Estrutura de Diretórios Sugerida

A seguir, uma proposta de organização do repositório para garantir modularidade, clareza e evitar conflitos na execução de algoritmos:

```
project-root/
├── README.md              # Visão geral do projeto, instruções de setup
├── LICENSE                # Licença do projeto
├── config/                # Arquivos de configuração (YAML, JSON)
│   ├── config.yaml        # Configurações gerais (caminhos,
hiperparâmetros)
│   └── logging.yaml       # Configuração de logging
├── data/
│   ├── raw/               # Dados brutos originais (CSV, JSON)
│   ├── interim/           # Dados pré-processados em etapas intermediárias
│   ├── processed/         # Dados finais prontos para treino/avaliação
│   └── external/          # Dados de fontes externas (ex: mapeamentos MCC)
├── notebooks/             # Jupyter Notebooks de exploração e prototipagem
│   ├── 01_EDA.ipynb
│   ├── 02_feature_engineering.ipynb
│   └── 03_model_training.ipynb
├── src/                   # Código-fonte do projeto
│   ├── __init__.py
│   ├── data/
│   │   ├── make_dataset.py      # Leitura e mesclagem de dados
│   │   └── preprocess.py        # Funções de limpeza e transformação
│   ├── features/
│   │   └── build_features.py    # Engenharia de features para fraude e
séries
│   ├── models/
│   │   ├── train_fraud_model.py # Training pipeline de detecção de fraudes
│   │   ├── train_ts_model.py    # Pipeline para modelos de séries temporais
│   │   └── predict.py           # Scripts de inferência em tempo real
│   ├── utils/
│   │   ├── logging.py           # Configuração de logs
│   │   ├── metrics.py           # Cálculo de métricas (AUC, F1, MAE)
│   │   └── visualization.py     # Funções de plotagem de gráficos
│   └── config.py                # Parsing de arquivos de configuração
├── tests/                 # Testes unitários e de integração
│   ├── test_preprocess.py
│   ├── test_features.py
│   └── test_models.py
├── scripts/               # Scripts de linha de comando
│   ├── run_ingest.sh      # Ingestão de dados completa
│   ├── run_train.sh       # Treinamento de modelos
│   └── run_eval.sh        # Avaliação e geração de relatórios
├── logs/                  # Logs gerados por execuções
├── outputs/               # Artefatos de saída (modelos, relatórios,
```

```
gráficos)
│   ├── models/             # Modelos treinados (pickle, joblib)
│   └── figures/            # Gráficos de EDA e métricas
└── environment.yml        # Dependências Conda / requirements.txt
```

Justificativas

- **Separação de dados**: diretórios `raw`, `interim` e `processed` evitam sobrescrever dados brutos e facilitam reprodutibilidade.
- **Modularização do código**: subdivisão em `data`, `features`, `models` e `utils` torna mais claro onde cada parte do pipeline reside.
- **Notebooks isolados**: notebooks separados por etapa para não poluir o código-fonte e facilitar a prototipagem.
- **Testes**: pasta `tests` com testes unitários garante qualidade e previne regressões.
- **Configurações centralizadas**: usar arquivos em `config/` evita hardcoding de parâmetros no código.
- **Scripts de automação**: scripts shell em `scripts/` permitem executar pipelines completas com um comando.
- **Logs e outputs**: diretórios dedicados simplificam análise de execução e versionamento de artefatos.

Dica adicional: use ferramentas como `make` ou `invoke` para encadear tarefas (ingestão, treino, avaliação) de forma padronizada e evitar conflitos de dependências entre etapas.