Contexto

Uma empresa de telecomunicações quer reduzir a rotatividade de clientes (“churn”). Seu papel é entregar um pipeline de dados que identifique quais clientes têm maior probabilidade de cancelar o serviço, apoiando ações de retenção.

Objetivos

1. **Entender e preparar os dados** (limpeza, tratamento de nulos, transformação de tipos).
2. **Analisar (EDA)** padrões de comportamento de clientes cancelados vs. ativos.
3. **Construir e comparar** pelo menos **dois modelos de classificação** (ex: Random Forest, XGBoost, Logistic Regression).
4. **Avaliar** desempenho usando métricas apropriadas (ex: AUC-ROC, precisão, recall, F1-score).
5. **Interpretar** o modelo vencedor para entender quais variáveis mais influenciam o churn.
6. **(Opcional)** Criar um pequeno dashboard (Streamlit, Power BI ou Dash) para visualizar:
    - Distribuição de variáveis chave
    - Curva ROC do modelo final
    - Impacto das features (ex.: gráfico SHAP)

Dados

Baixe o conjunto “Telco Customer Churn” em CSV:

Principais colunas:

- customerID
- gender, SeniorCitizen, Partner, Dependents
- tenure, PhoneService, InternetService, ...
- Contract, PaperlessBilling, PaymentMethod
- MonthlyCharges, TotalCharges
- Churn (variável-alvo: “Yes”/“No”)

Entregáveis no GITHUB de forma Pública.

1. **Notebook Python** (Jupyter ou Colab) contendo:

    - Carregamento e pré-processamento dos dados
    - EDA comentada
    - Treinamento e comparação de modelos
    - Avaliação e escolha do modelo final
    - Interpretação das features (ex.: SHAP ou LIME)
2. **Script ou módulo Python** separado (ex: train.py) que reproduza o pipeline.
3. **Arquivo requirements.txt** com as dependências.
4. **Relatório resumido** (README.md ou PDF) explicando:
    - Principais decisões de pré-processamento
    - Escolha de algoritmos e parâmetros
    - Métricas alcançadas
    - Insights de negócio

Ferramentas sugeridas

- **Linguagem**: Python 3.x
- **Bibliotecas**: pandas, scikit-learn, XGBoost (ou LightGBM), matplotlib/plotly, shap (ou lime)
- **Opcional**: Streamlit, Dash ou Power BI para o dashboard

Diferenciais na entrega (Opcional)

Repositório Git organizado

- Estrutura de pastas clara, ex:

```
├── data/
│   ├── raw/                # CSVs originais (não versionar grandes volumes)
│   └── processed/          # Dados já limpos/prontos para modelagem
├── notebooks/              # Explorações e experimentos em Jupyter
├── src/                    # Código-fonte (scripts e módulos)
├── models/                 # Modelos treinados e artefatos serializados
├── reports/                # Relatórios gerados (PDF, HTML ou Markdown)
├── Dockerfile              # Para criar ambiente reproduzível
├── requirements.txt        # Dependências Python
├── README.md               # Visão geral, instruções de uso e badges
├── .gitignore              # Ignorar arquivos temporários e dados pesados
└── .github/
    └── workflows/
        └── ci.yml          # Exemplo de GitHub Actions para lint, testes e build
```

README.md completo

- Descrição do projeto.
- Como clonar e rodar localmente.
- Exemplo de execução (com comandos).
- Estrutura de pastas explicada.
- Badges de build e cobertura de testes (se aplicável).

Scripts modulares

- src/data_preprocessing.py
- src/train_model.py
- src/evaluate_model.py
- Cada script deve aceitar argumentos (e.g. paths de entrada/saída) para facilitar automação.

Notebooks de Análise Exploratório e Comparação de Modelos

- Um notebook “01_EDA.ipynb” só com análise e visualizações.
- Um notebook “02_Modeling.ipynb” comparando os dois (ou mais) algoritmos.

Pipeline reproducível

- **Makefile** ou **scripts Bash** (ex: make preprocess, make train, make evaluate).
- Ou **GitHub Actions** simples que rodem toda a pipeline no push.

Ambiente isolado

- requirements.txt ou environment.yml (conda), para instalar todas as dependências de uma vez.
- **Dockerfile** fornecendo imagem pronta para rodar todo o projeto (incluindo notebooks).

Documentação de decisões técnicas

- reports/Decision_Log.md com:
    - Justificativa de escolhas de modelo e parâmetros.
    - Principais desafios encontrados.
    - Lições aprendidas.

(Opcional) Testes automatizados

- Testes unitários para funções de pré-processamento e métricas de avaliação (usando pytest).
- Integração contínua via GitHub Actions para garantir que nada quebre.