Documentação Técnica: Sistema de Classificação de Crédito em Tempo Real (MLOps)

1. Visão Geral e Objetivos do Projeto

Objetivo Geral

Desenvolver uma solução de Machine Learning de ponta-a-ponta (end-to-end) para classificação de risco de crédito. O objetivo é criar um algoritmo robusto que, ao ser exposto via API RESTI, seja capaz de receber instantaneamente os dados de um cliente de consórcio e retornar uma decisão binária de crédito (APROVADO ou NEGADO) com alta confiabilidade e baixa latência.

Objetivos Específicos

- **Garantir Reprodutibilidade**: Utilizar Poetry para gerenciamento de dependências e Makefile para orquestração de tarefas, assegurando ambientes de desenvolvimento e produção idênticos.
- **Qualidade de Código**: Adotar o Black Formatter e linters para manter um código limpo, padronizado e de fácil manutenção.
- **Implantação Ágil (MLOps)**: Containerizar o serviço via Docker Compose e implementar um *pipeline* de CI/CD para automatizar o *build*, teste e implantação da API.
- **Performance**: Selecionar um modelo de classificação com métricas de desempenho validadas (AUC-ROC e Recall) e expô-lo com uma latência de previsão inferior a 100ms.

---

2. Estrutura de Diretórios do Projeto

A organização de arquivos é crucial para um projeto MLOps. A estrutura recomendada é:

```
credit-scoring-project/
├── data/                 # Datasets brutos e processados
├── notebooks/            # Notebooks de EDA e experimentação
├── src/
│   ├── models/           # Modelo serializado (*.joblib) e pipelines
│   ├── preprocessing.py  # Lógica de transformação de features (Python script)
│   └── service/          # Código da API (backend)
│       ├── api.py        # Implementação da API (FastAPI)
```

```
│       └── schemas.py       # Esquemas de validação de dados (Pydantic)
├── tests/                  # Testes unitários e de integração (Pytest)
├── Dockerfile              # Definição do container Docker
├── pyproject.toml          # Gerenciamento de dependências (Poetry)
├── Makefile                # Orquestração de comandos
└── README.md
```

---

3. Etapas Detalhadas do Fluxo de Trabalho (End-to-End)

O projeto é dividido em três fases: Machine Learning, Backend e MLOps.

Fase I: Machine Learning e Preparação (ML)

O foco está na criação do ativo de ML e na garantia da reprodutibilidade das transformações.

1. **Aquisição e EDA (Análise Exploratória):**
    - Escolher e baixar um *dataset* de crédito do Kaggle.
    - Realizar a **Análise Exploratória de Dados (EDA)**, focando em entender a distribuição da variável alvo (aprovação/negação), identificação de dados faltantes (*missing values*) e *outliers*.
    - **Ferramentas: Spark** e **Matplotlib/Seaborn** nos *notebooks* dedicados.
2. **Pré-processamento e Feature Engineering:**
    - Criar um **Pipeline de Pré-processamento** (`sklearn.pipeline.Pipeline` e `ColumnTransformer`) para garantir que todas as etapas de limpeza e transformação (tratamento de nulos, *scaling* e *encoding*) sejam aplicadas de forma consistente.
    - **Melhor Prática:** A lógica de pré-processamento deve estar encapsulada em um *script* Python (`src/preprocessing.py`) e não depender apenas do *notebook*.
3. **Modelagem, Otimização e Teste:**
    - Treinar e comparar múltiplos algoritmos de classificação, como **Gradient Boosting** (XGBoost/LightGBM) e **Random Forest**.
    - Usar **validação cruzada** e otimização de hiperparâmetros para selecionar o modelo final. A **AUC-ROC** e o **Recall** são métricas-chave.
    - Testar o modelo final em um *dataset* de **teste** não visto e, se aprovado, salvar o **Pipeline completo** (transformações + modelo) em **`src/models/`** usando a biblioteca **`joblib`**.

Fase II: Desenvolvimento Backend (API REST)

Esta fase constrói a camada de serviço que transforma o modelo em um produto acessível via rede.

1. **Estrutura, Dependências e Qualidade de Código:**
    - Inicializar o projeto usando **Poetry** para gerenciar as dependências de forma isolada e rastreável (`pyproject.toml`).
    - Criar um **Makefile** com *targets* essenciais (Ex: `make install`, `make run`, `make test`).
    - Configurar o **Black** para formatação automática de código e *linters* (Ex: Flake8/Mypy) para garantir a qualidade.
2. **Criação da API de Previsão (FastAPI):**
    - Desenvolver o serviço principal (`src/service/api.py`) utilizando o *framework* **FastAPI** devido à sua alta performance e *Type Hinting*.
    - Definir os **Esquemas de Dados** (`src/service/schemas.py`) usando **Pydantic** para validar a entrada (payload JSON do cliente) e a saída da API.
    - Implementar o *endpoint* `/predict` (método **POST**) que deve: **(a)** Carregar o *Pipeline* `joblib` na inicialização, **(b)** Receber e validar o Pydantic, **(c)** Passar os dados pelo *Pipeline* carregado, **(d)** Fazer a previsão e retornar o status (APROVADO ou NEGADO) e a probabilidade de risco.
3. **Testes e Documentação da API:**
    - Escrever **testes unitários e de integração** usando **Pytest** para garantir que a lógica de *preprocessing* e o *endpoint* `/predict` funcionem como esperado.
    - Aproveitar a geração automática de documentação **Swagger/OpenAPI** do FastAPI, garantindo que as descrições dos *endpoints* estejam claras para a equipe de integração.

Fase III: MLOps e Implantação Contínua

Esta fase garante que o serviço seja operacional, escalável e fácil de manter.

1. **Containerização e Orquestração Local:**
    - Criar um **Dockerfile** otimizado para empacotar a aplicação FastAPI, o modelo serializado e todas as dependências do Poetry.
    - Configurar um **Docker Compose** (`docker-compose.yml`) para orquestrar e testar o serviço localmente, simulando o ambiente de produção.
2. **Versionamento e Pipeline CI/CD:**
    - Garantir que todo o código (incluindo o modelo serializado) esteja sob **Versionamento de Código** (Git/GitHub).
    - Implementar um *pipeline* de **Integração Contínua/Entrega Contínua (CI/CD)** usando ferramentas como **GitHub Actions** ou **GitLab CI**. O *pipeline* deve automatizar as seguintes etapas na ordem:
        - Rodar testes e *linters*.
        - Realizar o *build* da Imagem Docker.

        - Fazer o *push* da imagem para um Registro (Ex: Docker Hub, AWS ECR).
        - Implantar o container na Infraestrutura Host.
3. **Hospedagem e Monitoramento:**
    - Implantar o container Docker em uma plataforma de hospedagem *cloud* (AWS ECS/EKS, GCP GKE, Azure AKS, ou serviços mais simples como Heroku/Render).
    - Configurar um sistema de **Monitoramento** para acompanhar: a saúde da API (*uptime* e latência), **Drift de Dados** (mudança nas características da entrada) e **Drift de Conceito** (queda na performance do modelo em produção).