a função `retorna_temperatura_atual` para garantir que está funcionando corretamente.

```
retorna_temperatura_atual.invoke({'latitude': -30, 'longitude': -50})
```

```
20.3
```

Criando uma Tool de Busca no Wikipedia

Agora, vamos criar uma ferramenta que faz buscas no Wikipedia e retorna resumos das páginas encontradas. Para isso, utilizaremos a biblioteca `wikipedia`.

Passo a Passo para Criar a Função de Busca no Wikipedia

Primeiro, vamos configurar a biblioteca `wikipedia` para utilizar o idioma português e, em seguida, criar a função de busca. Vamos encapsular o código em uma função para facilitar o uso e já o utilizar o decorator tool para gerar uma ferramenta:

```
from langchain.agents import tool

@tool
def busca_wikipedia(query: str):
    """Faz busca no wikipedia e retorna resumos de páginas para a query"""
    titulos_paginas = wikipedia.search(query)
    resumos = []
    for titulo in titulos_paginas[:3]:
        try:
            wiki_page = wikipedia.page(title=titulo, auto_suggest=True)
            resumos.append(f'Título da página: {titulo}\nResumo: {wiki_page.summary}')
        except:
```

```
            pass
    if not resumos:
        return 'Busca não teve retorno'
    else:
        return '\n\n'.join(resumos)
```

Testando a Tool

Vamos testar a função `busca_wikipedia` para garantir que está funcionando corretamente.

```
busca_wikipedia.invoke({'query': 'futebol'})
```

```
'Título da página: Futebol\nResumo: O futebol, também referido como futebol de c

...
```

Integrando as Tools em uma Chain

Vamos agora integrar essas tools em uma chain para que possamos utilizá-las em conjunto com um modelo de linguagem, como o OpenAI.

Configurando a Chain

```
from langchain.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain_core.utils.function_calling import convert_to_openai_function

prompt = ChatPromptTemplate.from_messages([
    ('system', 'Você é um assistente amigável chamado Isaac'),
    ('user', '{input}')
])
chat = ChatOpenAI()

tools = [convert_to_openai_function(busca_wikipedia),
↪  convert_to_openai_function(retorna_temperatura_atual)]
chain = prompt | chat.bind(functions=tools)
```

Testando a Chain

Vamos testar nossa chain para garantir que ela está funcionando corretamente.

```
chain.invoke({'input': 'Olá'})
```

```
AIMessage(content='Olá! Como posso ajudar você hoje?', response_metadata={'token_usage':
↪  {'completion_tokens': 12, 'prompt_tokens': 148, 'total_tokens': 160}, 'model_name':
↪  'gpt-3.5-turbo', 'system_fingerprint': None, 'finish_reason': 'stop', 'logprobs': None},
↪  id='run-81757f8e-4e55-44e8-aac4-9f0f1d8aa09f-0')

chain.invoke({'input': 'Qual é a temperatura de Porto Alegre?'})

AIMessage(content='', additional_kwargs={'function_call': {'arguments':
↪  '{"latitude":-30.0346,"longitude":-51.2177}', 'name': 'retorna_temperatura_atual'}},
↪  response_metadata={'token_usage': {'completion_tokens': 28, 'prompt_tokens': 156,
↪  'total_tokens': 184}, 'model_name': 'gpt-3.5-turbo', 'system_fingerprint': None,
↪  'finish_reason': 'function_call', 'logprobs': None},
↪  id='run-962073d5-6bd1-4d75-86fc-40f9535288d8-0')

chain.invoke({'input': 'Quem foi Isaac Asimov?'})

AIMessage(content='', additional_kwargs={'function_call': {'arguments': '{"query":"Isaac
↪  Asimov"}', 'name': 'busca_wikipedia'}}, response_metadata={'token_usage':
↪  {'completion_tokens': 20, 'prompt_tokens': 154, 'total_tokens': 174}, 'model_name':
↪  'gpt-3.5-turbo', 'system_fingerprint': None, 'finish_reason': 'function_call', 'logprobs':
↪  None}, id='run-1c6dd1a6-a47b-4c5d-887b-beca183fa0a9-0')
```

Na próxima aula, vamos ver de fato como rodar essas tools, adicionando-as em um loop de forma que o modelo não só peça para rodar a ferramenta, mas também de fato a rode e obtenha a informação necessária. Muito obrigado por terem ficado até aqui. Finalmente entreguei umas tools para vocês que são úteis, né? Fazer uma busca no Wikipedia é legal, pegar a temperatura atual agora é legal, e vocês vão ver que as ferramentas vão ficando cada vez mais legais. Espero que tenham gostado, nos vemos no próximo capítulo.

09. Roteamento para Execução Automática de Tools

Olá, pessoal! Bem-vindos a mais um capítulo da nossa apostila. Nós vamos abordar agora um tema essencial para a construção de agentes inteligentes: o roteamento para execução automática de tools. Isso permitirá que o seu modelo de IA utilize ferramentas específicas de forma automática. O passo que estava faltando para criarmos nossa primeira estrutura de Agents. Nossos objetivos, portanto, serão:

- **Criar um roteamento**: Desenvolver um sistema de roteamento que decide quando e como executar uma ferramenta.
- **Utilizar o OpenAIFunctionsAgentOutputParser**: Entender como utilizar esse parser para processar a saída do modelo.

Lembrando que na última aula, criamos duas ferramentas: uma para obter a temperatura atual de uma localização específica e outra para buscar informações na Wikipedia. Agora, vamos dar um passo adiante e integrar essas ferramentas ao nosso modelo de IA, criando um sistema de roteamento que decide automaticamente quando e como executar essas ferramentas. Isso é crucial para construir agentes inteligentes que podem interagir de forma mais eficaz e eficiente com os usuários.

Recriando as Tools

Vamos começar recriando as tools que desenvolvemos na última aula. Isso nos ajudará a refrescar a memória e garantir que estamos todos na mesma página.

```
import requests
import datetime

from langchain.agents import tool
from pydantic import BaseModel, Field

import wikipedia
wikipedia.set_lang('pt')

class RetornTempArgs(BaseModel):
    latitude: float = Field(description='Latitude da localidade que buscamos a temperatura')
    longitude: float = Field(description='Longitude da localidade que buscamos a temperatura')

@tool(args_schema=RetornTempArgs)
def retorna_temperatura_atual(latitude: float, longitude: float):
    '''Retorna a temperatura atual para uma dada coordenada'''

    URL = 'https://api.open-meteo.com/v1/forecast'

    params = {
        'latitude': latitude,
```

```
        'longitude': longitude,
        'hourly': 'temperature_2m',
        'forecast_days': 1,
    }

    resposta = requests.get(URL, params=params)
    if resposta.status_code == 200:
        resultado = resposta.json()

        hora_agora = datetime.datetime.now(datetime.UTC).replace(tzinfo=None)
        lista_horas = [datetime.datetime.fromisoformat(temp_str) for temp_str in
↪   resultado['hourly']['time']]
        index_mais_prox = min(range(len(lista_horas)), key=lambda x: abs(lista_horas[x] -
↪   hora_agora))

        temp_atual = resultado['hourly']['temperature_2m'][index_mais_prox]
        return temp_atual
    else:
        raise Exception(f'Request para API {URL} falhou: {resposta.status_code}')

@tool
def busca_wikipedia(query: str):
    """Faz busca no wikipedia e retorna resumos de páginas para a query"""
    titulos_paginas = wikipedia.search(query)
    resumos = []
    for titulo in titulos_paginas[:3]:
        try:
            wiki_page = wikipedia.page(title=titulo, auto_suggest=True)
            resumos.append(f'Título da página: {titulo}\nResumo: {wiki_page.summary}')
        except:
            pass
    if not resumos:
        return 'Busca não teve retorno'
    else:
        return '\n\n'.join(resumos)
```

Integrando as Tools ao Modelo

Agora que recriamos nossas tools, vamos integrá-las ao nosso modelo de IA. Isso envolve criar uma chain que combina o prompt, o modelo de chat e as tools.

```
from langchain.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain_core.utils.function_calling import convert_to_openai_function

prompt = ChatPromptTemplate.from_messages([
    ('system', 'Você é um assistente amigável chamado Isaac'),
    ('user', '{input}')
])
chat = ChatOpenAI()

tools = [busca_wikipedia, retorna_temperatura_atual]
```

```
tools_json = [convert_to_openai_function(tool) for tool in tools]
tool_run = {tool.name: tool for tool in tools}

chain = prompt | chat.bind(functions=tools_json)
```

Criamos três componentes aqui importantes que não devem passar despercebidos.

- **tools** (list) - Este é o formato padrão de inicialização de ferramentas a um agente do LangChain. Elas devem sempre ser passadas para o agente na forma de uma lista.
- **tools_json** (list) - É uma lista com as tools convertidas para o padrão de jsons que a OpenAI reconhece (como vimos anterioremente nas aulas de reconhecimento de funções).
- **tool_run** (dict) - Ele permite rodar uma tool baseado no nome desta. Será utilizado posteriormente no roteamento.

Criando Roteamento para Rodar Ferramenta

Chamamos de roteamento o processo de análise da saída de um modelo que possui acesso a ferramentas. Neste processo, precisamos entender qual é o tipo de saída que o modelo está nos fornecendo. Caso seja uma saída já com a resposta final do modelo, podemos retorná-la ao usuário; caso o modelo esteja solicitando a chamada de uma ferramenta, temos que chamar essa ferramenta e mostrar o resultado ao usuário.

Adicionando OpenAIFunctionsAgentOutputParser

Para auxiliar neste processo, podemos utilizar o `OpenAIFunctionsAgentOutputParser`. Ele processa a saída de um modelo da OpenAI que possui ferramentas e determina o estado da mensagem devolvida.

```
from langchain.agents.output_parsers import OpenAIFunctionsAgentOutputParser

chain = prompt | chat.bind(functions=tools_json) | OpenAIFunctionsAgentOutputParser()
```

O retorno será um `AgentAction` caso necessite que uma ferramenta seja executada.

```
resultado = chain.invoke({'input': 'Quem foi Isaac Asimov?'})
resultado
```

```
AgentActionMessageLog(tool='busca_wikipedia', tool_input={'query': 'Isaac Asimov'},
↪  log="\nInvoking: `busca_wikipedia` with `{'query': 'Isaac Asimov'}`\n\n\n",
↪  message_log=[AIMessage(content='', additional_kwargs={'function_call': {'arguments':
↪  '{"query":"Isaac Asimov"}', 'name': 'busca_wikipedia'}}, response_metadata={'token_usage':
↪  {'completion_tokens': 20, 'prompt_tokens': 154, 'total_tokens': 174}, 'model_name':
↪  'gpt-3.5-turbo', 'system_fingerprint': None, 'finish_reason': 'function_call', 'logprobs':
↪  None}, id='run-1a81360a-a616-4e49-a4f9-727a23afeef7-0')])
```

```
resultado.tool
```

```
'busca_wikipedia'
```

```
resultado.tool_input
```

```
{'query': 'Isaac Asimov'}
```

E um `AgentFinish`, caso a mensagem esteja finalizada.

```
resultado = chain.invoke({'input': 'Olá'})
resultado
```

```
AgentFinish(return_values={'output': 'Olá! Como posso te ajudar hoje?'}, log='Olá! Como posso
↪  te ajudar hoje?')
```

```
resultado.return_values['output']
```

```
'Olá! Como posso te ajudar hoje?'
```

Rodando as Ferramentas

Podemos criar uma função simples de roteamento para lidar com os dois estados possíveis:

```
from langchain_core.agents import AgentFinish

def roteamento(resultado):
    if isinstance(resultado, AgentFinish):
        return resultado.return_values['output']
    else:
        return tool_run[resultado.tool].run(resultado.tool_input)
```

Podemos adicionar essa função a nossa chain:

```
chain = prompt | chat.bind(functions=tools_json) | OpenAIFunctionsAgentOutputParser() |
↪  roteamento
```

E rodamos primeiro com uma input que não necessitará utilização de tools:

```
chain.invoke({'input': 'Olá'})
```

E a resposta será diretamente a string de resposta:

```
'Olá! Como posso ajudar você hoje?'
```

Agora quando fazemos uma pergunta que necessita de uma tool:

```
chain.invoke({'input': 'Quem foi Isaac Asimov?'})
```

E o resultado é a observação gerada pela tool:

```
'Título da página: Isaac Asimov\nResumo: Isaac Asimov foi um escritor e bioquímico
↪  norte-americano, nascido na Rússia, autor de obras de ficção científica e divulgação
↪  científica.\nAsimov é considerado um dos mestres da ficção científica e, junto com Robert
↪  A. Heinlein e Arthur C. Clarke, foi considerado um dos "três grandes" dessa área da
↪  literatura. A obra mais famosa de Asimov é a Série da Fundação, também conhecida como
↪  Trilogia da Fundação, que faz parte da série do Império Galáctico e que logo combinou com
↪  a Série Robôs.

...
```

Então, pessoal, nessa aula aprendemos a criar um sistema de roteamento para execução automática de tools. Com isso, estamos cada vez mais próximos de criar agentes completos e funcionais.

Desafio - Enviando um Email

E agora que vocês já sabem como criar uma tool, vou propor um desafio para vocês!

Vamos criar uma tool de envio de email e depois fornecê-la ao nosso modelo de linguagem, criando uma estrutura mais simples de um agente.

Vocês podem utilizar o seguinte código em Python retirado do projeto Central de Emails para envio de emails:

```
from email.message import EmailMessage
import smtplib
import ssl

def envia_email(destinatario, titulo, corpo):

    email_usuario = 'ADICIONE SEU EMAIL AQUI'
    senha_app = 'ADICIONE SUA APP KEY AQUI'
    message_email = EmailMessage()
    message_email['From'] = email_usuario
    message_email['To'] = destinatario
    message_email['Subject'] = titulo

    message_email.set_content(corpo)
    safe = ssl.create_default_context()

    with smtplib.SMTP_SSL('smtp.gmail.com', 465, context=safe) as smtp:
        smtp.login(email_usuario, senha_app)
        smtp.sendmail(email_usuario, destinatario, message_email.as_string())
```

Vocês precisam fazer o seguinte: 1. A partir da função, criem uma tool 2. Desenvolvam o argschema da tool, lembrando que todos os argumentos são strings. 3. Combinem a tool a um modelo de linguagem. 4. Enviem um email utilizando o modelo de linguagem para 'aluno@asimov.academy' com o título 'Enviando um email com llm' e uma mensagem bonita no corpo do texto.

Lembrando que este código funciona para emails do Gmail. Você precisa criar uma senha de app no Gmail e adicionar ao código.

10. Explorando Tools Padrão da Biblioteca LangChain

Olá, pessoal! Agora vamos explorar algumas das ferramentas padrão que a biblioteca LangChain oferece. Até aqui entendemos o que são funções externas e tool e como criá-las, mas ignoramos algo muito relevante: o LangChain já possui inúmeras tools criadas pela própria comunidade que podemos acessar e oferecer aos nossos agentes.

Antes de começar o seu processo de criação de tools, sempre sugiro a dar uma pesquisada na documentação do LangChain e verificar se alguém já não criou uma tool para a mesma finalidade, só então comece a criar uma tool própria!

Nesta aula, vamos, portante:

- Conhecer as ferramentas padrão do LangChain.
- Aprender a utilizar essas ferramentas padrões.
- Entender as diferentes formas de importar uma tool no LangChain.
- Explorar exemplos práticos de uso das tools.

Explorando Tools Padrão

A biblioteca LangChain já possui diversas ferramentas padrão construídas que podemos utilizar. Elas podem ser verificadas neste link. Vamos mostrar como utilizar algumas.

ArXiv

Esta ferramenta utiliza a biblioteca do arXiv para retornar resumos de artigos científicos de um determinado tema solicitado. Temos três formas em geral para chamar uma tool, da mais baixo a mais alto nível:

1. **Criando a tool utilizando o Wrapper**: No LangChain foram criados diferentes wrappers para APIs e bibliotecas externas que facilitam a utilização das mesmas e permitem uma rápida criação de uma nova tool. Ela é mais customizável, pois podemos modificar as descrições e argumentos da tool às nossas necessidades, além de podermos modificar facilmente os parâmetros do wrapper.
2. **Instanciando a tool já criada**: Em geral, uma tool pronta já está criada dentro da biblioteca e podemos acessá-la diretamente.
3. **Utilizando o `load_tools`**: O LangChain possui uma ferramenta especial chamada `load_tools` que permite o carregamento de diversas ferramentas ao mesmo tempo.

Vamos explorar os três métodos agora:

Criando Tool Manualmente Através do Wrapper Primeira tool que a gente vai explorar vai ser a do ArXiv. Caso você não saiba, o ArXiv é um repositório de papers, de artigos científicos, e a gente consegue acessar o ArXiv através do LangChain e buscar o resumo de vários artigos científicos que estão na internet.

```
from langchain_community.utilities.arxiv import ArxivAPIWrapper
from pydantic import BaseModel, Field
from langchain.tools import StructuredTool

class ArxivArgs(BaseModel):
    query: str = Field(description='Query de busca no ArXiv')

tool_arxiv = StructuredTool.from_function(
    func=ArxivAPIWrapper(top_k_results=2).run,
    args_schema=ArxivArgs,
    name='arxiv',
    description=(
        "Uma ferramenta em torno do Arxiv.org. "
        "Útil para quando você precisa responder a perguntas sobre Física, Matemática, "
        "Ciência da Computação, Biologia Quantitativa, Finanças Quantitativas, Estatística, "
        "Engenharia Elétrica e Economia utilizando artigos científicos do arxiv.org. "
        "A entrada deve ser uma consulta de pesquisa em inglês."
    ),
    return_direct=True
)

print('Descrição:', tool_arxiv.description)
print('Args:', tool_arxiv.args)
```

Instanciando a Tool Já Criada Outra forma de criar a tool é instanciando diretamente a tool já criada dentro da biblioteca.

```
from langchain_community.tools.arxiv import ArxivQueryRun

tool_arxiv = ArxivQueryRun(api_wrapper=ArxivAPIWrapper(top_k_results=2))
print('Descrição:', tool_arxiv.description)
```

Utilizando o `load_tools` A última forma é utilizando o `load_tools`, que é mais alto nível e menos customizável, mas também mais simples.

```
from langchain.agents import load_tools

tools = load_tools(['arxiv'])
tool_arxiv = tools[0]
print('Descrição:', tool_arxiv.description)
```

Exemplo de uso E para rodar, basta utilizar o método run ou invoke:

```
tool_arxiv.run({'query': 'llm'})
```

E a resposta será:

```
"Published: 2024-01-08\nTitle: A Survey of Large Language Models for Code: Evolution,
↪  Benchmarking, and Future Trends\nAuthors: Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen
↪  Zhang, Dewu Zheng, Mingxi Ye, Jiachi Chen\nSummary: General large language models (LLMs),
↪  represented by ChatGPT, have\ndemonstrated
...
```

Python REPL

Outra ferramenta interessante é o Python REPL, que permite pode dar a um modelo a capacidade de rodar códigos Python diretamente.

```
from langchain.experimental.tools.python import PythonREPLTool

tool_python = PythonREPLTool()
print('Descrição:', tool_repl.description)
print('Args:', tool_repl.args)
```

```
Descrição: A Python shell. Use this to execute python commands. Input should be a valid python
↪  command. If you want to see the output of a value, you should print it out with
↪  `print(...)`.
Args: {'query': {'title': 'Query', 'type': 'string'}}
```

Exemplo de uso

```
result = tool_python.run("print('Hello, World!')")
print(result)
```

E a resposta será:

```
'Hello, World!'
```

StackOverflow

Podemos também utilizar a ferramenta do StackOverflow para buscar respostas de programação.

```
from langchain.agents import load_tools

tools = load_tools(['stack_exchange'])
tool_stack = tools[0]
print('Descrição:', tool_stack.description)
print('Args:', tool_stack.args)
```

```
Descrição: A wrapper around StackExchange. Useful for when you need to answer specific
↪  programming questionscode excerpts, code examples and solutionsInput should be a fully
↪  formed question.
Args: {'query': {'title': 'Query', 'type': 'string'}}
```

Exemplo de uso

```
tool_stack.run({'query': 'LangChain Agents'})
```

E a resposta será:

```
'Question: While using groq for my project im getting this error\ndef
↪  claim_agent(user_question):\n    memory =
↪  ConversationBufferWindowMemory(ai_prefix=&quot;Insurance <span
↪  class="highlight">Agent</span>&quot;, k=20)\n    prompt_template = PromptTemplate(\n
↪  input_variables=[&#39;history&#39;, &#39;input&#39;], &hellip;
↪  &quot;D:\\Project_tests.venv\\Lib\\site-packages\\<span
↪  class="highlight">langchain</span>\\chains\\base.py&quot;, line 163, in invoke\nraise
↪  e\nFile

...
```

File System

Por último, vamos explorar o toolkit de gerenciamento de arquivos, que permite criar, deletar, mover e listar arquivos.

```
from langchain_community.agent_toolkits.file_management import FileManagementToolkit

toolkit = FileManagementToolkit(root_dir='arquivos')
tools = toolkit.get_tools()
for tool in tools:
    print('Name:', tool.name)
    print('Descrição:', tool.description)
    print('Args:', tool.args)
    print()
```

```
Name: copy_file
Descrição: Create a copy of a file in a specified location
Args: {'source_path': {'title': 'Source Path', 'description': 'Path of the file to copy',
↪  'type': 'string'}, 'destination_path': {'title': 'Destination Path', 'description': 'Path
↪  to save the copied file', 'type': 'string'}}

Name: file_delete
Descrição: Delete a file
Args: {'file_path': {'title': 'File Path', 'description': 'Path of the file to delete',
↪  'type': 'string'}}

Name: file_search
Descrição: Recursively search for files in a subdirectory that match the regex pattern
Args: {'dir_path': {'title': 'Dir Path', 'description': 'Subdirectory to search in.',
↪  'default': '.', 'type': 'string'}, 'pattern': {'title': 'Pattern', 'description': 'Unix
↪  shell regex, where * matches everything.', 'type': 'string'}}

Name: move_file
Descrição: Move or rename a file from one location to another
```

```
Args: {'source_path': {'title': 'Source Path', 'description': 'Path of the file to move',
↪  'type': 'string'}, 'destination_path': {'title': 'Destination Path', 'description': 'New
↪  path for the moved file', 'type': 'string'}}

Name: read_file
Descrição: Read file from disk
Args: {'file_path': {'title': 'File Path', 'description': 'name of file', 'type': 'string'}}

Name: write_file
Descrição: Write file to disk
Args: {'file_path': {'title': 'File Path', 'description': 'name of file', 'type': 'string'},
↪  'text': {'title': 'Text', 'description': 'text to write to file', 'type': 'string'},
↪  'append': {'title': 'Append', 'description': 'Whether to append to an existing file.',
↪  'default': False, 'type': 'boolean'}}

Name: list_directory
Descrição: List files and directories in a specified folder
Args: {'dir_path': {'title': 'Dir Path', 'description': 'Subdirectory to list.', 'default':
↪  '.', 'type': 'string'}}
```

Podemos verificar que este toolkit apresenta 7 ferramentas diferentes:

- copy_file: copia arquivos
- file_delete: deleta arquivos
- file_search: busca arquivos
- move_file: move arquivos
- read_file: lê arquivos
- write_file: escreve arquivos
- list_directory: lista arquivos em uma pasta

Exemplo de uma aplicação

Podemos utilizar o toolkit de file system para criar uma aplicação que gerencia arquivos no nosso computador. Para isso, primeiro importamos as ferramentas:

```
from langchain_community.agent_toolkits.file_management.toolkit import FileManagementToolkit

tool_kit = FileManagementToolkit(
    root_dir='arquivos',
    selected_tools=['write_file', 'read_file', 'file_search','list_directory']
)
tools = tool_kit.get_tools()
```

Selecionamos apenas três ferrametas:

- file_search: busca arquivos
- read_file: lê arquivos
- write_file: escreve arquivos

- list_directory: lista arquivos em uma pasta

Quando passamos o parâmetro **root_dir**='arquivos', estamos dizendo às ferramentas que elas somente terão acesso aos arquivos dentro da pasta arquivos.

Agora recriamos nossa função de roteamento, como visto no capítulo anterior:

```
from langchain_core.agents import AgentFinish

def roteamento(resultado):
    if isinstance(resultado, AgentFinish):
        return resultado.return_values['output']
    else:
        return tool_run[resultado.tool].run(resultado.tool_input)
```

E criamos nossa chain:

```
from langchain.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain_core.utils.function_calling import convert_to_openai_function
from langchain.agents.output_parsers import OpenAIFunctionsAgentOutputParser


tools_json = [convert_to_openai_function(tool) for tool in tools]
tool_run = {tool.name: tool for tool in tools}

prompt = ChatPromptTemplate.from_messages([
    ('system', 'Você é um assistente amigável chamado Isaac capaz de gerenciar arquivos'),
    ('user', '{input}')
])
chat = ChatOpenAI()
chain = (prompt
         | chat.bind(functions=tools_json)
         | OpenAIFunctionsAgentOutputParser()
         | roteamento)
```

E agora já podemos utilizar nosso gerenciador de arquivos:

```
chain.invoke({'input': 'O que você é capaz de fazer?'})
```

```
'Olá! Eu sou um assistente amigável chamado Isaac e sou capaz de gerenciar arquivos. Posso
↪  escrever texto em arquivos, ler o conteúdo de arquivos, pesquisar arquivos com base em
↪  padrões de regex e listar arquivos e diretórios em uma pasta específica. Se precisar de
↪  ajuda com alguma dessas tarefas, fique à vontade para me pedir! Como posso ajudar você
↪  hoje?'
```

```
chain.invoke({'input': 'Crie um arquivo txt chamado notas com o seguinte conteúdo "Isso foi
↪  feito por uma LLM"'})
```

```
'File written successfully to notas.txt.'
```

```
chain.invoke({'input': 'Leia o arquivo notas.txt'})
```

```
'Isso foi feito por uma LLM'
```

E assim fizemos um modelo capaz de escrever arquivos e lê-los em nosso pc. Incrível, não?

Espero que tenham gostado e que essas ferramentas ajudem vocês em seus projetos!

11. Como um Agent é construído

Finalmente! Chegamos aos Agents! Parabéns, meus amigos e minhas amigas, por terem chegado até aqui. Precisamos de uma base bem sólida para entender profundamente os Agents, mas agora que estamos aqui, tudo vai fluir mais rápido. Vocês já estão familiarizados com a criação de chains, Prompt Templates, Output Parsers e Tools, então estamos prontos para dar o próximo passo. Mas antes, vamos fazer uma pequena definição conceitual sobre o que são os agents!

Definindo um agente

Os agentes baseados em modelos de linguagem são aplicações que podem executar tarefas complexas utilizando uma arquitetura que combina LLMs com módulos essenciais, como planejamento e memória. Em essência, um agente atua como um assistente inteligente que pode processar informações, tomar decisões e realizar ações com base nas solicitações dos usuários.

Estrutura de um Agente

Um agente é composto por alguns componentes principais:

- **Modelo LLM**: Este é o "cérebro" do agente, que coordena a lógica e as ações necessárias para responder a uma solicitação.
- **Planejamento**: Esse módulo ajuda o agente a decompor tarefas complexas em subtarefas mais simples, facilitando a resolução de problemas.
- **Memória**: Os agentes têm a capacidade de armazenar informações sobre interações passadas, o que os ajuda a manter o contexto e a continuidade nas conversas.
- **Ferramentas**: Os agentes podem utilizar ferramentas externas para realizar ações específicas, como buscar informações em APIs ou executar comandos.

Entendendo cada componente

Modelo LLM

Este é o núcleo do agente é o componente central que gerencia a lógica e as ações necessárias para responder a uma solicitação. Um modelo de linguagem grande (LLM) com capacidades de uso geral serve como o "cérebro" do agente. Esse núcleo é ativado usando um template de prompt que contém detalhes importantes sobre como o agente operará e quais ferramentas ele terá acesso.

Embora não seja obrigatório, um agente pode ser perfilado ou receber uma persona para definir seu papel. Essas informações de perfil geralmente são escritas no prompt e podem incluir detalhes específicos, como informações sobre o papel, personalidade, informações sociais e outras informações demográficas. Isso ajuda a moldar a forma como o agente interage com os usuários e como ele toma decisões. As estratégias para definir um perfil de agente podem incluir elaboração manual, geração por LLM ou orientada por dados.

Planejamento

O módulo de planejamento é fundamental para que o agente consiga dividir os passos necessários ou subtarefas que ele resolverá individualmente para responder à solicitação do usuário. Essa etapa é crucial, pois permite que o agente raciocine melhor sobre o problema e encontre uma solução de forma confiável. O planejamento utiliza um LLM para decompor um plano detalhado que incluirá subtarefas para ajudar a abordar a pergunta do usuário.

Memória

A memória é responsável por armazenar os registros internos do agente, incluindo pensamentos, ações e observações passadas do ambiente, além de todas as interações entre o agente e o usuário. Existem dois tipos principais de memória:

- **Memória de Curto Prazo**: Inclui informações contextuais sobre as situações atuais do agente. Isso é tipicamente realizado por meio de aprendizado em contexto, o que significa que é curto e finito devido a restrições de janela de contexto.
- **Memória de Longo Prazo**: Inclui os comportamentos e pensamentos passados do agente que precisam ser retidos e recuperados ao longo de um período prolongado. Isso geralmente utiliza um armazenamento vetorial externo acessível por meio de recuperação rápida e escalável para fornecer informações relevantes para o agente conforme necessário.

A memória híbrida combina tanto a memória de curto prazo quanto a de longo prazo, melhorando a capacidade do agente de raciocinar a longo prazo e acumular experiências.

Ferramentas

As ferramentas são um conjunto de recursos que permitem que o agente LLM interaja com ambientes externos, como a API de busca do Wikipedia, ferramentas de busca na web ou interpretadores de código. Elas também podem incluir bancos de dados, bases de conhecimento e modelos externos. Quando o agente interage com ferramentas externas, ele executa tarefas por meio de fluxos de trabalho

que ajudam a obter observações ou informações necessárias para completar subtarefas e satisfazer a solicitação do usuário.

Construindo um Agent

A ideia central dos Agents é usar um modelo de linguagem para escolher uma sequência de ações a serem executadas. Nas chains, uma sequência de ações é definida diretamente no código, ou seja, recebe uma input do usuário, depois faz um parsing da output, depois passa para outra chain, utiliza uma ferramenta, etc. Tudo é pré-definido por quem desenvolveu a chain. **Já nos Agents, um modelo de linguagem é usado como um motor de raciocínio para determinar quais ações devem ser tomadas e em qual ordem.**

Criando as Tools que Usaremos

Vamos começar criando as Tools que nossos Agents utilizarão. Essas Tools são funções que o modelo de linguagem pode chamar para obter informações ou realizar ações específicas.

```
import requests
import datetime

from langchain.agents import tool
from pydantic import BaseModel, Field

import wikipedia
wikipedia.set_lang('pt')

class RetornTempArgs(BaseModel):
    latitude: float = Field(description='Latitude da localidade que buscamos a temperatura')
    longitude: float = Field(description='Longitude da localidade que buscamos a temperatura')

@tool(args_schema=RetornTempArgs)
def retorna_temperatura_atual(latitude: float, longitude: float):
    '''Retorna a temperatura atual para uma dada coordenada'''

    URL = 'https://api.open-meteo.com/v1/forecast'

    params = {
        'latitude': latitude,
        'longitude': longitude,
        'hourly': 'temperature_2m',
        'forecast_days': 1,
    }

    resposta = requests.get(URL, params=params)
    if resposta.status_code == 200:
        resultado = resposta.json()
```

```
        hora_agora = datetime.datetime.now(datetime.UTC).replace(tzinfo=None)
        lista_horas = [datetime.datetime.fromisoformat(temp_str) for temp_str in
↪   resultado['hourly']['time']]
        index_mais_prox = min(range(len(lista_horas)), key=lambda x: abs(lista_horas[x] -
↪   hora_agora))

        temp_atual = resultado['hourly']['temperature_2m'][index_mais_prox]
        return f'{temp_atual}°C'
    else:
        raise Exception(f'Request para API {URL} falhou: {resposta.status_code}')

@tool
def busca_wikipedia(query: str):
    """Faz busca no wikipedia e retorna resumos de páginas para a query"""
    titulos_paginas = wikipedia.search(query)
    resumos = []
    for titulo in titulos_paginas[:3]:
        try:
            wiki_page = wikipedia.page(title=titulo, auto_suggest=True)
            resumos.append(f'Título da página: {titulo}\nResumo: {wiki_page.summary}')
        except:
            pass
    if not resumos:
        return 'Busca não teve retorno'
    else:
        return '\n\n'.join(resumos)
```

Revisando a Utilização das Tools

Vamos revisar a utilização das Tools que acabamos de criar. Primeiro, configuramos o prompt e o modelo de linguagem.

```
from langchain.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain_core.utils.function_calling import convert_to_openai_function

prompt = ChatPromptTemplate.from_messages([
    ('system', 'Você é um assistente amigável chamado Isaac'),
    ('user', '{input}')
])
chat = ChatOpenAI()

tools = [busca_wikipedia, retorna_temperatura_atual]
tools_json = [convert_to_openai_function(tool) for tool in tools]
tool_run = {tool.name: tool for tool in tools}

chain = prompt | chat.bind(functions=tools_json)
```

Em seguida, adicionamos o OutputParser para interpretar as respostas do modelo.

```
from langchain.agents.output_parsers import OpenAIFunctionsAgentOutputParser
```

```
chain = prompt | chat.bind(functions=tools_json) | OpenAIFunctionsAgentOutputParser()
```

Agora, vamos invocar a chain com uma pergunta específica.

```
resposta = chain.invoke({'input': 'Qual a temperatura em Floripa?'})
resposta
```

A resposta inclui informações sobre a Tool que foi chamada e os inputs utilizados.

```
resposta.tool
```

```
'retorna_temperatura_atual'
```

```
resposta.tool_input
```

```
{'latitude': -27.5953787, 'longitude': -48.5480499}
```

```
resposta.message_log
```

```
[AIMessage(content='', additional_kwargs={'function_call': {'arguments':
↪  '{"latitude":-27.5953787,"longitude":-48.5480499}', 'name': 'retorna_temperatura_atual'}},
↪  response_metadata={'token_usage': {'completion_tokens': 30, 'prompt_tokens': 153,
↪  'total_tokens': 183}, 'model_name': 'gpt-3.5-turbo', 'system_fingerprint': None,
↪  'finish_reason': 'function_call', 'logprobs': None},
↪  id='run-9ee389cd-e54e-477d-b466-abee4c4081be-0')]
```

Adicionando o Raciocínio do Agent às Mensagens (agent_scratchpad)

Temos que adicionar junto às nossas mensagens um campo que armazenará o raciocínio atual do modelo chamado `agent_scratchpad`. Para isso, utilizamos um `MessagesPlaceholder` no nosso prompt. Ele guardará espaço para o raciocínio e, caso o modelo não esteja gerando um raciocínio no momento, o `MessagesPlaceholder` não é utilizado.

```
from langchain.prompts import MessagesPlaceholder

prompt = ChatPromptTemplate.from_messages([
    ('system', 'Você é um assistente amigável chamado Isaac'),
    ('user', '{input}'),
    MessagesPlaceholder(variable_name='agent_scratchpad')
])

chain = prompt | chat.bind(functions=tools_json) | OpenAIFunctionsAgentOutputParser()
```

Vamos invocar a chain novamente, agora incluindo o `agent_scratchpad`.

```
resposta_inicial = chain.invoke({
    'input': 'Qual a temperatura em Floripa?',
    'agent_scratchpad': []})
resposta_inicial
```

A resposta inicial inclui a Tool que foi chamada e os inputs utilizados.

```
observacao = tool_run[resposta_inicial.tool].run(resposta_inicial.tool_input)
observacao
```

```
'21.9°C'
```

Podemos utilizar a função `format_to_openai_function_messages` para modificar o formato da resposta de forma que ela possa ser enviada, junto da observação, de volta ao modelo. No caso, o que está ocorrendo é que o modelo está pedindo que uma Tool seja rodada, estamos rodando a Tool e gerando uma observação, e agora enviamos novamente para o modelo a pergunta original, a mensagem do próprio modelo dizendo que precisava que a Tool fosse rodada e a observação gerada pela ferramenta.

```
from langchain.agents.format_scratchpad import format_to_openai_function_messages

format_to_openai_function_messages([(resposta_inicial, observacao)])
```

```
[AIMessage(content='', additional_kwargs={'function_call': {'arguments':
↪  '{"latitude":-27.5954,"longitude":-48.548}', 'name': 'retorna_temperatura_atual'}},
↪  response_metadata={'token_usage': {'completion_tokens': 27, 'prompt_tokens': 153,
↪  'total_tokens': 180}, 'model_name': 'gpt-3.5-turbo', 'system_fingerprint': None,
↪  'finish_reason': 'function_call', 'logprobs': None},
↪  id='run-722c78ea-a704-41e2-9205-564c3e8270ab-0'),

 FunctionMessage(content='21.9°C', name='retorna_temperatura_atual')]
```

Vamos invocar a chain novamente, agora incluindo a observação.

```
resposta_final = chain.invoke({
    'input': 'Qual a temperatura em Floripa?',
    'agent_scratchpad': format_to_openai_function_messages([(resposta_inicial, observacao)])})
resposta_final
```

```
AgentFinish(return_values={'output': 'A temperatura em Florianópolis é de 21.9°C no
↪  momento.'}, log='A temperatura em Florianópolis é de 21.9°C no momento.')
```

Criando um Loop de Raciocínio

Por fim, podemos criar um loop que adiciona automaticamente as chamadas de função e observações e fica chamando o modelo novamente até que a mensagem de `AgentFinish` seja recebida.

```
from langchain.schema.agent import AgentFinish

def run_agent(input):
    passos_intermediarios = []
    while True:
        resposta = chain.invoke({
            'input': input,
            'agent_scratchpad': format_to_openai_function_messages(passos_intermediarios)
        })
        if isinstance(resposta, AgentFinish):
```

```
            return resposta
        observacao = tool_run[resposta.tool].run(resposta.tool_input)
        passos_intermediarios.append((resposta, observacao))
```

Vamos testar o loop.

```
run_agent('Qual é a temperatura de Floripa?')
```

```
AgentFinish(return_values={'output': 'A temperatura atual em Florianópolis é de 21.5°C.'},
↪  log='A temperatura atual em Florianópolis é de 21.5°C.')
```

Modificamos um pouco o formato para padronizar ao funcionamento do LangChain para Agents.

```
from langchain.schema.agent import AgentFinish
from langchain.schema.runnable import RunnablePassthrough

pass_through = RunnablePassthrough.assign(
    agent_scratchpad = lambda x: format_to_openai_function_messages(x['intermediate_steps'])
)
chain = pass_through | prompt | chat.bind(functions=tools_json) |
↪  OpenAIFunctionsAgentOutputParser()

def run_agent(input):
    passos_intermediarios = []
    while True:
        resposta = chain.invoke({
            'input': input,
            'intermediate_steps': passos_intermediarios
        })
        if isinstance(resposta, AgentFinish):
            return resposta
        observacao = tool_run[resposta.tool].run(resposta.tool_input)
        passos_intermediarios.append((resposta, observacao))
```

Essa modificação é implementada utilizando um RunnablePassthrough. Esse *runnable* funciona como um dicionário, ao utilizarmos o método ***assign***, ele adiciona uma chave nova ao dicionário. No caso a chave sendo adicionada é de ***agent_scratchpad***, que será derivada da chave ***intermediate_steps***, mas convertida para o formato da openai utilizando a função ***format_to_openai_function_messages***.

```
pass_through.invoke({'input': 'Qual é a temperatura de Floripa?', 'intermediate_steps': []})
```

```
{'input': 'Qual é a temperatura de Floripa?',
 'intermediate_steps': [],
 'agent_scratchpad': []}
```

E testando novamente o podemos ver que o funcionamento é igual.

```
run_agent('Qual é a temperatura de Floripa?')
```

```
AgentFinish(return_values={'output': 'A temperatura atual em Floripa é de 21.5°C.'}, log='A
↪  temperatura atual em Floripa é de 21.5°C.')
```

O que Temos no Final?

No final temos duas estruturas: Agent + AgentExecutor segundo a nomenclatura do LangChain.

Um Agent

```
prompt = ChatPromptTemplate.from_messages([
    ('system', 'Você é um assistente amigável chamado Isaac'),
    ('user', '{input}'),
    MessagesPlaceholder(variable_name='agent_scratchpad')
])

pass_through = RunnablePassthrough.assign(
    agent_scratchpad = lambda x: format_to_openai_function_messages(x['intermediate_steps'])
)
chain = pass_through | prompt | chat.bind(functions=tools_json) |
↪  OpenAIFunctionsAgentOutputParser()
```

Um AgentExecutor

```
def run_agent(input):
    passos_intermediarios = []
    while True:
        resposta = chain.invoke({
            'input': input,
            'intermediate_steps': passos_intermediarios
        })
        if isinstance(resposta, AgentFinish):
            return resposta
        observacao = tool_run[resposta.tool].run(resposta.tool_input)
        passos_intermediarios.append((resposta, observacao))
```

Nos próximos capítulos, veremos que o LangChain já tem Agents e AgentExecutors prontos, então começaremos a utilizá-los para simplificar o processo.

Agora espero que tenha ficado claro para vocês o que é um Agent: ele é uma estrutura capaz de raciocinar, armazenar passos intermediários (no nosso caso, no `Agent Scratchpad`) e agregar mais informações ao modelo de linguagem, aumentando sua capacidade de fornecer respostas precisas. Isso é um Agent.

Espero que tenham gostado e que esta tenha sido uma boa introdução aos Agents!

12. Criando um AgentExecutor com Memória

Olá, pessoal! Na última aula, criamos uma estrutura de Agent e AgentExecutor manualmente, a premira contenda a chain executada e a segunda implementa o loop para executar todos os passos do raciocínio do modelo. Nesta aula, vamos explorar como criar um `AgentExecutor` próprio do LangChain e como adicionar memória a ele. Isso é crucial para desenvolver uma aplicação de chat funcional, onde o agente pode lembrar das interações anteriores com o usuário. Nosso objetivos, portanto, serão:

- Entender a importância de adicionar memória a um `AgentExecutor`.
- Aprender a criar e configurar um `AgentExecutor` com memória.

Por que as memórias são importantes em uma aplicação de IA?

Para criarmos uma aplicação de chat funcional com nosso agente, é necessário que ele tenha a capacidade de armazenar as informações trocadas com o usuário. Isso é essencial para que o agente possa responder de forma contextual e coerente, lembrando das interações anteriores.

Ao utilizarmos a função run_agent criada no capítulo anterior, vemos que nosso modelo não posui memória e, por isso, não é capaz nem de decorar o nosso nome. Se primeiro eu disser meu nome:

```
resposta = run_agent({'input': 'Olá, eu sou o Adriano'})
resposta
```

```
AgentFinish(return_values={'output': 'Olá, Adriano! Como posso te ajudar hoje?'}
```

E depois perguntar para ele o meu nome:

```
resposta = run_agent({'input': 'Qual o meu nome?'})
resposta
```

```
AgentFinish(return_values={'output': 'Seu nome é Isaac! Como posso te ajudar hoje?'}, log='Seu
↪  nome é Isaac! Como posso te ajudar hoje?')
```

Ele só não soube o meu nome como ainda se confundiu com o seu próprio nome. Nada bom, um modelo assim para conversação tem pouca utilidade. Mas vamos ver agora como superar essa limitação utilizando os AgentExecutors do próprio LangChain.

Criando um AgentExecutor do LangChain

Criando as Tools que Usaremos

Primeiro, vamos criar as ferramentas (`tools`) que nosso agente vai utilizar. As tools de busca_wikipedia e retorna_temperatura_atual já foram criadas anteriormente, portanto não vamos criá-las novamente.

```
tools = [busca_wikipedia, retorna_temperatura_atual]
```

Adicionando Memória

Agora, vamos adicionar memória ao nosso `AgentExecutor` para que ele possa lembrar das interações anteriores. Utilizaremos o ***ConversationBufferMemory***. As memórias são conteúdo do nosso curso inicila de LangChain, caso você queira saber mais sugerimos que deem uma olhada lá.

```
from langchain.memory import ConversationBufferMemory
memory = ConversationBufferMemory(
    return_messages=True,
    memory_key='chat_history'
)

prompt = ChatPromptTemplate.from_messages([
    ('system', 'Você é um assistente amigável chamado Isaac'),
    MessagesPlaceholder(variable_name='chat_history'),
    ('user', '{input}'),
    MessagesPlaceholder(variable_name='agent_scratchpad')
])

pass_through = RunnablePassthrough.assign(
    agent_scratchpad = lambda x: format_to_openai_function_messages(x['intermediate_steps'])
)
agent_chain = pass_through | prompt | chat.bind(functions=tools_json) |
↪  OpenAIFunctionsAgentOutputParser()
```

Adicionamos no prompt um campo com MessagesPlaceholder com a variável “chat_history”. Neste local será adicionado a lista de histórico de mensagens da memória.

Configurando o AgentExecutor com Memória

Agora podemos importar o AgentExecutos do Langchain e adicionar a chain do nosso agent e a memória que criamos previamente:

```
from langchain.agents import AgentExecutor

agent_executor = AgentExecutor(
    agent=agent_chain,
    tools=tools,
    memory=memory,
    verbose=True
)
```

Testando o AgentExecutor com Memória

Vamos testar nosso `AgentExecutor` com memória para ver como ele se comporta.

```
resposta = agent_executor.invoke({'input': 'Meu nome é Adriano'})
print(resposta)
```

```
[1m> Entering new AgentExecutor chain...[0m

[32;1m[1;3mOlá, Adriano! Como posso te ajudar hoje?[0m

[1m> Finished chain.[0m

{'input': 'Meu nome é Adriano',

 'chat_history': [HumanMessage(content='Meu nome é Adriano'),

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

 'output': 'Olá, Adriano! Como posso te ajudar hoje?'}
```

```
resposta = agent_executor.invoke({'input': 'Qual o meu nome?'})
print(resposta)
```

```
[1m> Entering new AgentExecutor chain...[0m

[32;1m[1;3mSeu nome é Adriano. Como posso te ajudar, Adriano?[0m

[1m> Finished chain.[0m

{'input': 'Qual o meu nome?',

 'chat_history': [HumanMessage(content='Meu nome é Adriano'),

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

  HumanMessage(content='Qual o meu nome?'),

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

 'output': 'Seu nome é Adriano. Como posso te ajudar, Adriano?'}
```

E podemos perceber que o nosso `AgentExecutor` mantém nosso histórico de conversa e está pronto para gerar uma aplicação real!

E é isso, pessoal! Agora vocês sabem como criar um `AgentExecutor` com memória utilizando o LangChain. Isso é fundamental para desenvolver aplicações de chat mais realistas e funcionais, onde o agente pode lembrar das interações anteriores e responder de forma mais contextual.

13. Agent Types - Entendendo os Agents Tool Calling e ReAct

Olá, pessoal! Agora vamos explorar um tópico super interessante e essencial para quem quer dominar a criação de agentes inteligentes: os diferentes tipos de Agents. Vamos focar em dois tipos principais: Tool Calling e ReAct. Nossos objetivos serão:

- Compreender o conceito de Agent Types.
- Aprender a criar e utilizar um Tool Calling Agent.
- Entender o funcionamento e a criação de um ReAct Agent.
- Explorar exemplos práticos de cada tipo de Agent.

Por que temos diferentes tipos de Agents?

Conforme a utilização de modelos de linguagem e ferramentas foi se consolidando, percebeu-se que modelos mais simples podem performar muito bem quando utilizamos técnicas de engenharia de prompt. Isso nos leva a criar diferentes tipos de Agents, cada um otimizado para utilizar ferramentas de maneira eficiente. Vamos explorar os dois tipos mais utilizados: Tool Calling e ReAct.

Tool Calling Agent

O que é?

O Tool Calling Agent é um tipo de agente que detecta quando uma ou mais ferramentas devem ser acionadas e responde com os dados que devem ser passados para essas ferramentas. Ele permite que o modelo escolha inteligentemente gerar um objeto estruturado, como JSON, contendo argumentos para acionar essas ferramentas. O tool calling agent é ótimo para modelos que já possuem um fine tuning para utilização de ferramentas, como os modelos GPT, Gemini e Claude.

Exemplo Prático

Vamos criar um pequeno Agent que tem acesso à ferramenta do Python para fazer cálculos mais rebuscados. Primeiro, importamos a ferramenta:

```
from langchain_experimental.tools import PythonAstREPLTool

tools = [PythonAstREPLTool()]
```

Agora, vamos criar o nosso Agent:

```
from langchain.agents import create_tool_calling_agent, AgentExecutor
from langchain.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI

system_msg = '''Você é um assistente amigável.
Certifique-se de usar a ferramenta PythonAstREPLTool para auxiliar a responder as perguntas'''

prompt = ChatPromptTemplate.from_messages([
    ('system', system_msg),
    ('placeholder', '{chat_history}'),
    ('human', '{input}'),
    ('placeholder', '{agent_scratchpad}')
])

chat = ChatOpenAI()
agent = create_tool_calling_agent(chat, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
```

Vamos rodar o Agent e fazer uma pergunta:

```
agent_executor.invoke({'input': 'Qual é o décimo valor da sequência Fibonacci?'})
```

O Agent tentará utilizar a ferramenta Python para calcular o décimo valor da sequência Fibonacci. Ele pode cometer alguns erros no processo, mas eventualmente deve conseguir chegar ao resultado correto.

```
{'input': 'Qual é o décimo valor da sequência Fibonacci?',
 'output': 'O décimo valor da sequência Fibonacci é 34.'}
```

Podemos tentar outra pergunta:

```
agent_executor.invoke({'input': 'Quantas letras tem a palavra LangChain?'})
```

```
{'input': 'Quantas letras tem a palavra LangChain?',
 'output': 'A palavra "LangChain" tem 9 letras.'}
```

Quando utilizamos o parâmetro verbose=True, o LangChain nos mostra todos os passos que o modelo está dando para chegar na resposta. No último caso, podemos ver que o modelo seguiu a seguinte linha de raciocínio:

```
> Entering new AgentExecutor chain...

Invoking: `python_repl_ast` with `{'query': "len('LangChain')"}`
9

A palavra "LangChain" tem 9 letras.

> Finished chain.
{'input': 'Quantas letras tem a palavra LangChain?',
 'output': 'A palavra "LangChain" tem 9 letras.'}
```

Podemos ver que ele solicitou a utilização da ferramenta *python_repl_ast* para conseguir gerar a resposta ao usuário. É exatamente o que é esperado de um agente, que ele raciocine, esteja ciente das ferramentas que possui e as utilize para gerar uma resposta mais acurada.

Refinando a System Message

Como falamos, o comportamento do modelo pode melhorar bastante com um pouco de engenharia de prompt. Vamos criar uma System Message mais refinada para melhorar a performance do nosso Agent:

```
from langchain.agents import create_tool_calling_agent, AgentExecutor
from langchain.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI

system_msg = """Você é um agente projetado para escrever e executar código Python para
↪  responder perguntas.
Você tem acesso a um REPL Python, que pode usar para executar código Python.
Se encontrar um erro, depure o código e tente novamente.
Use apenas a saída do seu código para responder à pergunta.
Você pode conhecer a resposta sem executar nenhum código, mas deve ainda assim executar o
↪  código para obter a resposta.
Se não parecer possível escrever código para responder à pergunta, simplesmente retorne "Não
↪  sei" como a resposta."""

prompt = ChatPromptTemplate.from_messages([
    ('system', system_msg),
    ('placeholder', '{chat_history}'),
    ('human', '{input}'),
    ('placeholder', '{agent_scratchpad}')
])

chat = ChatOpenAI()
agent = create_tool_calling_agent(chat, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
```

Vamos rodar novamente:

```
agent_executor.invoke({'input': 'Quantas letras tem a palavra LangChain?'})
```

```
{'input': 'Quantas letras tem a palavra LangChain?',
 'output': 'A palavra "LangChain" tem 9 letras.'}
```

ReAct Agent (Reason + Act)

O que é?

O ReAct é uma técnica de engenharia de prompt que significa Reason (pensar) mais Act (agir). Foi criada para permitir que os LLMs interajam com ferramentas externas para recuperar informações

adicionais, levando a respostas mais confiáveis e factuais. Ele é ótimo para modelos mais simples que não necessriamente passaram por um processo de fine tuning para utilização de ferramentas. Sugerimos que para modelos open source, o agent ReAct seja usado:

Exemplo Prático

Vamos criar um ReAct Agent. Primeiro, importamos a função necessária:

```
from langchain.agents import create_react_agent
```

Podemos utilizar o LangChain Hub para obter prompts prontos para React:

```
from langchain import hub

prompt = hub.pull('hwchase17/react')
prompt
```

Vamos dar uma olhada no prompt:

```
print(prompt.template)
```

```
Answer the following questions as best you can. You have access to the following tools:

{tools}

Use the following format:

Question: the input question you must answer
Thought: you should always think about what to do
Action: the action to take, should be one of [{tool_names}]
Action Input: the input to the action
Observation: the result of the action
... (this Thought/Action/Action Input/Observation can repeat N times)
Thought: I now know the final answer
Final Answer: the final answer to the original input question

Begin!

Question: {input}
Thought:{agent_scratchpad}
```

Agora, vamos criar o Agent:

```
chat = ChatOpenAI()
agent = create_react_agent(chat, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
```

Vamos rodar o Agent e fazer uma pergunta:

```
agent_executor.invoke({'input': 'Qual é o décimo valor da sequência Fibonacci?'})
```

```
{'input': 'Qual é o décimo valor da sequência Fibonacci?',
 'output': 'The tenth value of the Fibonacci sequence is 55.'}
```

Neste caso, o modelo até foi capaz de chegar em uma solução. Mas ao observamos a execução, verificamos que por diversas vezes ele tentou utilizar a ferramenta de python mas não conseguiu. Vamos melhorar o prompt para aumentar as chances de utilizarmos as ferramentas eficazmente.

Refinando o Prompt

Vamos criar um prompt mais refinado para o React Agent:

```
from langchain.prompts import PromptTemplate

prompt = PromptTemplate.from_template(
'''Answer the following questions as best you can. You have access to the following tools:

{tools}

Use the following format:

Question: the input question you must answer
Thought: you should always think about what to do
Action: the action to take, should be one of {tool_names}
Action Input: the input to the action
Observation: the result of the action
... (this Thought/Action/Action Input/Observation can repeat N times)
Thought: I now know the final answer
Final Answer: the final answer to the original input question

Begin!

Question: {input}
Thought:{agent_scratchpad}'''
)

chat = ChatOpenAI()
agent = create_react_agent(chat, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
```

Vamos rodar novamente:

```
agent_executor.invoke({'input': 'Qual é o décimo valor da sequência Fibonacci?'})
```

```
{'input': 'Qual é o décimo valor da sequência Fibonacci?',
 'output': 'The tenth value of the Fibonacci sequence is 55.'}
```

Quando utilizar Tool Calling e quando utilizar ReAct

Sugerimos que utilizem o ReAct agent para modelos mais simples que não passaram por processo de fine tuning para utilização de funções externa. Em geral, esse é o caso quando você tentar rodar um

modelo menor localemnte. Ao utilizar modelos maiores, dos principais provedores de modelos de linguagem como OpenAI, Google ou Anthropic, sugerimos que utilize o Tool Calling.

Hoje aprendemos sobre dois tipos importantes de Agents: Tool Calling e ReAct. Vimos como criar e utilizar cada um deles, além de entender suas aplicações e limitações. Agora, você está mais preparado para escolher e implementar o tipo de Agent que melhor se adapta às suas necessidades. Continue praticando e explorando as possibilidades que esses Agents oferecem!

14. Agents Toolkits - Criando Agents para Analisar Dataframes e SQL

Bem-vindos ao nosso último capítulo da apostila de Agents de IA com Python e LangChain. Vamos explorar estruturas muito poderosas do framework: os Agents Toolkits. Eles nos permitirão criar muito facilmente agents que podem analisar DataFrames e bases de dados SQL. Nossos objetivos serão:

- Entender o que são Toolkits e como eles são utilizados no LangChain.
- Aprender a criar um agent para analisar DataFrames utilizando Pandas.
- Aprender a criar um agent para manipular e consultar bases de dados SQL.

O que são os Tollkits?

Os Toolkits são conjuntos de ferramentas que, quando combinadas, formam uma aplicação poderosa. No LangChain, já existem alguns Toolkits prontos que facilitam a criação de agents para tarefas específicas. Hoje, vamos focar em dois Toolkits principais: Pandas DataFrame e SQL Database. Esses Toolkits são extremamente úteis para quem trabalha com análise de dados, pois permitem automatizar e simplificar muitas tarefas.

Pandas DataFrame

Vamos começar com o Toolkit de Pandas DataFrame. Este Toolkit permite que um agent analise um DataFrame, facilitando a manipulação e a extração de informações.

Exemplo Prático Primeiro, vamos importar as bibliotecas necessárias e carregar um DataFrame de exemplo:

```
from langchain_experimental.agents.agent_toolkits import create_pandas_dataframe_agent
from langchain_openai import ChatOpenAI
import pandas as pd

df = pd.read_csv(
    "https://raw.githubusercontent.com/pandas-dev/pandas/main/doc/data/titanic.csv"
)
df.head(5)
```

Isso nos dá uma visão inicial do DataFrame:

```
   PassengerId  Survived  Pclass  \
0            1         0       3
1            2         1       1
2            3         1       3
3            4         1       1
4            5         0       3
```

```
                                                Name     Sex   Age  SibSp  \
0                            Braund, Mr. Owen Harris    male  22.0      1
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0      1
2                             Heikkinen, Miss. Laina  female  26.0      0
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0      1
4                           Allen, Mr. William Henry    male  35.0      0

   Parch            Ticket     Fare Cabin Embarked
0      0         A/5 21171   7.2500   NaN        S
1      0          PC 17599  71.2833   C85        C
2      0  STON/O2. 3101282   7.9250   NaN        S
3      0            113803  53.1000  C123        S
4      0            373450   8.0500   NaN        S
```

Agora, vamos criar o nosso agent:

```
chat = ChatOpenAI(model='gpt-3.5-turbo-0125')
agent = create_pandas_dataframe_agent(
    chat,
    df,
    verbose=True,
    agent_type='tool-calling',
    allow_dangerous_code=True
)
```

Podemos fazer algumas perguntas ao nosso agent para ver como ele se sai:

```
agent.invoke({'input': 'Quantas linhas tem na tabela?'})
```

```
> Entering new AgentExecutor chain...

Invoking: `python_repl_ast` with `{'query': 'len(df)'}`

891
A tabela possui 891 linhas.

> Finished chain.

{'input': 'Quantas linhas tem na tabela?',
 'output': 'A tabela possui 891 linhas.'}
```

Você pode perceber que o toolkit está utilizando a ferramenta `python_repl_ast` que nós já conhecemos para manipular os dados do nosso DataFrame.

```
agent.invoke({'input': 'Qual a média da idade dos passageiros?'})
```

```
> Entering new AgentExecutor chain...

Invoking: `python_repl_ast` with `{'query': "df['Age'].mean()"}`

29.69911764705882
A média da idade dos passageiros é de aproximadamente 29.70 anos.
```

```
> Finished chain.

{'input': 'Qual a média da idade dos passageiros?',
 'output': 'A média da idade dos passageiros é de aproximadamente 29.70 anos.'}

agent.invoke({'input': 'Quantos passageiros sobreviveram?'})

> Entering new AgentExecutor chain...

Invoking: `python_repl_ast` with `{'query': "df['Survived'].sum()"}`

342
Houve um total de 342 passageiros que sobreviveram ao acidente.

> Finished chain.

{'input': 'Quantos passageiros sobreviveram?',
 'output': 'Houve um total de 342 passageiros que sobreviveram ao acidente.'}
```

E é impressionante como ele consegue fazer análises que, por mais que sejam simples, seriam completamente impossíveis para quem desconhece análise de dados. Isso permite uma acessibilidade muito maior de pessoas a principal ferramenta de análise de dados, o Python!

SQL Database

Agora, vamos ver como criar um agent para manipular e consultar uma base de dados SQL.

Exemplo Prático Primeiro, vamos importar as bibliotecas necessárias e carregar uma base de dados de exemplo:

```
from langchain_community.utilities.sql_database import SQLDatabase

db = SQLDatabase.from_uri('sqlite:///arquivos/Chinook.db')
```

Vamos criar o nosso agent para SQL:

```
from langchain_community.agent_toolkits.sql.base import create_sql_agent
from langchain_openai import ChatOpenAI

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

agent_executor = create_sql_agent(
    chat,
    db=db,
    agent_type='tool-calling',
    verbose=True
)
```

Podemos listar as tools disponíveis no agent:

```
for tool in agent_executor.tools:
    print(tool.name)
    print(tool.description)
    print()
```

```
sql_db_query

Input to this tool is a detailed and correct SQL query, output is a result from the database.
↪  If the query is not correct, an error message will be returned. If an error is returned,
↪  rewrite the query, check the query, and try again. If you encounter an issue with Unknown
↪  column 'xxxx' in 'field list', use sql_db_schema to query the correct table fields.

sql_db_schema

Input to this tool is a comma-separated list of tables, output is the schema and sample rows
↪  for those tables. Be sure that the tables actually exist by calling sql_db_list_tables
↪  first! Example Input: table1, table2, table3

sql_db_list_tables

Input is an empty string, output is a comma-separated list of tables in the database.

sql_db_query_checker

Use this tool to double check if your query is correct before executing it. Always use this
↪  tool before executing a query with sql_db_query!
```

Vemos que ele possui as seguintes ferramentas acopladas:

- `sql_db_query`: para realizar uma query na base de dados
- `sql_db_schema`: para entender a estrutura da base de dados
- `sql_db_list_tables`: para listar as tabelas da base de dados
- `sql_db_query_checker`: para verificar se a query gerada pelo **sql_db_query** está correta.

Vamos fazer uma consulta para descrever a base de dados:

```
agent_executor.invoke({'input': 'Me descreva a base de dados'})
```

```
> Entering new SQL Agent Executor chain...

Invoking: `sql_db_list_tables` with `{}`

Album, Artist, Customer, Employee, Genre, Invoice, InvoiceLine, MediaType, Playlist,
↪  PlaylistTrack, Track

Invoking: `sql_db_schema` with `{'table_names': 'Album, Artist, Customer, Employee, Genre,
↪  Invoice, InvoiceLine, MediaType, Playlist, PlaylistTrack, Track'}`

...
```

```
agent_executor.invoke({'input': 'Qual artista possui mais albuns?'})
```

```
...
```

```
Invoking: `sql_db_query` with `{'query': 'SELECT Artist.Name, COUNT(Album.AlbumId) AS
↪  Number_of_Albums FROM Artist JOIN Album ON Artist.ArtistId = Album.ArtistId GROUP BY
↪  Artist.ArtistId ORDER BY Number_of_Albums DESC LIMIT 1'}`


[('Iron Maiden', 21)]O artista que possui mais álbuns é "Iron Maiden", com um total de 21
↪  álbuns.

> Finished chain.
{'input': 'Qual artista possui mais albuns?',
 'output': 'O artista que possui mais álbuns é "Iron Maiden", com um total de 21 álbuns.'}
```

Podemos ver que o agente é capaz de realizar queries complexas a base de dados, e tudo de forma muito simples!

Neste capítulo, aprendemos a criar agents utilizando Toolkits para analisar DataFrames e bases de dados SQL. Vimos como esses Toolkits podem simplificar e automatizar tarefas complexas de análise de dados. Esperamos que vocês tenham gostado e que essas ferramentas sejam úteis no dia a dia de vocês.

15. Finalizando o curso

E assim encerramos o nosso curso. Muito obrigado, meus amigos, por terem ficado até aqui. Parabéns a vocês pela determinação em percorrer essa jornada, que eu sei que não foi fácil. Mas vocês sabem que quanto mais difícil a jornada, mais recompensador é chegar ao final. Então, parabéns!

Gostaria de pedir um favor a vocês: deixem nos comentários da aula, o que vocês acharam que faltou no curso, o que gostariam de ver em termos de aplicações, como vocês estão pensando em utilizar os seus agentes, quais dúvidas têm em geral sobre agentes e como utilizá-los. Isso nos ajudará a melhorar nosso curso, desenvolver mais aulas e criar projetos conforme as necessidades de vocês.

Queremos que cada vez mais vocês deem feedback, retornem para nós o que estão pensando e como estão utilizando as IAs, para que possamos customizar bem o nosso conteúdo conforme as suas necessidades. Então, fica aqui esse pedido para vocês.

Para consolidar bem o que aprendemos, fica sempre aquela dica: apliquem, tentem, exercitem, porque senão vocês vão acabar esquecendo tudo. Claro, vocês sempre podem recorrer novamente ao nosso curso e aos nossos materiais para reaprender, mas o melhor mesmo para consolidar é aplicar e tentar criar agentes para as suas aplicações. Assim, tenho certeza de que tudo o que fizemos será consolidado.

Muito obrigado novamente e nos vemos nos nossos próximos cursos e projetos aqui na Asimov. Grande abraço!