Diário de Estudo - Deploying a Scalable ML Pipeline with FastAPI

Documento para registrar o progresso e aprendizados durante o desenvolvimento do projeto.

Progresso do Projeto

1. Clonei o projeto starter

- Repositório clonado do Udacity
- Projeto base com estrutura de ML pipeline usando FastAPI

2. Instalei o ambiente virtual usando uv

- Criado `.venv` com Python 3.12
- Configurado `.python-version` para garantir versão consistente
- Migrado dependências do `requirements.txt` para `pyproject.toml`
- Dependências de produção: fastapi, gunicorn, pandas, scikit-learn, uvicorn
- Dependências de desenvolvimento: pytest, ruff

3. Criei o GitHub Actions (CI)

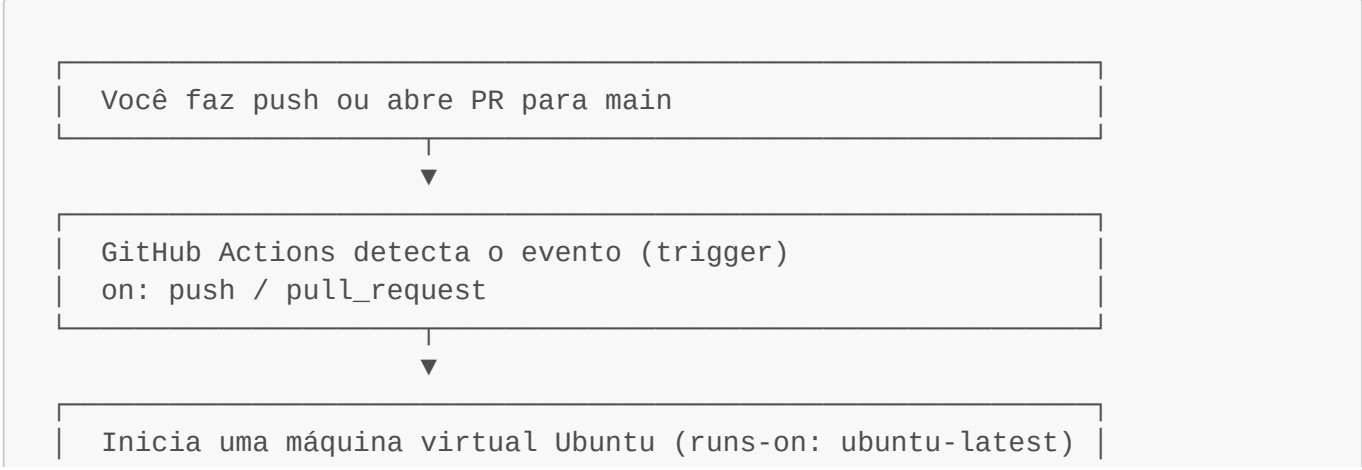
- Arquivo: `.github/workflows/ci.yml`
- Substituído flake8 por **ruff** (mais moderno e rápido)
- Configurado para rodar pytest e ruff

Explicações e Conceitos

GitHub Actions

GitHub Actions é um sistema de **CI/CD** (Integração Contínua / Entrega Contínua) integrado ao GitHub. Ele executa automaticamente tarefas quando eventos acontecem no repositório.

Fluxo de execução



```
┌─────────────────────────────────────────────────────────────┐
│                          ▼                                  │
┌─────────────────────────────────────────────────────────────┐
│  Executa os steps em sequência:                             │
│                                                             │
│  1. Checkout code      → Baixa seu código                   │
│  2. Install uv         → Instala o gerenciador uv           │
│  3. Set up Python      → Instala Python (lê .python-version)│
│  4. Install deps       → uv sync (lê uv.lock)               │
│  5. Ruff check         → Verifica erros de código           │
│  6. Ruff format        → Verifica formatação                │
│  7. Pytest             → Roda os testes                     │
└─────────────────────────────────────────────────────────────┘
                          ▼
┌─────────────────────────────────────────────────────────────┐
│  Se TODOS passarem → ✅ CI verde                            │
│  Se QUALQUER falhar → ✗ CI vermelho (bloqueia merge)        │
└─────────────────────────────────────────────────────────────┘
```

Estrutura do arquivo ci.yml

```yaml
name: CI                    # Nome do workflow

on:                         # QUANDO executar
  push:
    branches: [main]        # Em push para main
  pull_request:
    branches: [main]        # Em PRs para main

jobs:                       # O QUE executar
  test-and-lint:
    runs-on: ubuntu-latest  # ONDE executar

    steps:                  # PASSOS sequenciais
      - name: Checkout      # Cada step tem um nome
        uses: actions/...   # "uses" = action pronta (do marketplace)

      - name: Run tests
        run: uv run pytest  # "run" = comando bash
```

Por que usar CI/CD?

1. **Qualidade** - Garante que código quebrado não entre no main
2. **Automação** - Não precisa rodar testes manualmente
3. **Consistência** - Todo mundo segue o mesmo padrão
4. **Feedback rápido** - Sabe imediatamente se algo quebrou

Fase Zero: EDA - Análise Exploratória (Must Do Before ML Pipeline)

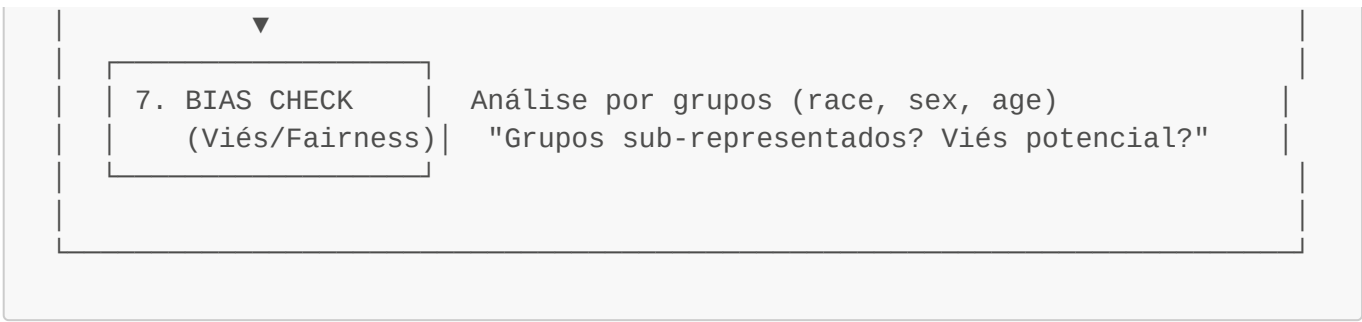
Antes de escrever qualquer código de `process_data()` ou `train_model()`, você **DEVE** entender profundamente seus dados. Pular essa etapa é o erro mais comum em projetos de ML.

Por que EDA é obrigatória?

```
┌──────────────────────────────────────────────────────────────────────┐
│  SEM EDA (Erro comum)                                                │
│  ─────────────────────                                               │
│  Dados → process_data() → train_model() → Métricas ruins → "Por quê?"  │
│                                                                      │
│  COM EDA (Abordagem correta)                                         │
│  ──────────────────────────                                          │
│  Dados → ENTENDER → Decisões informadas → Pipeline → Métricas boas   │
└──────────────────────────────────────────────────────────────────────┘
```

Workflow de Análise Recomendado

```
┌──────────────────────────────────────────────────────────────────────┐
│                    WORKFLOW EDA - 7 ETAPAS                           │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│   ┌─────────────────┐                                                │
│   │ 1. FIRST LOOK   │  df.shape, df.head(), df.info()                │
│   │    (Visão Geral)│  "Quantas linhas? Quantas colunas? Tipos?"     │
│   └────────┬────────┘                                                │
│            ▼                                                         │
│   ┌─────────────────┐                                                │
│   │ 2. TARGET       │  df['salary'].value_counts(normalize=True)     │
│   │    (Variável Y) │  "Balanceado ou desbalanceado?"                │
│   └────────┬────────┘                                                │
│            ▼                                                         │
│   ┌─────────────────┐                                                │
│   │ 3. MISSING DATA │  df.isnull().sum(), df.isna().mean()           │
│   │    (Dados Falt.)│  "Preciso imputar? Remover linhas?"            │
│   └────────┬────────┘                                                │
│            ▼                                                         │
│   ┌─────────────────┐                                                │
│   │ 4. CATEGORICAL  │  df.select_dtypes('object').nunique()          │
│   │    (Categóricas)│  "Quantas categorias? Alta cardinalidade?"     │
│   └────────┬────────┘                                                │
│            ▼                                                         │
│   ┌─────────────────┐                                                │
│   │ 5. NUMERICAL    │  df.describe(), histogramas, boxplots          │
│   │    (Numéricas)  │  "Outliers? Distribuição? Escala?"             │
│   └────────┬────────┘                                                │
│            ▼                                                         │
│   ┌─────────────────┐                                                │
│   │ 6. CORRELATIONS │  df.corr(), heatmap, pairplot                  │
│   │    (Correlações)│  "Features redundantes? Relação com target?"   │
│   └────────┬────────┘                                                │
```



Aplicação ao Dataset Census (census.csv)

Contexto do Dataset

Característica	Valor
Linhas	32,562
Colunas	15 (14 features + 1 target)
Problema	Classificação binária
Target	`salary` (<=50K ou >50K)
Origem	UCI Machine Learning Repository (Census Income)

Colunas do Dataset

Coluna	Tipo	Descrição
`age`	Numérica	Idade
`workclass`	Categórica	Tipo de empregador (Private, Gov, Self-emp...)
`fnlgt`	Numérica	Final weight (peso amostral do censo)
`education`	Categórica	Nível educacional
`education-num`	Numérica	Educação codificada numericamente
`marital-status`	Categórica	Estado civil
`occupation`	Categórica	Ocupação profissional
`relationship`	Categórica	Relação familiar
`race`	Categórica	Raça
`sex`	Categórica	Sexo
`capital-gain`	Numérica	Ganhos de capital
`capital-loss`	Numérica	Perdas de capital
`hours-per-week`	Numérica	Horas trabalhadas por semana
`native-country`	Categórica	País de origem

Coluna	Tipo	Descrição
salary	**Target**	<=50K ou >50K

Checklist EDA para Census

```
┌──────────────────────────────────────────────────────────────────────┐
│  CHECKLIST - Census Dataset                                          │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│  [ ] 1. DESBALANCEAMENTO DO TARGET                                   │
│      - salary: ~75% <=50K vs ~25% >50K                               │
│      - Decisão: Usar métricas além de accuracy (F1, Precision, Recall) │
│      - Considerar: class_weight='balanced' no modelo                 │
│                                                                      │
│  [ ] 2. VALORES FALTANTES                                            │
│      - workclass, occupation, native-country podem ter " ?"          │
│      - Decisão: Tratar " ?" como categoria ou imputar?               │
│                                                                      │
│  [ ] 3. FEATURES REDUNDANTES                                         │
│      - education vs education-num (mesma informação!)                │
│      - Decisão: Manter apenas uma                                    │
│                                                                      │
│  [ ] 4. ALTA CARDINALIDADE                                           │
│      - native-country: ~41 países únicos                            │
│      - Decisão: Agrupar países raros em "Other"?                     │
│                                                                      │
│  [ ] 5. OUTLIERS EM NUMÉRICAS                                        │
│      - capital-gain/loss: maioria é 0, poucos valores altos          │
│      - fnlgt: pesos amostrais (considerar remover?)                  │
│      - Decisão: Normalizar? Log transform?                           │
│                                                                      │
│  [ ] 6. ANÁLISE DE VIÉS (FAIRNESS)                                   │
│      - sex: Modelo performa igual para Male/Female?                  │
│      - race: Performance consistente entre grupos?                   │
│      - Decisão: Usar performance_on_categorical_slice()              │
│                                                                      │
│  [ ] 7. CORRELAÇÃO COM TARGET                                        │
│      - Quais features mais influenciam salary?                       │
│      - education-num, hours-per-week, age tendem a correlacionar     │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

Código Python para EDA Rápida

```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

```python
# 1. Carregar e visão geral
df = pd.read_csv("data/census.csv")
print(f"Shape: {df.shape}")
print(df.info())
print(df.head())

# 2. Target - Verificar desbalanceamento
print("\n=== TARGET DISTRIBUTION ===")
print(df['salary'].value_counts(normalize=True))

# 3. Missing values (incluindo " ?" que é comum neste dataset)
print("\n=== MISSING VALUES ===")
print(df.isnull().sum())
print("\nValores ' ?' por coluna:")
for col in df.select_dtypes('object').columns:
    count = (df[col].str.strip() == '?').sum()
    if count > 0:
        print(f"  {col}: {count}")

# 4. Categóricas - Cardinalidade
print("\n=== CATEGORICAL CARDINALITY ===")
for col in df.select_dtypes('object').columns:
    print(f"  {col}: {df[col].nunique()} unique values")

# 5. Numéricas - Estatísticas
print("\n=== NUMERICAL STATS ===")
print(df.describe())

# 6. Verificar redundância education vs education-num
print("\n=== REDUNDANCY CHECK ===")
print(df.groupby('education')['education-num'].mean().sort_values())

# 7. Correlação (apenas numéricas)
print("\n=== CORRELATION WITH TARGET ===")
df_numeric = df.copy()
df_numeric['salary_binary'] = (df['salary'] == '>50K').astype(int)
correlations = df_numeric.select_dtypes('number').corr()
['salary_binary'].drop('salary_binary')
print(correlations.sort_values(ascending=False))
```

Decisões que a EDA Informa

Descoberta na EDA	Impacto no Pipeline
Target desbalanceado (75/25)	Usar F1-score, não accuracy. Considerar `class_weight`
`education` e `education-num` redundantes	Remover uma delas no `process_data()`
" ?" como missing value	Decidir: tratar como categoria ou imputar
`native-country` com 41 valores	Considerar agrupar países raros
`fnlgt` é peso amostral	Provavelmente não usar como feature

Descoberta na EDA	Impacto no Pipeline
Maioria de `capital-gain/loss` é 0	Considerar binarizar (teve ganho? S/N)
Diferença de performance por `sex`/`race`	Usar `performance_on_categorical_slice()`

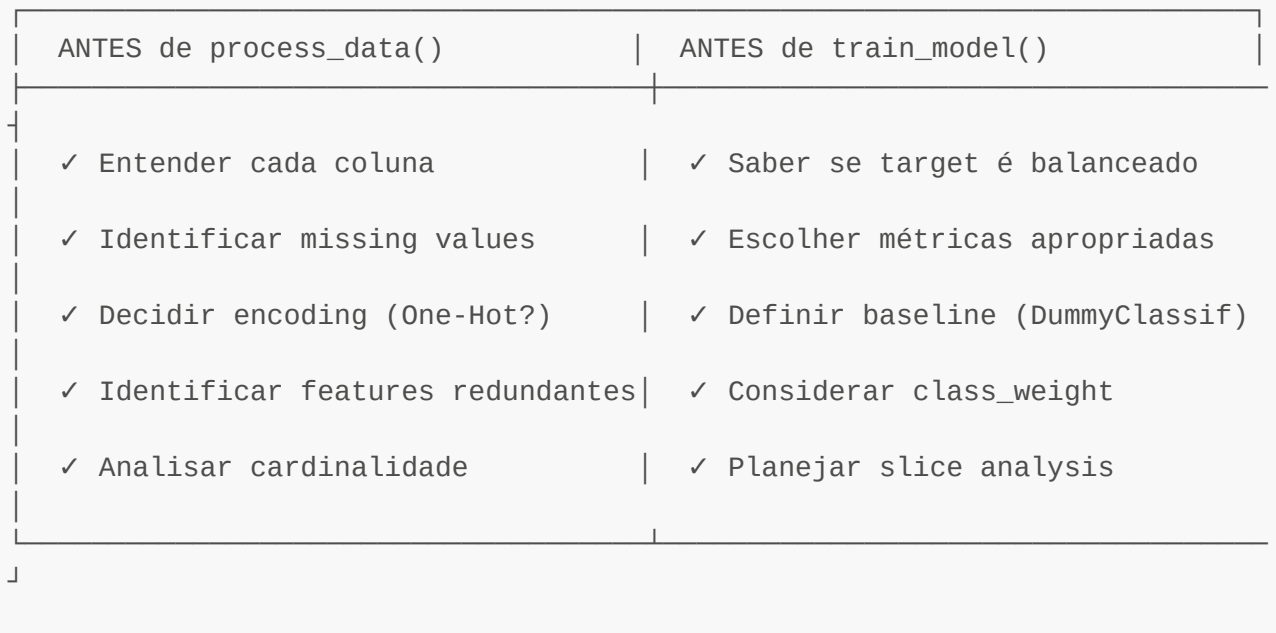
Ferramentas Recomendadas para EDA

Ferramenta	Uso	Instalação
pandas-profiling	Relatório EDA automático completo	`uv add ydata-profiling`
sweetviz	Comparação train/test visual	`uv add sweetviz`
matplotlib/seaborn	Visualizações customizadas	Já instalado com pandas

```
# Exemplo: Relatório automático com ydata-profiling
from ydata_profiling import ProfileReport

df = pd.read_csv("data/census.csv")
profile = ProfileReport(df, title="Census EDA Report", explorative=True)
profile.to_file("eda_report.html")
```

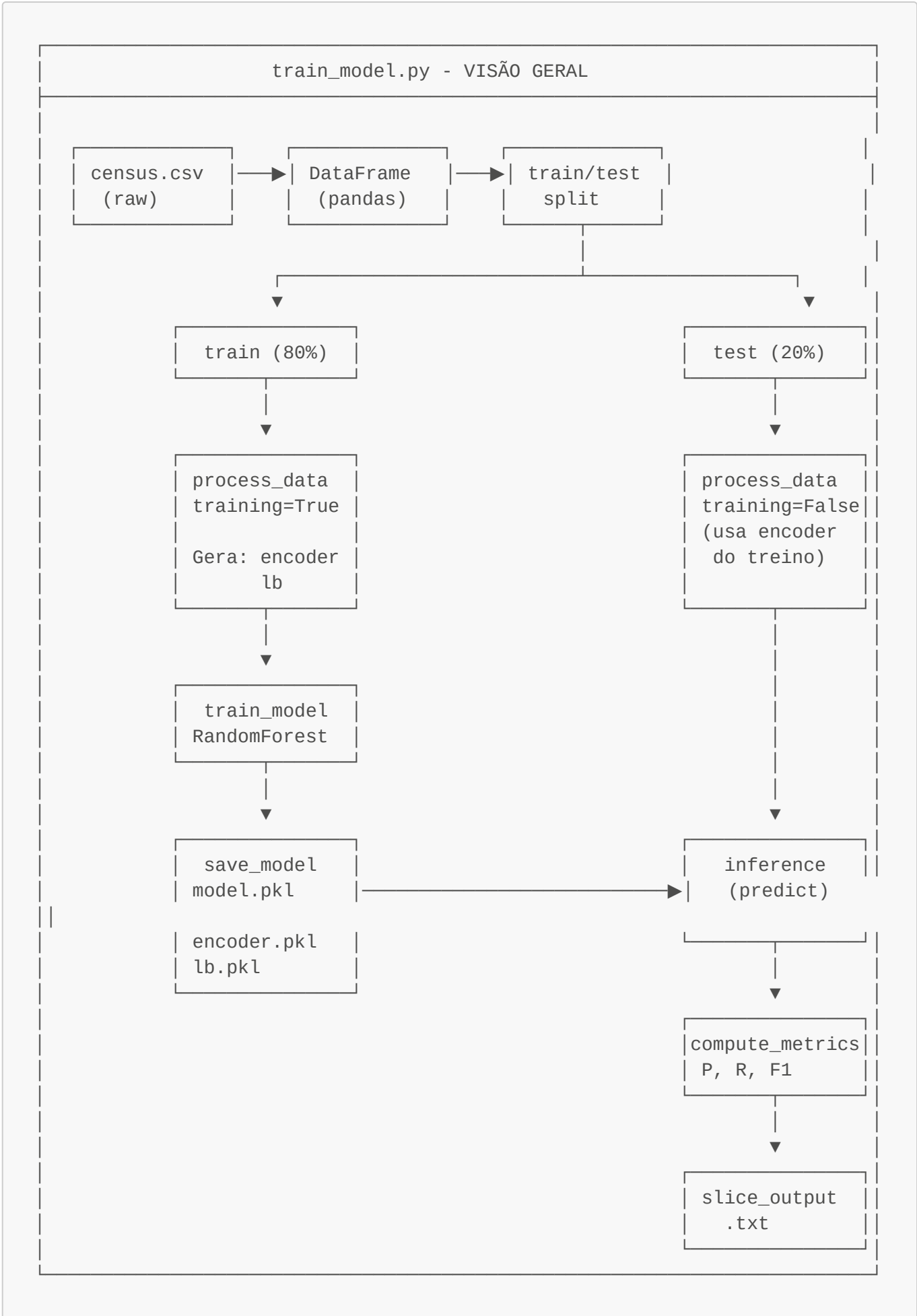
Resumo: O que fazer ANTES de codar o pipeline



Fase 1: Machine Learning Pipeline - Entendendo o train_model.py

O script `train_model.py` é o **orquestrador** do pipeline de ML. Ele conecta todas as peças: carregamento, processamento, treino, avaliação e serialização.

Anatomia do Pipeline



Passo a Passo Detalhado

1. Carregamento dos Dados

```
project_path = os.getcwd()
data_path = os.path.join(project_path, "data", "census.csv")
data = pd.read_csv(data_path)
```

Por que `os.getcwd()`?

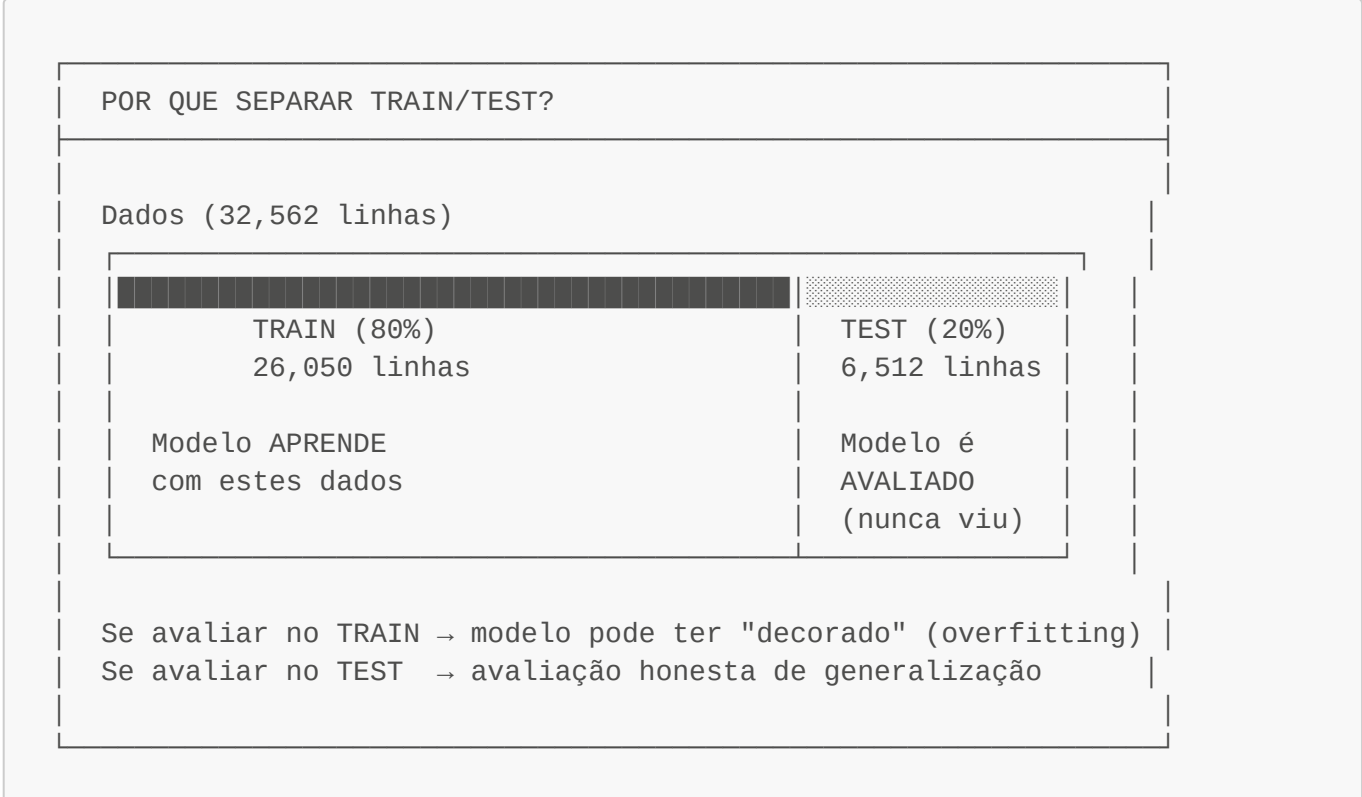
- Retorna o diretório atual de trabalho
- Torna o script **portável** (funciona em qualquer máquina)
- Alternativa: usar `Path(__file__).parent` para path relativo ao script

2. Split Train/Test

```
train, test = train_test_split(data, test_size=0.20, random_state=42)
```

Conceitos importantes:

Parâmetro	Valor	Significado
`test_size`	0.20	20% para teste, 80% para treino
`random_state`	42	Semente para reprodutibilidade



3. Processamento de Dados (process_data)

```
# TREINO: Cria os encoders
X_train, y_train, encoder, lb = process_data(
    train,
    categorical_features=cat_features,
    label="salary",
    training=True  # <-- IMPORTANTE
)

# TESTE: Usa os encoders do treino
X_test, y_test, _, _ = process_data(
    test,
    categorical_features=cat_features,
    label="salary",
    training=False,  # <-- IMPORTANTE
    encoder=encoder,  # <-- Reutiliza do treino
    lb=lb             # <-- Reutiliza do treino
)
```

Por que `training=True` vs `training=False`?

```
┌─────────────────────────────────────────────────────────────┐
│  DATA LEAKAGE - O Erro Mais Comum em ML                     │
├─────────────────────────────────────────────────────────────┤
│                                                             │
│  × ERRADO: Fazer fit no dataset completo                    │
│                                                             │
│     encoder.fit(TODOS_OS_DADOS)  # Inclui teste!            │
│     encoder.transform(train)                                │
│     encoder.transform(test)  # Vazou informação do teste!   │
│                                                             │
│  ✅ CORRETO: Fazer fit APENAS no treino                     │
│                                                             │
│     encoder.fit(train)       # training=True                │
│     encoder.transform(train)                                │
│     encoder.transform(test)  # training=False, usa encoder  │
│                                                             │
│  Data Leakage = Modelo "vê" dados de teste durante treino   │
│  Resultado: Métricas infladas, modelo ruim em produção      │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

O que `process_data` faz internamente:

```
┌─────────────────────────────────────────────────────────────┐
│  INPUT: DataFrame com colunas mistas                        │
│                                                             │
│  age | workclass | education | ... | salary                 │
│  39  | State-gov | Bachelors | ... | <=50K                  │
│                                                             │
```

```
┌─────────────────────────────────────────────────────────────────┐
│  STEP 1: Separar features categóricas e contínuas               │
│                                                                 │
│  Contínuas: age, fnlgt, education-num, capital-gain, ...        │
│  Categóricas: workclass, education, marital-status, ...         │
│                                                                 │
├─────────────────────────────────────────────────────────────────┤
│  STEP 2: One-Hot Encoding (categóricas → números)               │
│                                                                 │
│  workclass = "State-gov"                                        │
│  ↓                                                              │
│  workclass_Federal-gov: 0                                       │
│  workclass_Local-gov: 0                                         │
│  workclass_Private: 0                                           │
│  workclass_Self-emp-inc: 0                                      │
│  workclass_State-gov: 1  ← Hot!                                 │
│  ...                                                            │
│                                                                 │
├─────────────────────────────────────────────────────────────────┤
│  STEP 3: Label Binarizer (target → 0/1)                         │
│                                                                 │
│  salary = "<=50K" → 0                                           │
│  salary = ">50K"  → 1                                           │
│                                                                 │
├─────────────────────────────────────────────────────────────────┤
│  STEP 4: Concatenar tudo                                        │
│                                                                 │
│  X = [contínuas] + [categóricas_encoded]                        │
│  y = [labels_binarized]                                         │
│                                                                 │
├─────────────────────────────────────────────────────────────────┤
│  OUTPUT: Arrays numpy prontos para sklearn                      │
│                                                                 │
│  X: (26050, 108)  ← 108 features após encoding                  │
│  y: (26050,)      ← Labels binárias                             │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘
```

4. Treinamento do Modelo

```
model = train_model(X_train, y_train)
```

Internamente usa RandomForestClassifier:

```
┌─────────────────────────────────────────────────────────────────┐
│  RANDOM FOREST - Como Funciona                                  │
├─────────────────────────────────────────────────────────────────┤
│                                                                 │
│  É um "comitê" de árvores de decisão que votam:                 │
```

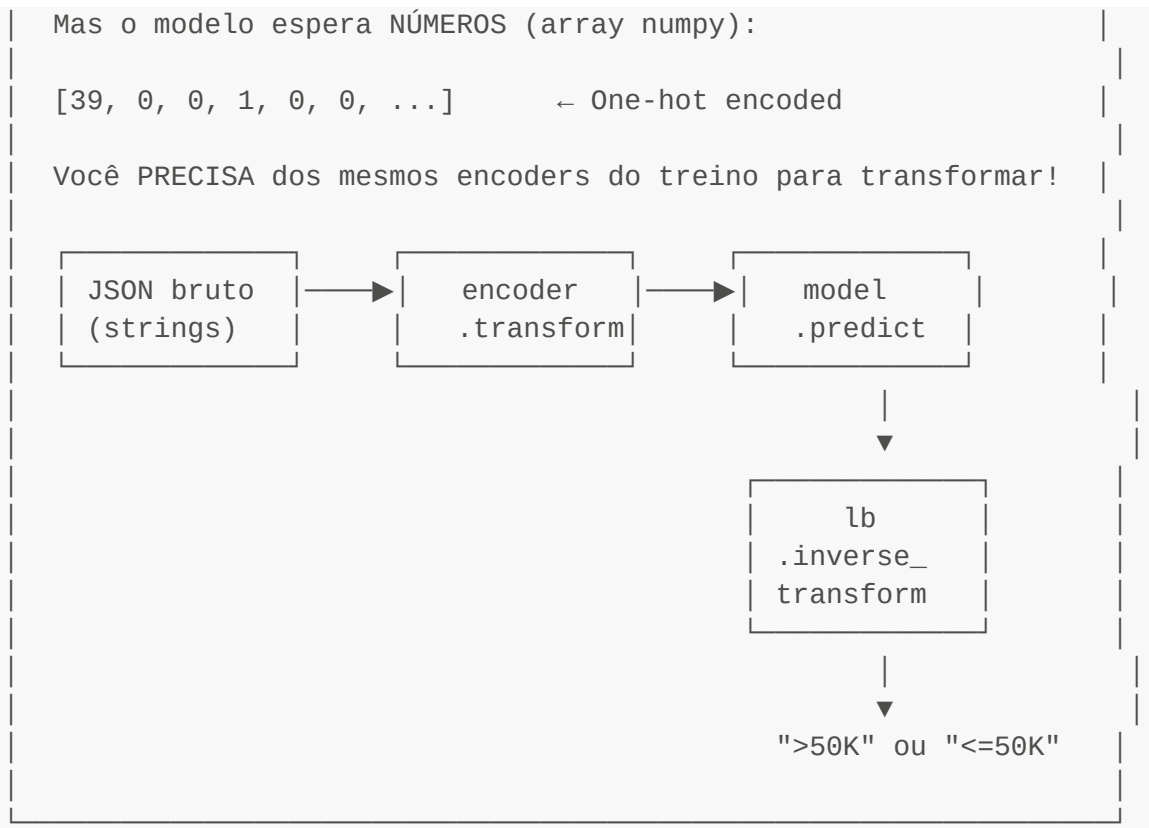
```
│                                                                      │
│      Árvore 1    Árvore 2    Árvore 3   ...   Árvore 100            │
│         │           │           │                 │                │
│         ▼           ▼           ▼                 ▼                │
│       <=50K       >50K       <=50K     ...     <=50K               │
│                                                                      │
│                   VOTAÇÃO                                            │
│                      │                                               │
│                      ▼                                               │
│              Maioria: <=50K ← Predição final                         │
│                                                                      │
├──────────────────────────────────────────────────────────────────────┤
│  Parâmetros usados:                                                  │
│                                                                      │
│  n_estimators=100  → 100 árvores no comitê                           │
│  random_state=42   → Reprodutibilidade                               │
│                                                                      │
├──────────────────────────────────────────────────────────────────────┤
│  Por que Random Forest?                                              │
│                                                                      │
│  ✓ Funciona bem "out of the box"                                     │
│  ✓ Não precisa escalar features                                      │
│  ✓ Robusto a outliers                                                │
│  ✓ Bom para dados tabulares                                          │
│  ✓ Feature importance grátis                                         │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

5. Serialização (Salvando Artefatos)

```
save_model(model, "model/model.pkl")
save_model(encoder, "model/encoder.pkl")
save_model(lb, "model/lb.pkl")
```

Por que salvar encoder e lb também?

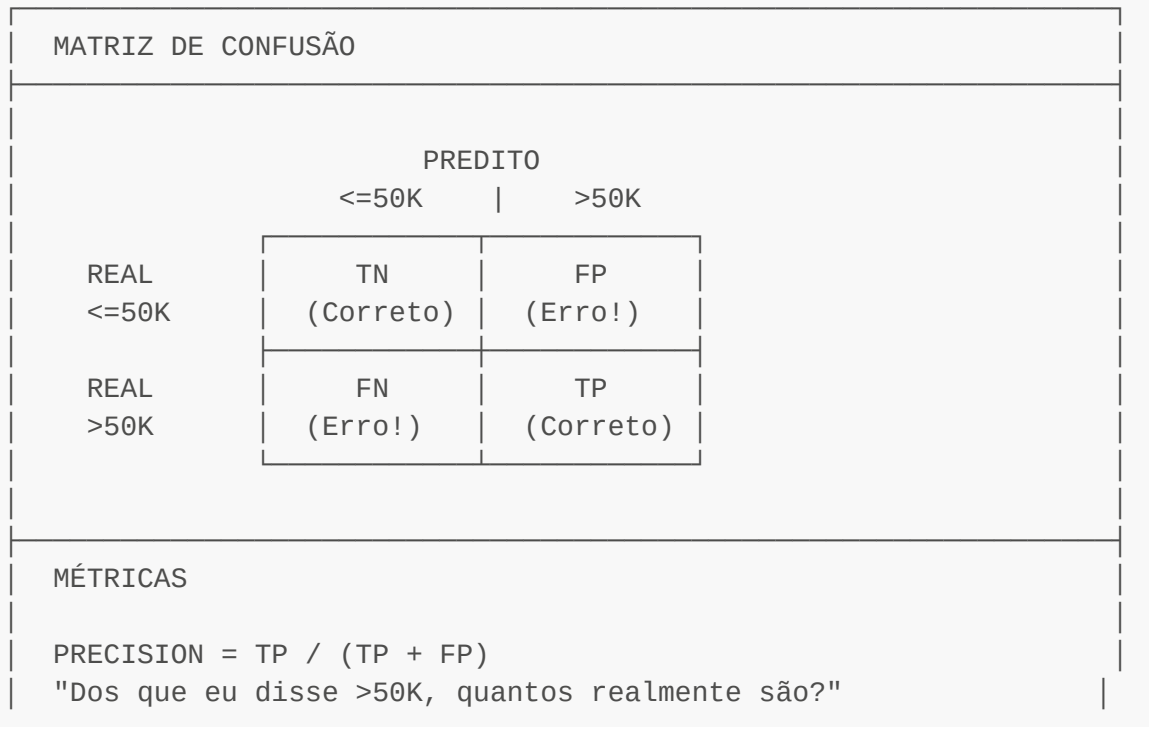
```
┌──────────────────────────────────────────────────────────────────────┐
│  ARTEFATOS NECESSÁRIOS PARA INFERÊNCIA                               │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│  Em PRODUÇÃO (FastAPI), você recebe dados BRUTOS:                    │
│                                                                      │
│  {                                                                   │
│    "age": 39,                                                        │
│    "workclass": "State-gov",   ← String!                            │
│    "education": "Bachelors",   ← String!                            │
│    ...                                                               │
│  }                                                                   │
│                                                                      │
```



6. Métricas de Avaliação

```
preds = inference(model, X_test)
p, r, fb = compute_model_metrics(y_test, preds)
```

Entendendo Precision, Recall e F1:



```
│  → Evita Falsos Positivos                                          │
│                                                                    │
│  RECALL = TP / (TP + FN)                                           │
│  "Dos que realmente são >50K, quantos eu encontrei?"               │
│  → Evita Falsos Negativos                                          │
│                                                                    │
│  F1 = 2 * (Precision * Recall) / (Precision + Recall)              │
│  "Média harmônica entre Precision e Recall"                        │
│  → Equilibra ambos                                                 │
│                                                                    │
├────────────────────────────────────────────────────────────────────┤
│  POR QUE NÃO USAR ACCURACY?                                        │
│                                                                    │
│  Dataset: 75% <=50K, 25% >50K (desbalanceado!)                     │
│                                                                    │
│  Modelo "burro" que SEMPRE prevê <=50K:                            │
│  → Accuracy = 75% (parece bom!)                                    │
│  → Precision = 0% (nunca acerta >50K)                              │
│  → Recall = 0% (nunca encontra >50K)                               │
│  → F1 = 0% (modelo inútil)                                         │
│                                                                    │
│  F1 é mais honesto para dados desbalanceados!                      │
│                                                                    │
└────────────────────────────────────────────────────────────────────┘
```

7. Slice Analysis (Análise de Viés)

```
for col in cat_features:
    for slicevalue in sorted(test[col].unique()):
        p, r, fb = performance_on_categorical_slice(
            data=test,
            column_name=col,
            slice_value=slicevalue,
            ...
        )
```

Por que isso é importante?

```
┌────────────────────────────────────────────────────────────────────┐
│  FAIRNESS / VIÉS DO MODELO                                         │
├────────────────────────────────────────────────────────────────────┤
│                                                                    │
│  Métricas GLOBAIS podem esconder problemas:                        │
│                                                                    │
│  F1 Global = 0.72 ← "Parece bom!"                                  │
│                                                                    │
│  Mas olhando por FATIAS (slices):                                  │
│                                                                    │
│  ┌──────────────────────────────────────────────────────┐          │
```

```
│   │ sex = Male      | F1 = 0.78  ← Melhor                  │       │
│   │ sex = Female    | F1 = 0.61  ← Pior!                   │       │
│   ├──────────────────────────────────────────────────────┤       │
│   │ race = White    | F1 = 0.74  ← Melhor                  │       │
│   │ race = Black    | F1 = 0.58  ← Pior!                   │       │
│   └──────────────────────────────────────────────────────┘       │
│                                                                   │
│  O modelo tem VIÉS! Performa pior para certos grupos.            │
│                                                                   │
│  Isso é crítico para:                                            │
│  - Compliance regulatório (LGPD, GDPR)                           │
│  - Ética em ML                                                   │
│  - Evitar discriminação algorítmica                              │
│                                                                   │
│  slice_output.txt documenta isso para auditoria.                 │
│                                                                   │
└──────────────────────────────────────────────────────────────────┘
```

Artefatos Gerados pelo Pipeline

Arquivo	Conteúdo	Uso
`model/model.pkl`	RandomForestClassifier treinado	Inferência na API
`model/encoder.pkl`	OneHotEncoder fitted	Transformar features categóricas
`model/lb.pkl`	LabelBinarizer fitted	Converter 0/1 → "<=50K"/">50K"
`slice_output.txt`	Métricas por categoria	Auditoria de viés/fairness

Resumo Visual do Pipeline

```
┌──────────────────────────────────────────────────────────────────┐
│                    CICLO DE VIDA ML                              │
├──────────────────────────────────────────────────────────────────┤
│                                                                   │
│    TREINO (train_model.py)          PRODUÇÃO (main.py)           │
│    ──────────────────────           ──────────────────           │
│                                                                   │
│    1. Carregar census.csv           1. Receber JSON request      │
│           ↓                                ↓                     │
│    2. Split train/test              2. Carregar .pkl files       │
│           ↓                                ↓                     │
│    3. process_data()                3. encoder.transform()       │
│           ↓                                ↓                     │
│    4. train_model()                 4. model.predict()           │
│           ↓                                ↓                     │
│    5. compute_metrics()             5. lb.inverse_transform()    │
│           ↓                                ↓                     │
│    6. save_model()                  6. Retornar ">50K"/"<=50K"   │
│           ↓                                                      │
│    7. slice_analysis()                                           │
```

```
│                                                                    │
│    Resultado:                        Resultado:                    │
│    - model.pkl                       - Predição em tempo real      │
│    - encoder.pkl                     - Latência baixa              │
│    - lb.pkl                          - Escalável                   │
│    - slice_output.txt                                              │
│                                                                    │
└────────────────────────────────────────────────────────────────────┘
```

Análise Completa do Projeto

Estrutura do Projeto

```
Project Root/
├── data/
│   └── census.csv          # Dataset de classificação (30k+ linhas)
├── model/
│   └── .gitignore          # Diretório para salvar modelos treinados
├── ml/
│   ├── __init__.py
│   ├── data.py             # ✅ Processamento de dados (COMPLETO)
│   └── model.py            # ⚠ Funções ML (PARCIAL - tem TODOs)
├── .github/workflows/
│   └── ci.yml              # ✅ CI/CD Pipeline (COMPLETO)
├── main.py                 # × FastAPI app (TODO)
├── train_model.py          # × Script de treino (TODO)
├── test_ml.py              # × Testes unitários (TODO)
├── local_api.py            # × Cliente API (TODO)
├── model_card_template.md  # × Documentação do modelo (TODO)
├── pyproject.toml          # ✅ Configuração moderna
└── .python-version         # ✅ Python 3.12
```

Fluxo do ML Pipeline

```
┌──────────────────────┐
│ 1. CARREGAR DADOS    │  train_model.py → pd.read_csv("data/census.csv")
└──────────┬───────────┘
           ▼
┌──────────────────────┐
│ 2. SPLIT DADOS       │  train_test_split() → train, test
└──────────┬───────────┘
           ▼
┌──────────────────────┐
│ 3. PROCESSAR         │  ml/data.py → process_data()
│    - One-Hot Encode  │  Categorical → números
│    - Label Binarize  │  salary → 0/1
└──────────┬───────────┘
           ▼
```

```
┌─────────────────────┐
│ 4. TREINAR MODELO   │  ml/model.py → train_model()
│    (RandomForest?)  │  Escolher algoritmo ML
└─────────────────────┘
          │
          ▼
┌─────────────────────┐
│ 5. AVALIAR          │  compute_model_metrics()
│    - Precision      │  performance_on_categorical_slice()
│    - Recall         │  → slice_output.txt
│    - F1 Score       │
└─────────────────────┘
          │
          ▼
┌─────────────────────┐
│ 6. SALVAR MODELO    │  save_model() → model/model.pkl
└─────────────────────┘
          │
          ▼
┌─────────────────────┐
│ 7. DEPLOY API       │  main.py → FastAPI
│    GET /            │  Mensagem de boas-vindas
│    POST /data/      │  Inferência do modelo
└─────────────────────┘
```

Status de Implementação

Componente	Arquivo	Status	O que falta
Processamento dados	ml/data.py	✅ Completo	-
Métricas	ml/model.py	✅ Completo	-
train_model()	ml/model.py	✗ TODO	Escolher algoritmo ML
inference()	ml/model.py	✗ TODO	Implementar predição
save/load_model()	ml/model.py	✗ TODO	Serialização pickle
slice_performance()	ml/model.py	✗ TODO	Filtrar por categoria
Script treino	train_model.py	✗ TODO	6 partes para completar
FastAPI app	main.py	✗ TODO	Endpoints GET/POST
Testes	test_ml.py	✗ TODO	Mínimo 3 testes
Cliente API	local_api.py	✗ TODO	Testar endpoints
Model Card	model_card_template.md	✗ TODO	Documentação

Próximos Passos (Ordem Recomendada)

Fase 1: Core ML

- ☐ Implementar `train_model()` em ml/model.py
- ☐ Implementar `inference()` em ml/model.py

- [ ] Implementar `save_model()` e `load_model()` em ml/model.py
- [ ] Implementar `performance_on_categorical_slice()` em ml/model.py

Fase 2: Pipeline de Treino

- [ ] Completar train_model.py (6 TODOs)
- [ ] Rodar treino e gerar modelo
- [ ] Gerar slice_output.txt

Fase 3: Testes

- [ ] Escrever 3+ testes em test_ml.py
- [ ] Garantir que CI passa

Fase 4: API

- [x] Implementar FastAPI em main.py
- [x] Implementar GET / e POST /data/
- [x] Testar com local_api.py

Fase 5: Documentação

- [ ] Preencher model_card_template.md
- [ ] Screenshots para entrega

Fase 4 Concluída: FastAPI + Cliente HTTP

O que foi implementado

1. main.py - API RESTful com FastAPI

```
# Estrutura básica de uma API FastAPI para ML
from fastapi import FastAPI
from pydantic import BaseModel, Field

# 1. Carregar artefatos do modelo ao iniciar
encoder = load_model("model/encoder.pkl")
model = load_model("model/model.pkl")

# 2. Criar instância da aplicação
app = FastAPI()

# 3. Endpoint GET - Health check / Welcome
@app.get("/")
async def get_root():
    return {"message": "Welcome to the ML inference API!"}

# 4. Endpoint POST - Inferência
@app.post("/data/")
async def post_inference(data: Data):
```

```
    # Processar dados → Inferência → Retornar resultado
    return {"result": apply_label(_inference)}
```

2. local_api.py - Cliente para testar a API

```
import requests

# GET request
r = requests.get("http://127.0.0.1:8000")
print(f"Status code: {r.status_code}")
print(f"Welcome message: {r.json()['message']}")

# POST request com dados JSON
r = requests.post("http://127.0.0.1:8000/data/", json=data)
print(f"Status code: {r.status_code}")
print(f"Result: {r.json()['result']}")
```

Padrão FastAPI para ML - Template Reutilizável

```
┌──────────────────────────────────────────────────────────────────────┐
│  PADRÃO: FastAPI + Modelo ML                                         │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│  1. DEFINIR SCHEMA (Pydantic)                                        │
│     ─────────────────────────                                        │
│     class InputData(BaseModel):                                      │
│         feature1: int = Field(..., example=42)                       │
│         feature2: str = Field(..., example="value")                  │
│                                                                      │
│     → Valida dados automaticamente                                   │
│     → Gera documentação OpenAPI                                      │
│     → Use `alias` para nomes com hífen/caracteres especiais          │
│                                                                      │
│  2. CARREGAR MODELOS NO STARTUP                                      │
│     ──────────────────────────                                       │
│     encoder = load_model("path/encoder.pkl")                         │
│     model = load_model("path/model.pkl")                             │
│                                                                      │
│     → Carrega UMA vez, não a cada request                            │
│     → Reduz latência significativamente                              │
│                                                                      │
│  3. ENDPOINT DE INFERÊNCIA                                           │
│     ──────────────────────                                           │
│     @app.post("/predict/")                                           │
│     async def predict(data: InputData):                              │
│         processed = preprocess(data)                                 │
│         result = model.predict(processed)                            │
│         return {"prediction": result}                                │
```

```
│                                                                              │
└──────────────────────────────────────────────────────────────────────────────┘
```

Pontos Importantes para Projetos Futuros

Conceito	O que fazer	Por quê
Pydantic alias	`alias="feature-name"`	Python não aceita `-` em variáveis
Field examples	`Field(..., example=42)`	Documenta API automaticamente
Carregar modelo global	Fora das funções	Evita reload a cada request
process_data training=False	Sempre em inferência	Usa encoder existente, não cria novo
requests.post com json=	`json=data` não `data=data`	Serializa dict para JSON

Como testar a API

```
# Terminal 1: Iniciar servidor
uvicorn main:app --reload

# Terminal 2: Testar com cliente
python local_api.py

# Alternativa: Acessar docs interativos
# http://127.0.0.1:8000/docs (Swagger UI)
# http://127.0.0.1:8000/redoc (ReDoc)
```

Fluxo de Dados na Inferência

```
┌──────────────────────────────────────────────────────────────────────────────┐
│  REQUEST → RESPONSE                                                          │
├──────────────────────────────────────────────────────────────────────────────┤
│                                                                              │
│  1. Cliente envia JSON                                                       │
│     {"age": 37, "workclass": "Private", ...}                                 │
│                          │                                                   │
│                          ▼                                                   │
│  2. Pydantic valida e converte para objeto Data                              │
│     data = Data(age=37, workclass="Private", ...)                            │
│                          │                                                   │
│                          ▼                                                   │
│  3. Converte para DataFrame (nomes com hífen)                                │
│     {"age": [37], "workclass": ["Private"], ...}                             │
│                          │                                                   │
│                          ▼                                                   │
│  4. process_data() transforma com encoder salvo                              │
│     [37, 0, 0, 1, 0, ...]  ← array numpy                                     │
│                          │                                                   │
```

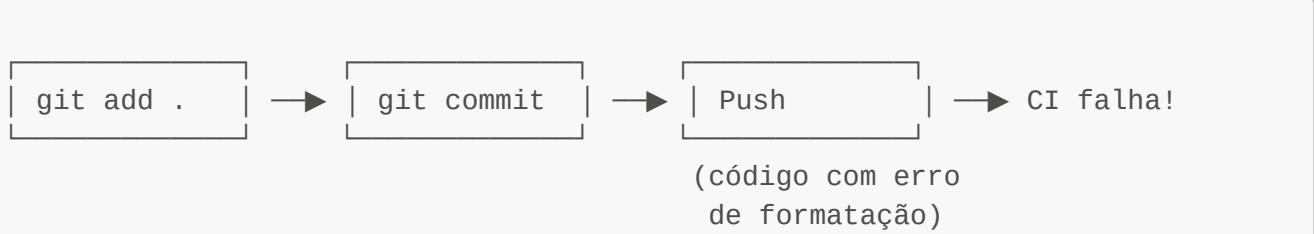
```
│                          ▼                                                  │
│  5. model.predict() retorna 0 ou 1                                          │
│                          │                                                  │
│                          ▼                                                  │
│  6. apply_label() converte para string                                      │
│     0 → "<=50K"  |  1 → ">50K"                                              │
│                          │                                                  │
│                          ▼                                                  │
│  7. Retorna JSON                                                            │
│     {"result": "<=50K"}                                                     │
│                                                                             │
└─────────────────────────────────────────────────────────────────────────────┘
```

Sugestões de Melhoria para Aprendizado

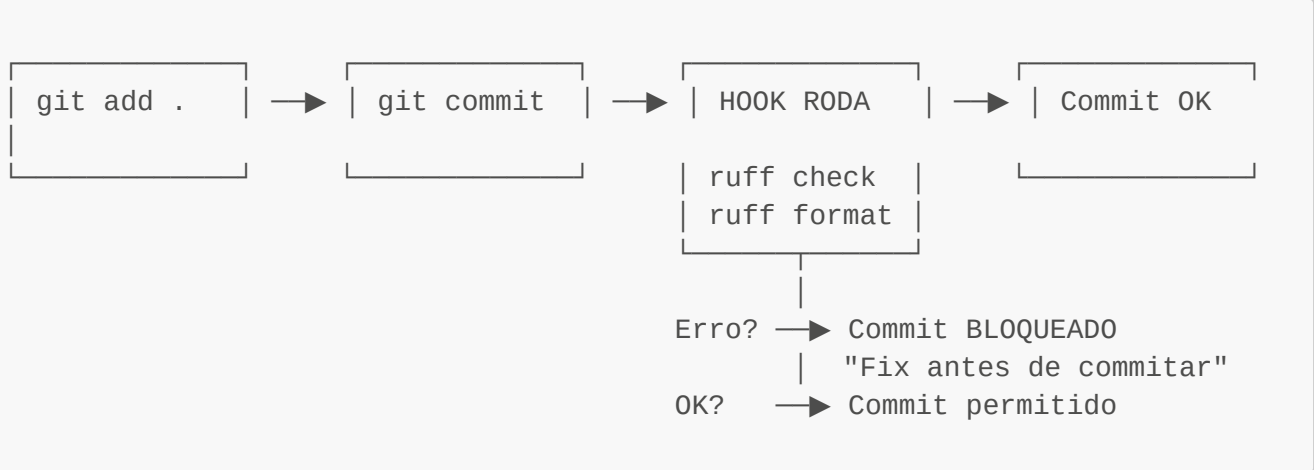
1. Pre-commit Hooks (Automatizar qualidade)

Pre-commit hooks são scripts que rodam automaticamente **antes** de cada commit. Se o script falhar, o commit é bloqueado.

Fluxo sem pre-commit



Fluxo com pre-commit



Benefícios

Sem pre-commit	Com pre-commit

Sem pre-commit	Com pre-commit
Descobre erro no CI (depois do push)	Descobre erro antes do commit
Perde tempo esperando CI	Feedback instantâneo
Commits com código feio	Código sempre formatado

Como configurar

Passo 1 - Instalar pre-commit:

```
uv add --dev pre-commit
```

Passo 2 - Criar `.pre-commit-config.yaml`:

```
repos:
  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.9.0
    hooks:
      - id: ruff          # linting
        args: [--fix]
      - id: ruff-format   # formatação
```

Passo 3 - Ativar os hooks:

```
uv run pre-commit install
```

Passo 4 - Pronto! Agora todo `git commit` roda ruff automaticamente.

2. Cobertura de Testes

Adicionar pytest-cov para ver % de código testado:

```
uv add --dev pytest-cov
uv run pytest --cov=ml --cov-report=html
```

3. Type Hints

Adicionar tipagem ao código para melhor documentação:

```
def train_model(X_train: np.ndarray, y_train: np.ndarray) ->
RandomForestClassifier:
    ...
```

4. Logging

Substituir prints por logging para produção:

```
import logging
logging.info("Treinando modelo...")
```

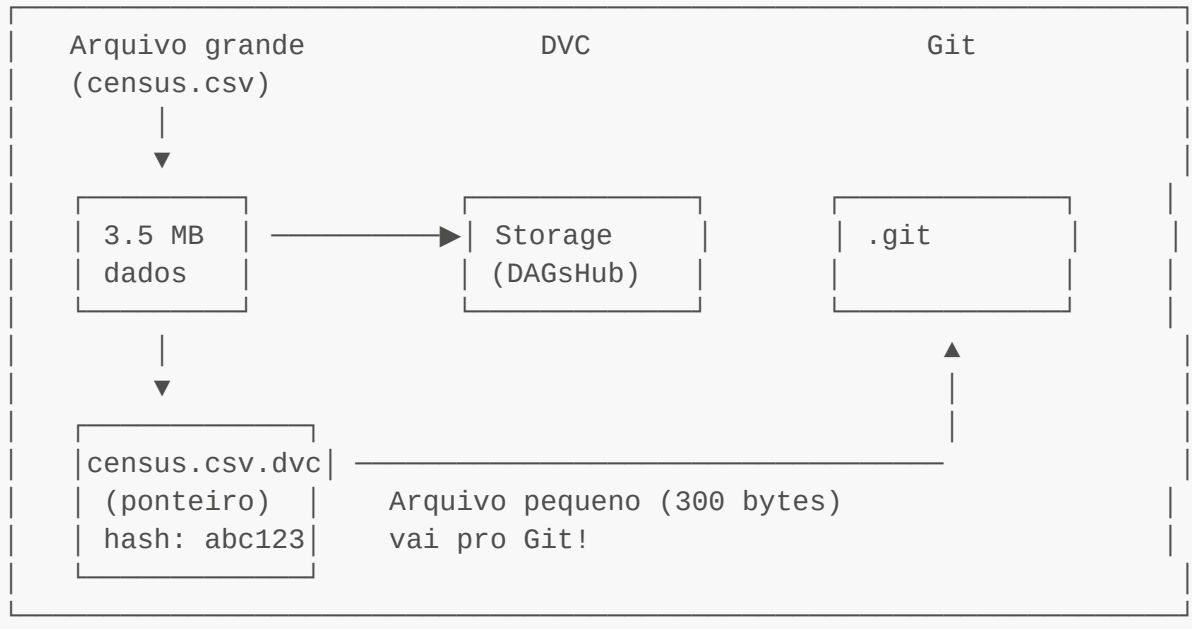
5. DVC (Data Version Control)

DVC é o "Git para dados". O Git não foi feito para arquivos grandes (datasets, modelos ML), então o DVC resolve isso.

O Problema

```
┌──────────────────────────────────┬──────────────────────────────────┐
│ Git funciona bem para:           │ Git NÃO funciona para:           │
├──────────────────────────────────┼──────────────────────────────────┤
│ Código (.py, .js, .yaml)         │ Datasets grandes (100MB+)        │
│ Arquivos pequenos                │ Modelos treinados (.pkl, .h5)    │
│ Texto                            │ Imagens/Vídeos para treino       │
└──────────────────────────────────┴──────────────────────────────────┘
```

Como o DVC funciona



Configuração com DAGsHub (nosso projeto)

Passo 1 - Instalar DVC:

```
uv add --dev dvc
```

Passo 2 - Inicializar DVC:

```
dvc init
```

Passo 3 - Configurar remote do DAGsHub:

```
# Adicionar remote
dvc remote add -d origin https://dagshub.com/FabioCLima/Deploying-a-
Scalable-ML-Pipeline-with-FastAPI.dvc

# Configurar autenticação (use seu token do DAGsHub)
dvc remote modify origin --local auth basic
dvc remote modify origin --local user FabioCLima
dvc remote modify origin --local password SEU_TOKEN_DAGSHUB
```

Passo 4 - Adicionar dataset ao DVC:

```
# Remover do Git (se já estava rastreado)
git rm -r --cached data/census.csv

# Adicionar ao DVC
dvc add data/census.csv

# Commitar ponteiro no Git
git add data/census.csv.dvc data/.gitignore
git commit -m "Track dataset with DVC"
```

Passo 5 - Enviar dados para DAGsHub:

```
dvc push
```

Comandos DVC úteis

```
# Baixar dados (outro dev ou CI)
dvc pull

# Ver status dos arquivos
```

```
dvc status

# Ver arquivos rastreados
dvc list .
```

Onde obter o token DAGsHub

1. Acesse: dagshub.com/user/settings/tokens
2. Clique em **"Generate New Token"**
3. Dê um nome (ex: "dvc-local")
4. Copie e guarde em local seguro

Comandos Úteis

```
# Sincronizar dependências
uv sync

# Rodar testes
uv run pytest

# Verificar linting
uv run ruff check .

# Corrigir linting automaticamente
uv run ruff check --fix .

# Verificar formatação
uv run ruff format --check .

# Formatar código
uv run ruff format .
```