

Desenvolvimento de Data-Driven Apps com Python

Teste de Performance 2 (TP2)

Instituto Infnet - Rafael Dottori de Oliveira

21/11/2024

Enunciado

Este Teste de Performance (TP) é composto por 10 questões, nas quais você deve demonstrar habilidades práticas no desenvolvimento de aplicações simples utilizando FastAPI ou LangChain com modelos de linguagem (LLMs).

- **Orientações:**

O código deve ser disponibilizado no GitHub, seguindo boas práticas de projeto.

Em questões relacionadas à arquitetura, os alunos devem fornecer diagramas que representem a estrutura da aplicação.

Nas questões de comparação, as respostas devem ser organizadas em formato de tabela.

- **Critérios de Avaliação:**

Organização e clareza do código.

Uso de boas práticas de desenvolvimento.

Implementação correta e funcional dos aplicativos.

Explicações concisas com diagramas ou tabelas conforme solicitado.

- **Entrega Final:**

Todo o código deve ser submetido no GitHub, organizado em repositórios separados para cada questão prática 1-2 (Parte 1) e 1-2-3 (Parte 2).

Incluir um arquivo README.md com instruções detalhadas sobre como executar cada aplicação.

As explicações e diagramas para as questões 3-4 (Parte 1) e 4-5-6 (Parte 2) devem ser enviadas em um arquivo PDF, juntamente com os diagramas e tabelas solicitados.

Link para o Github:

<https://github.com/R-Dottori/dev-apps-TP2>

Parte 1: FastAPI

Questão 1

Crie uma aplicação simples em FastAPI que utilize o modelo GPT-2 da HuggingFace para gerar textos a partir de uma entrada fornecida via requisição HTTP.

- *O aplicativo deve:*

Receber uma frase de entrada como JSON.

Utilizar a biblioteca transformers do HuggingFace para gerar um texto de saída.

Retornar o texto gerado em uma resposta HTTP.

- *O que é esperado:*

O aplicativo deve gerar uma continuação de texto a partir de uma frase de entrada e retornar a resposta formatada como JSON.

Vamos organizar os códigos das aplicações em três arquivos .py separados: o com os formatos dos dados enviados utilizando BaseModel, o com as rotas das páginas adicionais da API (por exemplo, localhost:8000/pagina_extra) e um arquivo principal para rodar a API e apontar para as rotas criadas.

Desse jeito, é mais fácil adicionar e manter novas rotas e modelos.

Normalmente escreveríamos o conteúdo diretamente nos arquivos .py. Ao longo do trabalho utilizaremos comandos de *cell magic* para poder exibir no próprio *notebook* o conteúdo desses arquivos.

```
In [35]: %%writefile ./app_fastapi/models.py
```

```
from pydantic import BaseModel

class ModeloTexto(BaseModel):
    texto: str
```

Overwriting ./app_fastapi/models.py

```
In [36]: %%writefile ./app_fastapi/routers.py
```

```
from fastapi import APIRouter
from .models import ModeloTexto
from transformers import pipeline

router = APIRouter()
```

```
@router.post('/gerar_texto/')
async def gerar_texto(body: ModeloTexto):
    modelo = pipeline('text-generation', model='gpt2')
    resposta = modelo(body.texto)
    return {'resposta': resposta}
```

Overwriting ./app_fastapi/routers.py

```
In [37]: %%writefile ./app_fastapi/main.py

from fastapi import FastAPI
from .routers import router

app = FastAPI()

app.include_router(router, prefix='/chat')

@app.get('/')
async def raiz():
    return {'mensagem': 'Página raiz'}
```

Overwriting ./app_fastapi/main.py

Para testarmos a aplicação, utilizamos o uvicorn via terminal:

```
uvicorn app_fastapi.main:app
```

E com o httpie, podemos testar os métodos de HTTP, como acessar a página raiz da aplicação:

```
http GET localhost:8000/
```

Abaixo, testaremos o modelo GPT-2 do HuggingFace com uma pergunta qualquer

```
http POST localhost:8000/chat/gerar_texto/ texto='How are you?'
```

```
[{"generated_text": "How are you? The first thing you notice is that your body's expression changes with different body types. You don't grow taller, and your mouth doesn't grow longer. Your facial hair grows longer. You don't sweat more. The facial hair"}]
```

Questão 2

Crie um aplicativo FastAPI que utiliza o modelo de tradução Helsinki-NLP/opus-mt-en-fr da HuggingFace para traduzir textos do inglês para o francês.

- A aplicação deve:

Receber um texto em inglês via uma requisição HTTP.

Traduzir o texto para o francês utilizando o modelo de tradução.

Retornar o texto traduzido em uma resposta JSON.

- O que é esperado:

A API deve receber um texto em inglês e retornar sua tradução para o francês, processando tanto frases curtas quanto textos mais longos.

Como explicamos anteriormente, podemos melhorar a aplicação sem mexer no arquivo main.py.

Vamos adicionar mais um modelo e uma nova rota para gerar as traduções.

```
In [38]: %%writefile ./app_fastapi/models.py
```

```
from pydantic import BaseModel

class ModeloTexto(BaseModel):
    texto: str

class ModeloTraducao(BaseModel):
    texto: str
```

Overwriting ./app_fastapi/models.py

```
In [39]: %%writefile ./app_fastapi/routers.py
```

```
from fastapi import APIRouter
from .models import ModeloTexto, ModeloTraducao
from transformers import pipeline

router = APIRouter()

@router.post('/gerar_texto/')
async def gerar_texto(body: ModeloTexto):
    modelo = pipeline('text-generation', model='gpt2')
    resposta = modelo(body.texto)
    return {'resposta': resposta}

@router.post('/traduzir/')
async def traduzir_texto(body: ModeloTraducao):
    modelo = pipeline('translation_en_to_fr', model='Helsinki-NLP/opus-mt-en-fr')
    resposta = modelo(body.texto)
    return {'resposta': resposta}
```

Overwriting ./app_fastapi/routers.py

Rodamos novamente nosso aplicativo via uvicorn e podemos usar o método POST para testarmos as traduções.

```
http POST localhost:8000/chat/traduzir/ texto='Translate this text, please.'
```

```
HTTP/1.1 200 OK content-length: 61 content-type: application/json date: Tue, 12 Nov
2024 20:53:59 GMT server: uvicorn
```

```
[ { "translation_text": "Traduire ce texte, s'il vous plaît." } ]
```

```
http POST localhost:8000/chat/traduzir/ texto="Today's the debut of our  
new play. I've told everyone to break a leg!"
```

```
HTTP/1.1 200 OK content-length: 139 content-type: application/json date: Wed, 13 Nov  
2024 17:29:29 GMT server: uvicorn
```

```
{ "resposta": [ { "translation_text": "Aujourd'hui c'est le début de notre nouvelle pièce. J'ai  
dit à tout le monde de casser une jambe !" } ] }
```

Questão 3

Com base na API desenvolvida na Questão 2 (Parte1), explique as principais limitações do modelo de tradução utilizado.

- *Enumere e discuta:*

Limitações quanto à precisão da tradução.

Desafios de tempo de resposta e desempenho em grande escala.

Restrições de custo e escalabilidade.

Limitações na tradução de gírias, expressões idiomáticas ou linguagem de contexto.

A aplicação criada demonstrou algumas limitações, principalmente no tempo de resposta. Os testes foram feitos pelo terminal do Visual Studio Code, utilizando as bibliotecas uvicorn e httpie.

A princípio, as traduções geradas do inglês para o francês foram satisfatórias, incluindo palavras que possam ter mais de um significado ("play"). Ao inserir a expressão idiomática "break a leg", o tradutor a traduziu literalmente.

Se pensássemos nessa mesma aplicação em um volume maior de dados, teríamos ainda mais problemas com as traduções, com o custo do treinamento, tempo de respostas, etc.

Questão 4

Com base no modelo GPT-2 utilizado na Questão 1 (Parte 1), explique as principais limitações do modelo no contexto da geração de texto.

- *Discuta:*

A coerência do texto gerado.

Possíveis falhas ou incoerências geradas por LLMs.

Desempenho e questões de latência.

Limitações na geração de conteúdo apropriado.

No próprio exemplo que visualizamos no primeiro exercício já conseguimos apontar limitações do modelo:

How are you? The first thing you notice is that your body's expression changes with different body types. You don't grow taller, and your mouth doesn't grow longer. Your facial hair grows longer. You don't sweat more. The facial hair

Provavelmente por conta de um treinamento limitado, o modelo tem dificuldade de entender contextos e começa a mudar de assunto ao longo da resposta. Além disso, a resposta nem segue um sentido lógico da nossa realidade (alucinações de uma LLM).

Essa falta de contexto também poderia levar a respostas que não seguem o formato ou tom pedido — por exemplo, responder uma pergunta com outra pergunta, responder uma pergunta séria com uma piada, etc.

Para resolver tais problemas, esbarramos também em limitações técnicas ou de custo. Seria necessário treinar o modelo com uma base muito maior de dados e com ajustes e parâmetros que fizessem o modelo entender contextos específicos.

Parte 2: LangChain

Questão 1

Desenvolva um protótipo utilizando LangChain que simule um chatbot simples com Fake LLM.

- *A aplicação deve:*

Receber um input de texto via FastAPI.

Retornar uma resposta simulada pelo Fake LLM.

- *O que é esperado:*

O protótipo deve simular um chatbot básico que responde a perguntas pré-definidas.

A arquitetura deve ser simples, e você deve explicar a importância de usar Fake LLM para testes rápidos.

Desenhe um diagrama simples da arquitetura do aplicativo, detalhando as principais etapas do fluxo de dados.

Vamos montar uma nova aplicação seguindo a estrutura da anterior, com um arquivo para os modelos, um para as rotas e outro principal para rodarmos a aplicação.

In [40]: `%%writefile ./app_langchain/models.py`

```
from pydantic import BaseModel

class ModeloPergunta(BaseModel):
    pergunta: str
```

Overwriting ./app_langchain/models.py

In []: `%%writefile ./app_langchain/routers.py`

```
from fastapi import APIRouter
from .models import ModeloPergunta
from langchain_community.llms import FakeListLLM

router = APIRouter()

# Instanciamos o modelo fora da função correspondente pois foi a forma de fazê-lo
modelo_falso = FakeListLLM(responses=[
    'Olá, eu sou um robô!',
    'Estou bem, e você?',
    'Não sei te dizer...',
    'Essa é a última resposta da lista.'
])

@router.post('/conversar/')
async def conversa_falsa(body: ModeloPergunta):
    return {'resposta': modelo_falso.invoke(body.pergunta)}
```

Overwriting ./app_langchain/routers.py

In [42]: `%%writefile ./app_langchain/main.py`

```
from fastapi import FastAPI
from .routers import router

app = FastAPI()

app.include_router(router)

@app.get('/')
async def raiz():
    return {'mensagem': 'Página raiz'}
```

Overwriting ./app_langchain/main.py

Podemos fazer requisições POST seguidas para observarmos o Fake LLM utilizando as respostas falsas em sequência.

`http POST localhost:8000/conversar/ pergunta='Olá'`

HTTP/1.1 200 OK content-length: 34 date: Wed, 13 Nov 2024 19:29:33 GMT server: uvicorn

`{"resposta": "Olá, eu sou um robô!" }`

```
http POST localhost:8000/conversar/ pergunta='Olá'
```

```
HTTP/1.1 200 OK content-length: 34 date: Wed, 13 Nov 2024 19:29:37 GMT server:
unicorn
```

```
{ "resposta": "Estou bem, e você?" }
```


```
http POST localhost:8000/conversar/ pergunta='Uma pergunta qualquer'
```

```
HTTP/1.1 200 OK content-length: 35 content-type: application/json date: Wed, 13 Nov
2024 19:29:46 GMT server: unicorn
```

```
{ "resposta": "Não sei te dizer..." }
```

Ao utilizarmos Fake LLM para os testes, temos respostas obtidas de maneira rápida — já que o modelo não está gerando uma resposta de verdade — e sem os gastos por resposta que alguns modelos cobram (como os da OpenAI).

Abaixo temos um diagrama da arquitetura do aplicativo:

arquitetura

Questão 2

Desenvolva um aplicativo que utilize LangChain para integrar a API da OpenAI.

- O aplicativo deve:

Receber um texto em inglês via FastAPI.

Traduzir o texto para o francês utilizando um modelo da OpenAI via LangChain.

Retornar o texto traduzido em uma resposta JSON.

- O que é esperado:

O aplicativo deve funcionar como uma API de tradução, semelhante à questão 2 (Parte 1), mas utilizando a OpenAI via LangChain.

A aplicação deve gerenciar as chamadas à API da OpenAI e retornar a tradução com baixa latência.

Forneça um diagrama da arquitetura da aplicação, destacando os componentes principais, como FastAPI, LangChain, e OpenAI.

Não foi possível utilizar nenhuma versão gratuita da API da OpenAI. Portanto, nos exercícios que a pedem, utilizei a API do Google para rodar o modelo Gemini em seu lugar.

```
In [ ]: %%writefile ./app_langchain/models.py
```



```
from pydantic import BaseModel

class ModeloPergunta(BaseModel):
    pergunta: str

class ModeloTraducao(BaseModel):
    texto: str
```

Overwriting ./app_langchain/models.py

In [1]: `%%writefile ./app_langchain/routers.py`

```
from fastapi import APIRouter
from .models import ModeloPergunta, ModeloTraducao
from langchain_community.llms import FakeListLLM
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.prompts import ChatPromptTemplate
from dotenv import load_dotenv
import os

router = APIRouter()

load_dotenv()

# Instanciamos o modelo fora da função correspondente pois foi a forma de fazê-lo
modelo_falso = FakeListLLM(responses=[
    'Olá, eu sou um robô!',
    'Estou bem, e você?',
    'Não sei te dizer...',
    'Essa é a última resposta da lista.'
])

@router.post('/conversar/')
async def conversa_falsa(body: ModeloPergunta):
    return {'resposta': modelo_falso.invoke(body.pergunta)}

@router.post('/traduzir/')
async def traduzir_texto(body: ModeloTraducao):
    modelo_traducao = ChatGoogleGenerativeAI(model='gemini-1.5-flash', api_key=os.getenv('GEMINI_API_KEY'))
    resposta = ChatPromptTemplate([
        ('system', 'Translate the english texts to french.'),
        ('user', 'Translate the following text: {texto}')
    ])

    return {'resposta': modelo_traducao.invoke(resposta.format_messages(texto=body.texto))}
```

Overwriting ./app_langchain/routers.py

http POST localhost:8000/traduzir/ texto='Please, translate this text.'

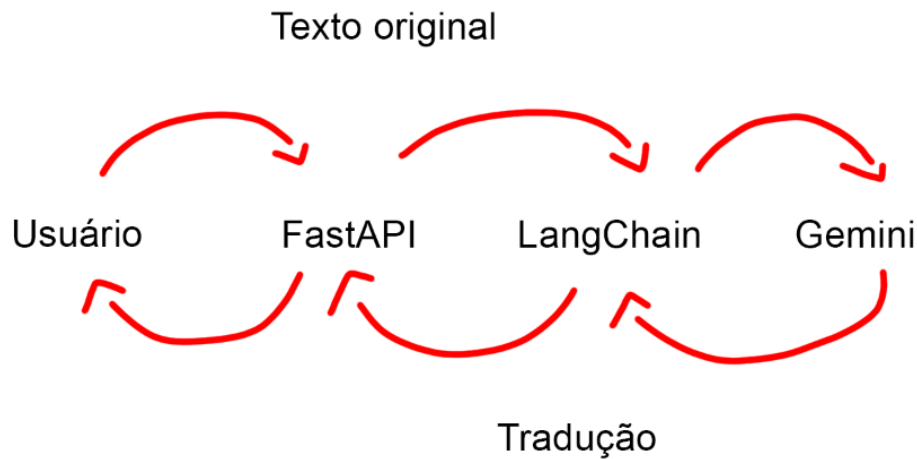
HTTP/1.1 200 OK content-length: 45 content-type: application/json date: Wed, 13 Nov 2024 20:37:40 GMT server: uvicorn

```
{ "resposta": "Veuillez traduire ce texte. \n" }
```

```
http POST localhost:8000/traduzir/ texto='Today is wednesday. Is such a nice day outside.'
```

```
HTTP/1.1 200 OK content-length: 77 content-type: application/json date: Wed, 13 Nov 2024 20:38:40 GMT server: uvicorn
```

```
{ "resposta": "Aujourd'hui, c'est mercredi. Il fait tellement beau dehors. \n" }
```



Questão 3

Crie uma API semelhante à Questão 2 (Parte 2), mas que utilize o modelo Helsinki-NLP/opus-mt-en-de da HuggingFace para traduzir textos do inglês para o alemão.

- A aplicação deve:

Receber um texto em inglês via FastAPI.

Utilizar o LangChain para gerenciar as chamadas ao modelo HuggingFace.

Retornar o texto traduzido para o alemão como resposta JSON.

- O que é esperado:

O objetivo é que a aplicação funcione de maneira semelhante às Questões 2 (Parte 1) e 2 (Parte 2), mas desta vez integrando LangChain com HuggingFace.

O modelo a ser utilizado deve ser o Helsinki-NLP/opus-mt-en-de.

Forneça um diagrama detalhado da arquitetura da aplicação, destacando as interações entre FastAPI, LangChain, e HuggingFace.

```
In [25]: %%writefile ./app_langchain/routers.py

from fastapi import APIRouter
from .models import ModeloPergunta, ModeloTraducao
```

```

from langchain_community.llms import FakeListLLM
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_huggingface.llms import HuggingFacePipeline
from dotenv import load_dotenv
import os

router = APIRouter()

load_dotenv()

# Instanciamos o modelo fora da função correspondente pois foi a forma de fazê-lo
modelo_falso = FakeListLLM(responses=[
    'Olá, eu sou um robô!',
    'Estou bem, e você?',
    'Não sei te dizer...',
    'Essa é a última resposta da lista.'
])

@router.post('/conversar/')
async def conversa_falsa(body: ModeloPergunta):
    return {'resposta': modelo_falso.invoke(body.pergunta)}

@router.post('/traduzir/frances/')
async def traduzir_gemini(body: ModeloTraducao):
    modelo_traducao_gemini = ChatGoogleGenerativeAI(model='gemini-1.5-flash', api_key=API_KEY)
    resposta = ChatPromptTemplate([
        ('system', 'Translate the english texts to french.'),
        ('user', 'Translate the following text: {texto}')
    ])
    return {'resposta': modelo_traducao_gemini.invoke(resposta.format_messages(text=texto))}

@router.post('/traduzir/alemao/')
async def traduzir_huggingface(body: ModeloTraducao):
    modelo_traducao_huggingface = HuggingFacePipeline.from_model_id(model_id='HuggingFaceH4/zephyr-7b-beta', token=API_KEY)
    return {'resposta': modelo_traducao_huggingface.invoke(body.texto)}

```

Overwriting ./app_langchain/routers.py

```

http POST localhost:8000/traduzir/alemao/ texto='Please, translate this
text.'

```

HTTP/1.1 200 OK content-length: 49 content-type: application/json server: uvicorn

```
{ "resposta": "Bitte übersetzen Sie diesen Text." }
```

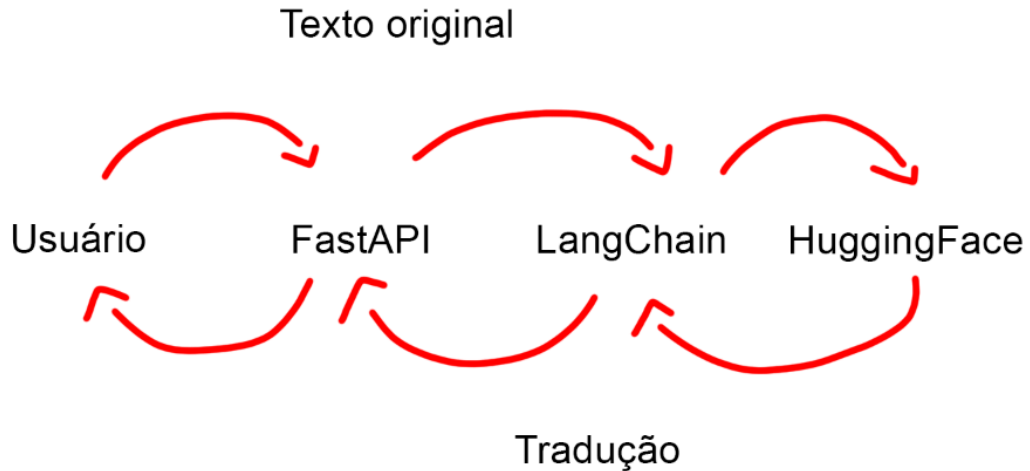
```

http POST localhost:8000/traduzir/alemao/ texto='Today is wednesday.
What a beautiful day!'

```

HTTP/1.1 200 OK content-length: 61 content-type: application/json date: Wed, 13 Nov 2024 21:11:14 GMT server: uvicorn

```
{ "resposta": "Heute ist Mittwoch. Was für ein schöner Tag!" }
```



Questão 4

Com base na implementação da Questão 2 (Parte 2), explique as principais limitações de utilizar LangChain para integrar a API da OpenAI.

- Discuta os seguintes aspectos:

Latência de resposta.

Limites de uso da API da OpenAI.

Desafios de escalabilidade e custo.

Qualidade das traduções geradas em comparação com outros modelos.

Em relação as respostas obtidas na primeira parte do TP, a aplicação utilizando LangChain teve respostas muito mais demoradas (tanto com o Gemini quanto o modelo via HuggingFace).

Como comentado anteriormente, os limites de custo da API da OpenAI também foram um problema. Não foi possível realizar nenhuma consulta gratuita e, portanto, optamos por utilizar o Gemini.

Sobre as traduções em si: os resultados foram muito parecidos. Traduções boas de maneira geral, com dificuldades em contextos bem específicos como traduzir algumas expressões idiomáticas.

Questão 5

Com base na aplicação desenvolvida na 3 (Parte 2), explique as limitações de usar LangChain para integrar o modelo HuggingFace de tradução.

• Discuta aspectos como:

Desempenho e tempo de resposta.

Consumo de recursos computacionais.

Possíveis limitações no ajuste fino do modelo.

Comparação com o uso direto da API HuggingFace.

De forma parecida com o aplicativo utilizando Gemini, foi possível observar um tempo de resposta maior ao montar aplicações com o LangChain.

Quanto as respostas em si, mesmo se tratando de dois idiomas diferentes (o uso direto do HuggingFace foi de inglês para francês e aqui traduzimos de inglês para alemão) tivemos traduções razoáveis.

Abaixo montaremos uma tabela comparando diretamente as duas abordagens.

Questão 6

Com base nas questões 1-2 (Parte 1) e 2-3 (Parte 2), desenvolva uma tabela comparativa que aborde os seguintes critérios:

Facilidade de uso/configuração.

Latência e desempenho.

Flexibilidade para diferentes modelos.

Custo e escalabilidade.

Adequação para protótipos versus aplicações em produção.

A comparação deve ser apresentada em formato de tabela, com colunas dedicadas a cada critério e linhas comparando FastAPI puro com LangChain.

	FastAPI Puro	LangChain
Facilidade de uso	Método de pipeline direto, mais fácil de entender os parâmetros e documentação	Documentação mais complexa, precisando passar o ID do modelo
Latência e desempenho	Respostas mais rápidas	Respostas mais demoradas
Flexibilidade para diferentes modelos	Os modelos utilizados costumam resolver apenas um tipo específico de tarefa (como tradução)	Suporte a fluxos mais complexos dos modelos
Custo e escalabilidade	Custo inicial mais baixo, porém menos escalável	Mais custoso e pesado por exigir recursos do LangChain, porém mais escalável
Adequação para protótipos versus aplicações em produção	Testes mais demorados que dependem da resposta gerada pelo modelo	Melhor para testes utilizando ferramentas como FakeLLM

Link para o Github:

<https://github.com/R-Dottori/dev-apps-TP2>