Lab. 08 (Parte 2): Colocando o Modelo em Produção com FastAPI e Docker

Aléssio Miranda Júnior

Setembro - 2026/2

Lab. 08 (Parte 2): Deploy com FastAPI e Docker

⋆ Objetivo da Sessão 2

No ecossistema corporativo de engenharia de software, um modelo de Machine Learning que permanece restrito a um notebook ou a um script isolado possui **impacto operacional nulo**. O valor de negócio se concretiza quando o modelo opera como um microsserviço resiliente, validado e acessível via rede por outros sistemas corporativos em poucos milissegundos.

Dinâmica Operacional da Sessão de Hoje:

- **Alunos com a Parte 1 em andamento:** Utilizem a primeira parte desta sessão para concluir o treinamento do classificador desbalanceado, gerar a matriz de confusão, serializar o modelo (`.keras`) e o escalonador (`.joblib`), assegurando a aprovação nos testes locais e no pipeline do GitLab CI (`pytest tests/test_avaliacao.py`).
- **Alunos com a Parte 1 concluída (ou assim que concluírem):** Iniciem imediatamente a Parte 2. Vocês transformarão os artefatos de IA em uma **API REST assíncrona** usando **FastAPI** e **Pydantic**, testarão a interface interativa com o **Swagger UI** e encapsularão o serviço em um container isolado com **Docker**.

1 Parte 1: A Arquitetura de Inferência em Produção (10 min)

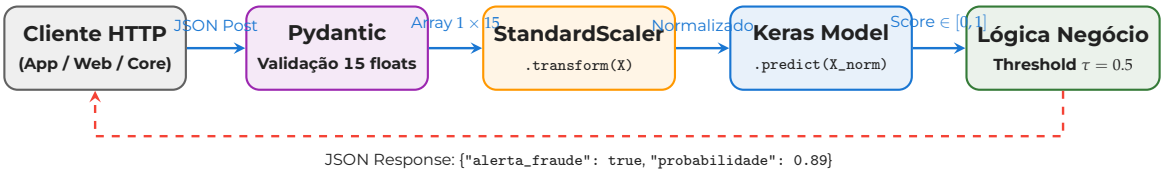
Na Sessão 1, você gerou dois artefatos fundamentais:

1. `modelo_fraude.keras`: Contém a topologia da rede neural e os tensores de pesos (W e b) otimizados.

2 `scaler_fraude.joblib`: Contém a média (μ) e o desvio padrão (σ) de cada uma das 15 características calculadas no conjunto de treino.

△ Importante: Atenção de Engenharia: A Armadilha do Escalonador Esquecido

Uma falha recorrente em pipelines amadores de deploy é expor o modelo diretamente sem o escalonador. Lembre-se: o modelo foi treinado em dados z-score ($\mu \approx 0, \sigma \approx 1$). Se o cliente HTTP enviar um payload com o valor de transação bruto (ex: $R\$\,1.500,00$), a rede neural receberá uma magnitude centenas de vezes superior à faixa de ativação de treino, resultando em previsões erráticas. **O escalonador faz parte indissociável da inferência.**



2 Parte 2: Construindo a API REST com FastAPI (15 min)

O framework **FastAPI** estabeleceu-se como referência para serviços de Inteligência Artificial devido à sua alta performance assíncrona (baseada em Starlette/ASGI), validação declarativa de dados com **Pydantic** e geração automática de especificações OpenAPI.

Crie o arquivo `src/api.py` no repositório do projeto:

```python
import os
import joblib
import numpy as np
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field
from tensorflow.keras.models import load_model

# 1. Instanciacao da aplicacao com metadados
app = FastAPI(
    title="NeoBank - Sentinela de Fraudes API",
    description="Microsservico de inferencia em tempo real para
        deteccao de transacoes fraudulentas.",
    version="1.0.0"
)

# 2. Variaveis globais de persistencia
MODEL_PATH = "modelo_fraude.keras"
SCALER_PATH = "scaler_fraude.joblib"

```

```python
modelo = None
scaler = None

# Carregamento sob demanda ou inicializacao
@app.on_event("startup")
def carregar_artefatos():
    global modelo, scaler
    if not os.path.exists(MODEL_PATH) or not os.path.exists(SCALER_PATH):
        raise RuntimeError(
            f"Artefatos nao encontrados! Execute o treinamento primeiro. "
            f"Arquivos necessarios: {MODEL_PATH} e {SCALER_PATH}"
        )
    modelo = load_model(MODEL_PATH)
    scaler = joblib.load(SCALER_PATH)
    print("Artefatos de inferencia (modelo e scaler) carregados com sucesso.")

# 3. Contratos de Dados (Schemas Pydantic)
class TransacaoInput(BaseModel):
    features: list[float] = Field(
        ...,
        min_items=15,
        max_items=15,
        description="Vetor com exatamente 15 caracteristicas numericas da transacao.",
        example=[1250.0, 2.5, 0.8, -1.2, 0.05, 3.1, -0.4, 1.8,
                 0.9, -0.2, 0.1, 0.4, -0.05, 0.12, -0.3]
    )

class PredicaoOutput(BaseModel):
    probabilidade: float
    classe: int
    alerta_fraude: bool
    status: str

# 4. Endpoints da API
@app.get("/health", tags=["Monitoramento"])
def health_check():
    """Endpoint de telemetria e sondagem de saude (Liveness/Readiness)."""
    return {
        "status": "healthy",
        "service": "neobank-sentinela",
        "modelo_carregado": modelo is not None,
        "scaler_carregado": scaler is not None
    }

@app.post("/predict", response_model=PredicaoOutput, tags=["Inferencia"])
def predict(transacao: TransacaoInput):
```

```
    """Executa a predicao de fraude para uma transacao recebida."""
    try:
        # Converter para array bidimensional (1 amostra, 15 colunas)
        X_raw = np.array(transacao.features).reshape(1, -1)

        # Aplicar a normalizacao com os parametros de treino
        X_norm = scaler.transform(X_raw)

        # Executar inferencia na rede neural
        score = float(modelo.predict(X_norm, verbose=0)[0][0])
        classe = 1 if score > 0.5 else 0

        return PredicaoOutput(
            probabilidade=round(score, 4),
            classe=classe,
            alerta_fraude=bool(classe == 1),
            status="BLOQUEADA" if classe == 1 else "APROVADA"
        )
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Falha de
            inferencia: {str(e)}")
```

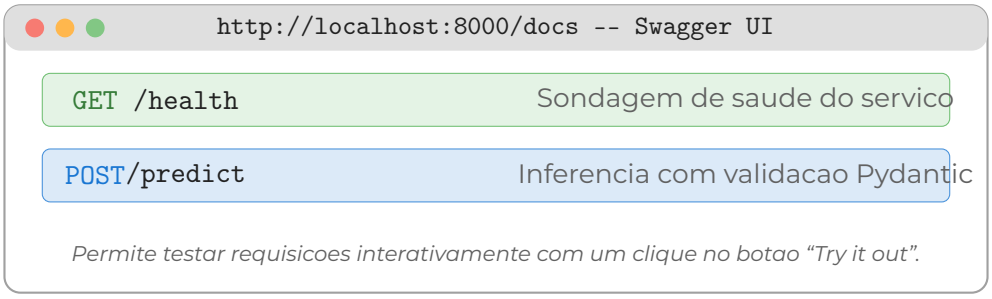
3 Parte 3: Execução Local e Validação com Swagger (10 min)

Para executar a API em ambiente de desenvolvimento, utilize o servidor ASGI de alta performance **Uvicorn**:

```
uvicorn src.api:app --host 0.0.0.0 --port 8000 --reload
```

Abra o navegador e acesse a URL:

`http://localhost:8000/docs`



Exercício de Verificação de Integridade:

1. Clique em **POST /predict** → **Try it out**.
2. Envie o payload padrão com 15 números e clique em **Execute**. Observe a resposta HTTP 200 com a chave `"probabilidade"`.

3 Agora teste a robustez do Pydantic: **remova um número** (enviando 14 características). Clique em **Execute**.

4 Verifique que a API respondeu imediatamente com **HTTP 422 (Unprocessable Entity)**, informando que o vetor é inválido, sem lançar exceções não tratadas e sem derrubar o servidor.

4 Parte 4: Conteinerização com Docker (10 min)

Na engenharia de produção, a aplicação deve ser independente do sistema operacional hospedeiro. Para garantir isolamento de dependências e portabilidade completa, criaremos o arquivo `Dockerfile` na raiz do projeto.

```
# 1. Imagem base oficial enxuta com Python 3.12
FROM python:3.12-slim

# 2. Definir variaveis de ambiente para otimizar execucao Python
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

# 3. Diretorio de trabalho da aplicacao
WORKDIR /app

# 4. Copiar arquivo de dependencias e instalar
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# 5. Copiar codigo-fonte e os artefatos de modelo treinados
COPY src/ ./src/
COPY modelo_fraude.keras .
COPY scaler_fraude.joblib .

# 6. Declarar porta de rede exposta
EXPOSE 8000

# 7. Comando de inicializacao do servidor de producao
CMD ["uvicorn", "src.api:app", "--host", "0.0.0.0", "--port", "8000"]
```

Crie também o arquivo `.dockerignore` para evitar cópia de arquivos temporários desnecessários para a imagem:

```
__pycache__/
*.pyc
.git
.venv
tests/
.pytest_cache
*.png
```

Certifique-se de que o arquivo `requirements.txt` contenha as dependências mínimas necessárias:

```
fastapi>=0.110.0
uvicorn>=0.28.0
pydantic>=2.6.0
tensorflow-cpu>=2.16.0
scikit-learn>=1.4.0
joblib>=1.3.0
numpy>=1.26.0
```

5 Parte 5: Construção e Execução do Container (10 min)

No terminal do seu ambiente local, execute a montagem da imagem Docker:

```
# Construir a imagem com a tag neobank-fraude:v1
docker build -t neobank-fraude:v1 .
```

Após a compilação das camadas, inicialize o container mapeando a porta 8000 do host para a porta 8000 interna do container:

```
# Inicializar o container em background (daemon)
docker run -d -p 8000:8000 --name sentinela-container neobank-fraude:
    v1
```

Verifique o status do container em execução e inspecione os logs:

```
docker ps
docker logs sentinela-container
```

Se a saída exibir a mensagem confirmando a inicialização do Uvicorn na porta 8000, sua IA está oficialmente operando em um ambiente isolado de produção.

6 Parte 6: Auditoria de Inferência em Produção (5 min)

Para validar que o serviço responde a requisições externas como esperado em um sistema distribuído, execute uma requisição utilizando o utilitário `curl` diretamente no terminal:

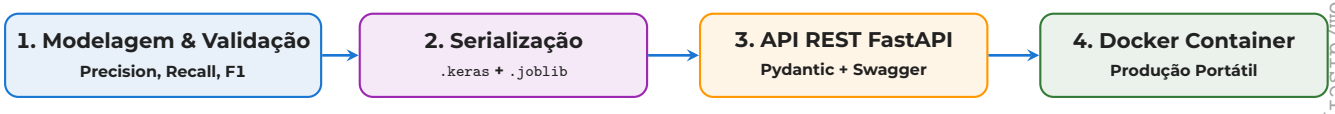
```
curl -X POST "http://localhost:8000/predict" \
     -H "Content-Type: application/json" \
     -d '{
       "features": [
         1250.0, 2.5, 0.8, -1.2, 0.05, 3.1, -0.4, 1.8,
         0.9, -0.2, 0.1, 0.4, -0.05, 0.12, -0.3
       ]
     }'
```

A resposta retornará no formato JSON estruturado:

```
{
  "probabilidade": 0.0412,
  "classe": 0,
  "alerta_fraude": false,
  "status": "APROVADA"
}
```

Conexão Estratégica: A Ponte para o TP1

Parabéns! Você acaba de construir o fluxo corporativo completo que conecta a Ciência de Dados à Engenharia de Software.



◇ **Informação**

O que você fará no Trabalho Prático 1 (TP1): No TP1, você e sua equipe desenvolverão uma aplicação completa de Inteligência Artificial — seja implementando a especificação oficial definida pelo professor (como o Classificador de Habilidades do ecossistema WOKDEX) ou selecionando um desafio real em plataformas abertas como o **Kaggle**, conforme a orientação da disciplina. O ciclo que você executou hoje (Labs 08 Parte 1 e Parte 2) serve como template arquitetural obrigatório:

- Coleta, limpeza e exploração do dataset real.
- Treinamento de uma arquitetura profunda com prevenção de sobreajuste (Dropout/EarlyStopping).
- Avaliação crítica através de métricas adequadas ao desbalanceamento do problema.
- Exposição da inferência através de uma API FastAPI documentada e empacotada em container Docker.

✔ Takeaways

- **Artefatos Complementares:** O modelo (`.keras`) armazena pesos e topologia, enquanto o escalonador (`.joblib`) garante que os dados em tempo de inferência recebam a mesma transformação estocástica do treino.
- **Contrato Estrito de Dados:** O Pydantic valida os tipos e a cardinalidade das entradas, rejeitando payloads malformados na fronteira da rede e protegendo o núcleo de IA contra erros de execução.
- **Independência de Ambiente:** O Docker elimina inconsistências entre sistemas operacionais, encapsulando código, runtime Python e dependências binárias em uma imagem imutável.
- **Fechamento do Módulo 1:** Você dominou a trilha que vai do cálculo manual do Perceptron solitário até o deploy conteinerizado de microsserviços de IA.