# Programação para Ciência de Dados

## Pré-processamento de Dados - Transformação e Normalização

Arthur Casals

6 de Novembro de 2025

# Avisos

- Aula prática de amanhã (07/11): TEM ENTREGA

# Agenda

- Introdução: Por que Transformar Dados?
- Bloco 0: Preview de Machine Learning (Contexto)
- Bloco 1: Feature Engineering
- Bloco 2: Encoding de Variáveis Categóricas
- Bloco 3: Normalização e Padronização
- Bloco 4: Pipeline Completo e Análise Final

# A Próxima Etapa: Transformação

**Onde estamos no pipeline de Ciência de Dados:**

**1. Carregar Dados** ✓

↓

**2. Exploração (EDA)** ✓

↓

**3. Limpeza** ✓ (Aula 09)

↓

**4. TRANSFORMAÇÃO (HOJE!)**

*Enriquecer e preparar dados*

↓

**5. Análise Avançada**

*Insights profundos*

# Por que Transformar Dados?

**Dados limpos $\neq$ Dados prontos para análise avançada**

**Problemas que transformação resolve:**

- **Limitação em análises numéricas**
  - Categorias (texto) não podem ser correlacionadas
  - Precisa converter para números
  - ”Alto”, ”Médio”, ”Baixo”$\rightarrow$ 2, 1, 0
- **Escalas muito diferentes dificultam comparação**
  - Idade: 20-80 vs Salário: 1000-100000
  - Impossível comparar ”importância”visualmente
  - Gráficos ficam distorcidos

# Por que Transformar Dados?

**Dados limpos $\neq$ Dados prontos para análise avançada**

**Problemas que transformação resolve:**

- **Informação ”escondida”nos dados**
  - Insights não são óbvios nas variáveis originais
  - Precisa criar novas variáveis derivadas
  - Ex: não temos ”gasto médio mensal”, mas podemos calcular
- **Distribuições problemáticas**
  - Muito assimétrica dificulta visualização
  - Transformações podem normalizar

# Transformação: Impacto na Análise

**Exemplo: Analisar Cancelamento de Clientes (Churn)**

**Sem transformação adequada:**

- Análises superficiais e limitadas
- Não consegue correlacionar categorias
- Comparações injustas (escalas diferentes)
- Padrões importantes não são visíveis
- Visualizações ruins

**Com transformação adequada:**

- Análises profundas e quantitativas
- Correlações entre todas as variáveis
- Comparações justas e interpretáveis
- Padrões ocultos revelados
- Visualizações claras e informativas

# Transformação: Impacto na Análise

**Nota Importante**

Transformação = Preparar dados para extrair insights máximos

# Caso Real: Netflix

**Sistema de Análise de Preferências**

**Desafio inicial:**

- Entender quais fatores influenciam preferências
- Dados: gêneros (texto), duração (minutos), ano, rating
- Escalas diferentes: ano (1900-2024) vs rating (0-10)
- Categorias: mais de 100 gêneros diferentes

**Transformações aplicadas:**

- **Encoding:** Gêneros $\rightarrow$ Representação numérica
- **Feature Engineering:** ”Anos desde lançamento”, ”Era do filme”
- **Normalização:** Todas as variáveis comparáveis
- **Agregação:** Combinações de gêneros populares

# Caso Real: Netflix

**Sistema de Análise de Preferências**

**Resultado:**

- Análises revelam padrões claros de preferência
- Base para sistema de recomendação (Machine Learning)
- Economia de $1 bilhão/ano em retenção

# Transformação *vs* Machine Learning

**Relacionamento entre conceitos:**

**Transformação de Dados (Esta Aula):**

- Técnicas para preparar e enriquecer dados
- Útil para *qualquer* análise quantitativa
- Análise exploratória avançada
- Visualizações melhores
- Cálculo de correlações e estatísticas

**Machine Learning (Cursos Futuros):**

- Usa *exatamente as mesmas transformações*
- Mas para construir modelos preditivos
- Objetivo: fazer previsões automáticas
- Requer dados transformados da mesma forma

# Transformação *vs* Machine Learning

**Nota Importante**

Hoje: aprenderemos transformações para análise.
Bonus: são as mesmas usadas em ML!

# Tipos de Transformações

**1. Feature Engineering (CRIAÇÃO)**

- Criar novas variáveis a partir das existentes
- Extrair informação ”escondida”
- Combinar features para revelar padrões
- Ex: TotalCharges / tenure = ”Gasto médio mensal”

**2. Encoding (CONVERSÃO)**

- Texto $\rightarrow$ números
- Necessário para análises quantitativas
- Ex: ”Sim”/”Não”$\rightarrow$ 1/0

# Tipos de Transformações

**3. Scaling (NORMALIZAÇÃO)**

- Colocar variáveis em escalas comparáveis
- Facilita interpretação e visualização
- Ex: Idade e Salário na mesma escala [0, 1]

**4. Mathematical Transform (CORREÇÃO)**

- Log, sqrt, potência
- Corrigir distribuições assimétricas
- Ex: log(salário) deixa distribuição mais normal

# Dataset de Hoje: Telco Customer Churn

**Dados de clientes de empresa de telecomunicações**

**O que é Churn?**

- Churn = Cancelamento (cliente sai da empresa)
- Métrica crítica para negócios de assinatura
- Muito caro adquirir novo cliente vs manter existente

**Objetivo da Análise:**

- **Entender:** Quais fatores estão associados ao cancelamento?
- **Comparar:** Como clientes que cancelam diferem dos que ficam?
- **Identificar:** Quais características são mais comuns em cada grupo?
- **Revelar:** Padrões não óbvios através de transformações

**Dados disponíveis:**

- 7000 clientes
- 20 variáveis (demográficas, serviços, contratos, pagamentos)
- Target: Churn (Yes/No)

# Telco Churn: Variáveis do Dataset

**Variáveis Demográficas:**

- `gender`: Gênero (Male/Female)
- `SeniorCitizen`: Idoso? (0/1)
- `Partner`: Tem parceiro? (Yes/No)
- `Dependents`: Tem dependentes? (Yes/No)

**Serviços Contratados:**

- `PhoneService`, `MultipleLines`: Telefone
- `InternetService`: DSL, Fiber optic, No
- `OnlineSecurity`, `OnlineBackup`, `DeviceProtection`, `TechSupport`: Serviços adicionais
- `StreamingTV`, `StreamingMovies`: Streaming

**Contrato e Pagamento:**

- `Contract`: Month-to-month, One year, Two year
- `PaperlessBilling`: Sem papel? (Yes/No)
- `PaymentMethod`: Método de pagamento (4 tipos)

# Telco Churn: Variáveis do Dataset (cont.)

**Variáveis Numéricas:**

- `tenure`: Meses como cliente (0-72)
- `MonthlyCharges`: Valor mensal (R$ 18-118)
- `TotalCharges`: Total gasto (R$ 18-8684)

**Target (o que queremos analisar):**

- `Churn`: Cancelou? (Yes/No)
- Aproximadamente 27% de churn
- Esta é nossa variável de interesse

**Tipos de variáveis presentes:**

- ✓ Numéricas contínuas (charges, tenure)
- ✓ Categóricas binárias (Yes/No)
- ✓ Categóricas ordinais (Contract tem ordem)
- ✓ Categóricas nominais (PaymentMethod sem ordem)

# Por que Este Dataset é Bom Exemplo?

**Requer todas as transformações que vamos aprender:**

**1. Feature Engineering**

- Podemos criar: gasto médio mensal, razão charges/tenure
- Agregações: total de serviços contratados
- Flags: cliente premium? valor alto?

**2. Encoding**

- Binárias: gender, Partner, Dependents $\rightarrow$ 0/1
- Ordinais: Contract (ordem de compromisso)
- Nominais: PaymentMethod, InternetService

# Por que Este Dataset é Bom Exemplo?

**Requer todas as transformações que vamos aprender:**

**3. Normalização**

- tenure (0-72) vs TotalCharges (18-8684)
- Escalas muito diferentes precisam normalização

**4. Fácil de Entender**

- Telecomunicações é familiar
- Variáveis têm significado claro
- Resultados são interpretáveis

# Objetivos de Aprendizado

**Ao final desta aula, você será capaz de:**

**Conhecimento:**

- Identificar quando e por que transformar dados
- Compreender diferentes tipos de encoding
- Conhecer técnicas de normalização e suas diferenças
- Entender o conceito de feature engineering

**Habilidades:**

- Criar features úteis a partir de dados existentes
- Aplicar encoding apropriado para cada tipo de variável
- Normalizar dados com Min-Max, Standardization, Robust Scaling
- Construir pipeline completo de transformação
- Validar transformações através de análise exploratória

# Objetivos de Aprendizado (cont.)

**Atitudes:**

- Pensar criativamente sobre features
- Documentar decisões de transformação
- Validar resultados sistematicamente

# Ferramentas de Hoje

**Pandas: Manipulação e transformação**

- `pd.get_dummies()`: One-Hot Encoding
- `map()`, `replace()`: Label Encoding
- `apply()`, `transform()`: Operações customizadas

**Scikit-learn: Transformações padronizadas**

- `MinMaxScaler()`: Normalização [0, 1]
- `StandardScaler()`: Padronização (Z-score)
- `RobustScaler()`: Scaling robusto a outliers
- `LabelEncoder()`: Encoding de labels

# Ferramentas de Hoje

**NumPy: Operações matemáticas**

- `np.log(), np.sqrt()`: Transformações matemáticas
- Operações vetorizadas eficientes

**Visualização:**

- Matplotlib, Seaborn
- Validar transformações visualmente

# Ordem das Operações

**Pipeline lógico de transformação:**

**1. Feature Engineering**
*Criar novas variáveis*
↓
**2. Encoding**
*Categorias → números*
↓
**3. Normalização/Scaling**
*Ajustar escalas*
↓
**4. Validação**
*Verificar resultado*

# Ordem das Operações

**⚠ Atenção**

A ordem importa! Encoding antes de scaling, por exemplo.

# Setup: Carregar Dados

**Python**

```python
# Imports necessarios
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler, StandardScaler,
    RobustScaler

# Configuracoes
pd.set_option('display.max_columns', None)
pd.set_option('display.float_format', '{:.2f}'.format)
plt.style.use('seaborn-v0_8-darkgrid')
```

# Setup: Carregar Dados (cont.)

**</> Python**

```python
# URL
url = 'https://raw.githubusercontent.com/IBM/telco-customer-churn-on-icp4d/master/data/Telco-Customer-Churn.csv'

# Carregar
df_original = pd.read_csv(url)
df = df_original.copy()
print(f"Dataset carregado: {df.shape}")
print(f"Linhas: {df.shape[0]:,} | Colunas: {df.shape[1]}")
# Dataset carregado: (7043, 21)
# Linhas: 7,043 | Colunas: 21
```

# Exploração Inicial

**Python**

```python
# Primeiras linhas
print("=== PRIMEIRAS LINHAS ===")
display(df.head())

# Informacoes gerais
print("\n=== INFO ===")
df.info()

# Estatisticas descritivas
print("\n=== ESTATISTICAS ===")
display(df.describe())
```

# Exploração Inicial (cont.)

**Python**

```python
# Distribuicao do target
print("\n=== DISTRIBUICAO CHURN ===")
print(df['Churn'].value_counts())
print(df['Churn'].value_counts(normalize=True))
# Churn: No - 0.73, Yes - 0.27
```

# Identificar Tipos de Variáveis

**</> Python**

```python
# Separar variaveis por tipo
numeric_cols = df.select_dtypes(include=['int64', 'float64']).columns.
    tolist()
categorical_cols = df.select_dtypes(include=['object']).columns.tolist()
# Remover target da lista
if 'Churn' in categorical_cols:
    categorical_cols.remove('Churn')
print(f"Numericas ({len(numeric_cols)}): {numeric_cols}")
print(f"Categoricas ({len(categorical_cols)}): {categorical_cols}")
# Analisar cardinalidade das categoricas
print("\n=== CARDINALIDADE ===")
for col in categorical_cols:
    n_unique = df[col].nunique()
    print(f"{col}: {n_unique} valores unicos")
```

# Identificar Tipos de Variáveis (cont.)

**Python**

```
# === CARDINALIDADE ===
# customerID: 7043 valores unicos
# gender: 2 valores unicos
# Partner: 2 valores unicos
# Dependents: 2 valores unicos
# PhoneService: 2 valores unicos
# MultipleLines: 3 valores unicos
# InternetService: 3 valores unicos
# OnlineSecurity: 3 valores unicos
# OnlineBackup: 3 valores unicos
# DeviceProtection: 3 valores unicos
# ...
# PaymentMethod: 4 valores unicos
# TotalCharges: 6531 valores unicos
```

# Fluxo de Transformação

**Processo sistemático que seguiremos:**

**ETAPA 1: Feature Engineering (Bloco 1)**

- Criar variáveis derivadas úteis
- Agregações e combinações
- Flags e indicadores
- Validar utilidade (correlação com Churn)

**ETAPA 2: Encoding (Bloco 2)**

- Identificar tipo de cada categórica
- Aplicar encoding apropriado
- Binárias $\rightarrow$ 0/1
- Ordinais $\rightarrow$ Label Encoding
- Nominais $\rightarrow$ One-Hot Encoding

# Fluxo de Transformação (cont.)

**ETAPA 3: Normalização (Bloco 3)**

- Identificar variáveis com escalas muito diferentes
- Escolher técnica apropriada
- Aplicar scaling
- Validar resultado

**ETAPA 4: Pipeline Final (Bloco 4)**

- Integrar todas as transformações
- Criar função reutilizável
- Documentar decisões
- Análise exploratória avançada com dados transformados

**Nota Importante**

Cada etapa será validada antes de prosseguir!

# Considerações Importantes

**1. Sempre preserve os dados originais**

- Trabalhe em cópias
- Mantenha dataset original intacto
- Facilita comparações antes/depois

**2. Documente suas decisões**

- Por que criou cada feature?
- Por que escolheu aquele encoding?
- Raciocínio claro e reproduzível

**3. Valide cada transformação**

- Verificar se faz sentido
- Analisar correlações
- Visualizar resultados
- Estatísticas descritivas

# Considerações Importantes

**4. Pense no significado do negócio**

- Features devem fazer sentido prático
- Consulte especialistas do domínio
- Interpretabilidade é importante

**5. Comece simples, evolua gradualmente**

- Transformações óbvias primeiro
- Teste uma de cada vez
- Adicione complexidade progressivamente

**6. Cuidado com ”data leakage”**

- Não use informação ”do futuro”
- Importante quando eventualmente usar ML
- Boa prática: sempre separar dados para validação

# Considerações Importantes

**⚠ Atenção**

Transformação é ciência + arte + conhecimento de domínio!

# Expectativas Realistas

**O que você conseguirá fazer hoje:**

- ✓ Criar features úteis sistematicamente
- ✓ Encodar qualquer tipo de variável categórica
- ✓ Normalizar dados para análises comparativas
- ✓ Construir pipeline completo reproduzível
- ✓ Validar transformações através de EDA

**O que NÃO faremos hoje (mas você aprenderá depois):**

- ✗ Construir modelos de Machine Learning
- ✗ Fazer previsões automáticas
- ✗ Otimizar hiperparâmetros
- ✗ Avaliar performance preditiva

# Expectativas Realistas

**Nota Importante**

Foco: Transformação para análise exploratória avançada.
Bônus: Mesmas técnicas são usadas em ML!

# Caso de Uso Motivador

**Pergunta de Negócio:**

*"Quais fatores estão mais fortemente associados ao cancelamento de clientes?"*

**Para responder, precisamos:**

1. **Enriquecer dados** com features derivadas
   - Ex: "Gasto médio mensal", "Total de serviços"
2. **Converter categorias** para análise quantitativa
   - Calcular correlações com Churn
3. **Normalizar escalas** para comparação justa
   - Identificar quais variáveis têm maior "peso"
4. **Visualizar padrões** de forma clara
   - Comparar clientes que cancelam vs que ficam

**Ao final:** Teremos insights acionáveis para reduzir churn!

# Estrutura da Aula

**Bloco 0: Preview de Machine Learning**

- Contexto: o que é ML e por que importa
- Relação entre transformação e ML

**Bloco 1: Feature Engineering**

- Criar variáveis derivadas úteis

**Bloco 2: Encoding**

- Converter categorias em números

**Bloco 3: Normalização**

- Ajustar escalas para comparabilidade

**Bloco 4: Pipeline Final**

- Integrar tudo + análise final

# Bloco 0: Preview de Machine Learning

**Por que falar de ML numa aula de transformação?**

- As transformações que aprenderemos são *essenciais* para ML
- Mas também são úteis para análise sem ML
- Importante entender o contexto mais amplo
- Ajuda a entender *por que* fazemos cada transformação

**Estrutura deste bloco:**

1. O que é Machine Learning? (conceitual)
2. Tipos de problemas de ML
3. Por que ML precisa de dados transformados?
4. Pipeline típico de Ciência de Dados
5. Transição para transformações (foco da aula)

# Bloco 0: Preview de Machine Learning

**Nota Importante**

NÃO iremos aprender ML hoje! Esta parte da aula é para contexto somente.

# O que é Machine Learning?

**Definição simples:**

*”Ensinar computadores a aprender padrões nos dados e fazer previsões automaticamente”*

**Diferença fundamental:**

**Programação tradicional:**

- Você escreve as regras explicitamente
- Ex: SE idade > 65 ENTÃO ”idoso”

**Machine Learning:**

- Computador descobre as regras automaticamente
- Você fornece exemplos (dados)
- Algoritmo aprende padrões
- Ex: Mostrar 1000 casos de churn, algoritmo aprende padrão

# Machine Learning: Analogia

**Aprender a identificar spam:**

**Abordagem tradicional (regras):**

- SE contém ”ganhe dinheiro”ENTÃO spam
- SE contém ”clique aqui”ENTÃO spam
- SE remetente desconhecido ENTÃO spam
- Problema: spammers se adaptam, regras ficam desatualizadas

**Abordagem Machine Learning:**

- Dar 10.000 exemplos de emails (spam e não-spam)
- Algoritmo aprende padrões automaticamente
- Consegue identificar spam novo que nunca viu
- Se adapta com novos dados

# Machine Learning: Analogia

**Nota Importante**

ML = Aprender com exemplos ao invés de programar regras

# Exemplos Cotidianos de ML

**Recomendações:**

- Netflix: ”Você pode gostar deste filme”
- Amazon: ”Clientes que compraram X também compraram Y”
- Spotify: Playlists personalizadas

**Reconhecimento:**

- Reconhecimento facial (desbloquear celular)
- Assistentes de voz (Alexa, Siri)
- Tradução automática (Google Translate)

**Previsões:**

- Previsão do tempo
- Detecção de fraude em cartão de crédito
- Diagnóstico médico assistido
- Risco de crédito em bancos

# Tipos de Problemas de ML

**1. Classificação (prever categoria)**

- Exemplo: Email é spam ou não-spam?
- Exemplo: Cliente vai cancelar ou não? (Churn)
- Exemplo: Doença está presente ou ausente?
- Resposta: categoria/classe

**2. Regressão (prever número)**

- Exemplo: Qual será o preço desta casa?
- Exemplo: Quantas vendas teremos no próximo mês?
- Exemplo: Qual a temperatura amanhã?
- Resposta: valor numérico contínuo

**3. Clustering (agrupar similares)**

- Exemplo: Segmentar clientes em grupos
- Exemplo: Agrupar notícias por tema
- Sem ”resposta correta”prévia

# Problema de Classificação: Detalhes

**Exemplo: Prever Churn (classificação binária)**

**Dados de entrada (features):**

- tenure = 12 meses
- MonthlyCharges = R$ 85
- Contract = ”Month-to-month”
- InternetService = ”Fiber optic”
- ... (outras variáveis)

**Saída desejada (target):**

- Churn = ”Yes”ou ”No”
- Ou: probabilidade de churn (0-100%)

**Como funciona:**

1. Treinar com exemplos históricos (já sabemos quem cancelou)
2. Algoritmo aprende padrões
3. Usar para prever novos clientes

# Problema de Regressão: Detalhes

**Exemplo: Prever Preço de Casa**

**Dados de entrada (features):**

- Área = 120 m²
- Quartos = 3
- Banheiros = 2
- Localização = ”Centro”
- Idade = 10 anos

**Saída desejada (target):**

- Preço = R$ 450.000
- Valor numérico contínuo

**Como funciona:**

1. Treinar com casas já vendidas (preço conhecido)
2. Algoritmo aprende relação features $\rightarrow$ preço
3. Usar para estimar preço de casas novas

# Por que ML Precisa de Dados Transformados?

**Problema 1: Algoritmos trabalham com números**

- ML usa matemática (álgebra linear, cálculo)
- Não consegue processar texto diretamente
- ”Month-to-month”precisa virar número
- **Solução: Encoding**

**Problema 2: Escalas diferentes confundem**

- tenure: 0-72 vs TotalCharges: 0-8684
- Algoritmo pode dar peso errado
- Variável com números maiores parece ”mais importante”
- **Solução: Normalização**

**Problema 3: Informação está escondida**

- Padrões não são óbvios nas variáveis originais
- Combinações podem ser mais úteis
- **Solução: Feature Engineering**

# Exemplo: Por que Encoding Importa

**Variável: Contract (tipo de contrato)**

- Valores: ”Month-to-month”, ”One year”, ”Two year”

**Problema:**

- Algoritmo não consegue processar texto
- Não dá para calcular: ”Month-to-month”+ ”One year”= ?

**Solução: Encoding**

- Month-to-month $\rightarrow$ 0
- One year $\rightarrow$ 1
- Two year $\rightarrow$ 2

**Agora o algoritmo consegue:**

- Processar matematicamente
- Identificar padrão: contratos mais longos = menos churn
- Fazer previsões baseadas nisso

# Exemplo: Por que Normalização Importa

**Duas variáveis importantes:**

- tenure: 0-72 (meses)
- TotalCharges: 18-8684 (reais)

**Problema:**

- TotalCharges tem números muito maiores
- Algoritmos baseados em distância ficam confusos
- Podem dar peso excessivo a TotalCharges
- Não porque é mais importante, mas porque números são maiores!

**Solução: Normalização**

- Colocar ambas na mesma escala [0, 1]
- tenure: 0-72 $\rightarrow$ 0.0-1.0
- TotalCharges: 18-8684 $\rightarrow$ 0.0-1.0
- Agora são comparáveis de forma justa

# Pipeline Típico de Ciência de Dados com ML

**Processo completo:**

**1. Coleta de Dados**
*Obter dados relevantes*
↓
**2. Exploração (EDA)**
*Entender os dados*
↓
**3. Limpeza**
*Tratar missing, outliers, inconsistências*
↓
**4. TRANSFORMAÇÃO** ← HOJE
*Feature Engineering + Encoding + Scaling*

# Pipeline Típico de Ciência de Dados com ML

**Processo completo:**

**4. TRANSFORMAÇÃO** ← HOJE
*Feature Engineering + Encoding + Scaling*
↓
**5. Modelagem (ML)**
*Treinar algoritmos, fazer previsões*
↓
**6. Avaliação**
*Validar performance, ajustar*
↓
**7. Deploy**
*Colocar em produção*

# Onde Estamos no Curso?

| Aula | Tema | Status |
|---|---|---|
| 1-2 | Python Básico | ✓ Completo |
| 3-4 | NumPy | ✓ Completo |
| 5-6 | Pandas | ✓ Completo |
| 7-8 | EDA | ✓ Completo |
| 9 | Limpeza | ✓ Completo |
| 10 | Transformação | ← HOJE |
| 11 | Séries Temporais | Futuro |
| 12 | Dados Multidimensionais | Futuro |

**Sobre Machine Learning:**

- NÃO faz parte deste curso (introdutório)
- Será visto em cursos avançados
- Mas: as técnicas de hoje são fundamentais para ML
- Você já estará preparado quando chegar lá!

# Relação: Análise vs Machine Learning

**Mesmas técnicas, objetivos diferentes:**

**Análise Exploratória (nosso foco hoje):**

- Calcular correlações entre variáveis
- Comparar grupos (clientes que cancelam vs não cancelam)
- Visualizações mais informativas
- Identificar fatores associados ao target
- Gerar insights acionáveis

**Machine Learning (cursos futuros):**

- Usar *exatamente as mesmas transformações*
- Mas para treinar modelos preditivos
- Objetivo: fazer previsões automáticas
- Aplicar em dados novos

# Exemplo Concreto: Duas Abordagens

**Problema: Entender Churn**

**Abordagem 1: Análise (hoje)**

1. Transformar dados (encoding, normalização, features)
2. Calcular correlações: qual variável mais correlacionada com Churn?
3. Visualizar: comparar perfis de quem cancela vs quem fica
4. Insight: ”Clientes com contrato mensal têm 3x mais churn”
5. Ação: Oferecer incentivos para contratos anuais

**Abordagem 2: ML (futuro)**

1. Transformar dados (*da mesma forma*)
2. Treinar modelo de classificação
3. Para cada cliente novo: prever probabilidade de churn
4. Sistema automático: se prob > 70%, acionar retenção
5. Prevenção proativa

# O que Aprenderemos Hoje

**Foco: Transformações para análise exploratória**

**Feature Engineering:**

- Criar variáveis como ”gasto médio mensal”
- Objetivo: revelar padrões não óbvios
- Validação: correlação com Churn, visualizações

**Encoding:**

- Converter categorias em números
- Objetivo: permitir análises quantitativas
- Validação: verificar se faz sentido

**Normalização:**

- Colocar variáveis em escalas comparáveis
- Objetivo: comparações justas, visualizações melhores
- Validação: estatísticas descritivas, gráficos

# O que NÃO Faremos Hoje

**Deixando claro (para não gerar expectativas erradas):**

**NÃO faremos:**

- ✗ Treinar modelos de Machine Learning
- ✗ Fazer previsões automáticas
- ✗ Usar algoritmos de classificação/regressão
- ✗ Avaliar métricas como accuracy, precision, recall
- ✗ Split train/test para validação de modelo
- ✗ Otimização de hiperparâmetros

**Por quê?**

- Este é um curso introdutório
- ML requer curso dedicado (muitos conceitos novos)
- Hoje: fundação essencial (transformação de dados)
- Futuro: você usará essas mesmas técnicas em ML

# Contexto

**O que sabemos:**

- ✓ ML aprende padrões em dados automaticamente
- ✓ ML precisa de dados bem transformados
- ✓ As transformações também melhoram análises sem ML
- ✓ Vamos focar em análise exploratória

**Próximos passos:**

- **Bloco 1:** Feature Engineering (criar variáveis úteis)
- **Bloco 2:** Encoding (categorias $\rightarrow$ números)
- **Bloco 3:** Normalização (ajustar escalas)
- **Bloco 4:** Pipeline final + análise avançada

**Método de validação:**

- Após cada transformação: análise exploratória
- Visualizações, correlações, estatísticas
- Verificar se faz sentido para o negócio

# Bloco 1

Feature Engineering

# O que vamos aprender

**Conceito:**

- O que é feature engineering
- Por que criar novas variáveis
- Tipos de features que podemos criar

**Prática:**

- 12 técnicas diferentes com código
- Aplicadas ao dataset Telco Churn
- Validação através de análise exploratória

**Objetivo:**

- Revelar padrões escondidos nos dados
- Criar variáveis mais interpretáveis
- Melhorar análises e visualizações

# O que é Feature Engineering?

**Definição:**

*”Processo de criar novas variáveis (features) a partir das variáveis existentes no dataset”*

**Por que fazer isso?**

- Informação útil está ”escondida”nos dados brutos
- Combinações de variáveis revelam padrões
- Algumas relações não são óbvias
- Features derivadas podem ser mais interpretáveis

**Exemplo simples:**

- Temos: `TotalCharges` e `tenure`
- Criamos: `AvgMonthlySpend = TotalCharges / tenure`
- Por quê? Gasto médio mensal é mais interpretável
- Facilita comparações entre clientes

# Por que Feature Engineering Importa?

**Impacto na qualidade da análise:**

**Sem feature engineering:**

- Análise limitada às variáveis originais
- Padrões importantes podem passar despercebidos
- Comparações são difíceis
- Visualizações menos informativas

**Com feature engineering:**

- Padrões ocultos são revelados
- Comparações mais justas e interpretáveis
- Visualizações mais ricas
- Insights mais profundos
- Recomendações mais embasadas

# Por que Feature Engineering Importa?

**Nota Importante**

Boas features = Diferença entre análise superficial e insights profundos

# Tipos de Feature Engineering

**12 técnicas que aprenderemos:**

1. Operações aritméticas simples (+, -, *, /)
2. Diferenças entre variáveis
3. Razões e proporções
4. Binning (discretização)
5. Agregações (somas, contagens)
6. Flags booleanas (indicadores 0/1)
7. Interações entre variáveis
8. Extração de informação de strings
9. Features temporais (conceitual)
10. Percentis e rankings
11. Z-score para detecção de anomalias
12. Features baseadas em conhecimento do domínio

# Tipos de Feature Engineering

**Para cada técnica:**

- Exemplo prático com código
- Validação através de análise exploratória
- Interpretação do resultado

# Feature Engineering: Método sistemático

**1. PENSAR**
*Que informação seria útil?*
↓
**2. CRIAR**
*Implementar a feature*
↓
**3. VALIDAR**
*Verificar se faz sentido*
↓
**4. ANALISAR**
*Correlação, visualização*
↓
**5. DECIDIR**
*Manter ou descartar*

# Feature 1: Operações Aritméticas Simples

## Gasto médio mensal

**</> Python**

```python
# Feature: Average Monthly Spend
# Raciocínio: Clientes com gasto médio alto podem ter
# comportamento diferente
df['TotalCharges'] = pd.to_numeric(df['TotalCharges'], errors='coerce')
df['AvgMonthlySpend'] = df['TotalCharges'] / df['tenure']
# Tratar casos especiais
# Se tenure = 0, evitar divisão por zero
df['AvgMonthlySpend'] = df['AvgMonthlySpend'].replace([np.inf, -np.inf],
    np.nan)
# Para tenure = 0, usar MonthlyCharges como aproximação
df.loc[df['tenure'] == 0, 'AvgMonthlySpend'] = df.loc[df['tenure'] == 0, '
    MonthlyCharges']
print("=== FEATURE CRIADA: AvgMonthlySpend ===")
print(df['AvgMonthlySpend'].describe())
```

# Por que AvgMonthlySpend é Útil?

**Validação através de análise:**

**</> Python**

```python
# Comparar entre clientes que cancelam e que ficam
print("=== COMPARACAO POR CHURN ===")
print(df.groupby('Churn')['AvgMonthlySpend'].describe())

# Visualizar distribuicao
import matplotlib.pyplot as plt
import seaborn as sns

fig, ax = plt.subplots(1, 2, figsize=(14, 5))

# Boxplot
sns.boxplot(data=df, x='Churn', y='AvgMonthlySpend', ax=ax[0])
ax[0].set_title('Gasto Médio Mensal por Churn')
```

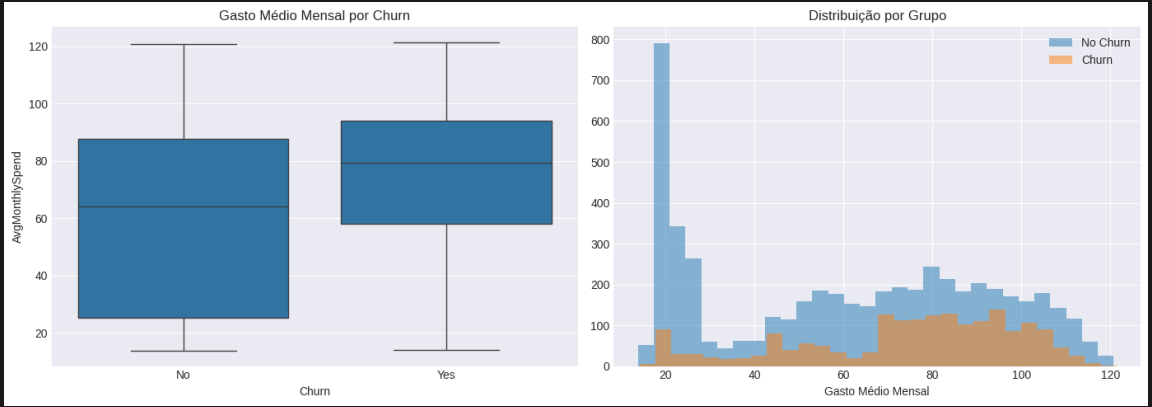
# Por que AvgMonthlySpend é Útil? (cont.)

**Validação através de análise:**

**Python**

```python
# Histograma
df[df['Churn']=='No']['AvgMonthlySpend'].hist(bins=30, alpha=0.5, label='No Churn', ax=ax[1])
df[df['Churn']=='Yes']['AvgMonthlySpend'].hist(bins=30, alpha=0.5, label='Churn', ax=ax[1])
ax[1].set_xlabel('Gasto Médio Mensal')
ax[1].set_title('Distribuição por Grupo')
ax[1].legend()

plt.tight_layout()
plt.show()
```

# Por que AvgMonthlySpend é Útil? (cont.)



# Feature 2: Diferença entre Variáveis

## Variação no gasto

**</> Python**

```python
# Feature: Charge Difference
# Raciocínio: Se MonthlyCharges >> AvgMonthlySpend,
# pode indicar aumento recente (cliente insatisfeito?)
df['ChargeDiff'] = df['MonthlyCharges'] - df['AvgMonthlySpend']
print("=== FEATURE CRIADA: ChargeDiff ===")
print(df['ChargeDiff'].describe())
# Interpretar
print("\n=== INTERPRETACAO ===")
print("ChargeDiff > 0: Gasto atual maior que média (aumento recente)")
print("ChargeDiff < 0: Gasto atual menor que média (diminuição)")
print("ChargeDiff  0: Gasto estável")
# Analisar por churn
print("\n=== MEDIA POR CHURN ===")
print(df.groupby('Churn')['ChargeDiff'].mean())
```

# Feature 3: Razões/Proporções

## Proporção do tempo como cliente

**</> Python**

```python
# Feature: Tenure Ratio
# Raciocínio: Normalizar tenure em relação ao máximo possível
# Facilita comparações

max_tenure = df['tenure'].max()
df['TenureRatio'] = df['tenure'] / max_tenure

print("=== FEATURE CRIADA: TenureRatio ===")
print(f"Range: [{df['TenureRatio'].min():.2f}, {df['TenureRatio'].max():.2
    f}]")
```

# Feature 3: Razões/Proporções (cont.)

## Proporção do tempo como cliente

**</> Python**

```python
# Criar categorias
df['TenureCategory'] = pd.cut(df['TenureRatio'],
                              bins=[0, 0.33, 0.66, 1.0],
                              labels=['Novo', 'Médio', 'Veterano'])

print("\n=== DISTRIBUICAO POR CATEGORIA ===")
print(pd.crosstab(df['TenureCategory'], df['Churn'], normalize='index'))
```

# Feature 4: Binning (Discretização)

**Transformar numérica contínua em categórica ordinal**

**</> Python**

```python
# Feature: MonthlyCharges em faixas
# Raciocínio: Facilita análise por grupos de preço

# Método 1: Bins com larguras iguais
df['ChargesBin_Equal'] = pd.cut(df['MonthlyCharges'],
                                bins=5,
                                labels=['Muito Baixo', 'Baixo', 'Médio',
                                        'Alto', 'Muito Alto'])
```

# Feature 4: Binning (Discretização) (cont.)

**Transformar numérica contínua em categórica ordinal**

**</> Python**

```python
# Método 2: Quantis (mesma quantidade em cada bin)
df['ChargesBin_Quantile'] = pd.qcut(df['MonthlyCharges'],
                                    q=5,
                                    labels=['Q1', 'Q2', 'Q3', 'Q4', 'Q5'
    ])

print("=== BINNING CRIADO ===")
print("\nDistribuição - Bins Iguais:")
print(df['ChargesBin_Equal'].value_counts().sort_index())
```

# Feature 4: Binning (Discretização) (cont.)

**Python**

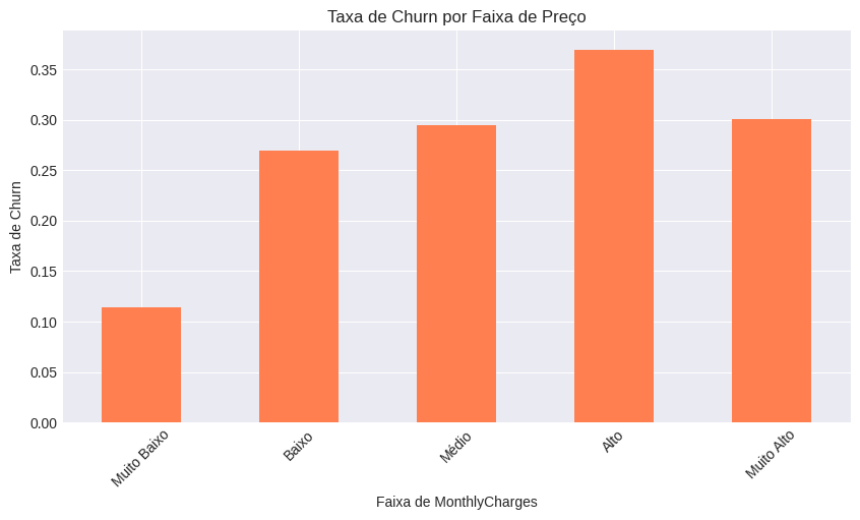
```python
print("\nDistribuição - Quantis:")
print(df['ChargesBin_Quantile'].value_counts().sort_index())

# Analisar churn por faixa de preço
print("\n=== CHURN RATE POR FAIXA ===")
churn_by_bin = df.groupby('ChargesBin_Equal', observed=False)['Churn'].
    apply(
    lambda x: (x == 'Yes').mean()
)
print(churn_by_bin)
```

# Feature 4: Binning (Discretização) (cont.)

**Python**

```python
# Visualizar
churn_by_bin.plot(kind='bar', color='coral', figsize=(10, 5))
plt.title('Taxa de Churn por Faixa de Preço')
plt.ylabel('Taxa de Churn')
plt.xlabel('Faixa de MonthlyCharges')
plt.xticks(rotation=45)
plt.show()
```

# Feature 4: Binning (Discretização) (cont.)



# Por que Binning é Útil?

**Vantagens da discretização:**

**1. Interpretabilidade**
- ”Clientes de preço alto”é mais claro que ”R$ 89,47”
- Facilita comunicação com não-técnicos
- Categorias são intuitivas

**2. Análise por grupos**
- Comparar comportamento entre faixas
- Identificar thresholds importantes
- Ex: ”Churn dispara acima de R$ 70/mês”

**3. Lidar com não-linearidade**
- Relação pode não ser linear
- Ex: churn alto em valores muito baixos E muito altos
- Bins capturam melhor esses padrões

# Feature 5: Agregação - Contar Serviços

**Total de serviços adicionais contratados**

**</> Python**

```python
# Feature: Total Services
# Raciocínio: Clientes com mais serviços podem estar
# mais "engajados" (menos propensos a cancelar?)

# Lista de colunas de serviços
service_cols = [
    'PhoneService', 'MultipleLines',
    'InternetService', 'OnlineSecurity',
    'OnlineBackup', 'DeviceProtection',
    'TechSupport', 'StreamingTV', 'StreamingMovies'
]
```

# Feature 5: Agregação - Contar Serviços (cont.)

**Total de serviços adicionais contratados**

**</> Python**

```python
# Contar quantos serviços tem "Yes" ou não é "No"
df['TotalServices'] = 0

for col in service_cols:
    # Adiciona 1 se não for "No" ou "No internet service" ou "No phone service"
    df['TotalServices'] += (~df[col].isin(['No', 'No internet service', 'No phone service'])).astype(int)
```

# Feature 5: Agregação - Contar Serviços (cont.)

**Python**

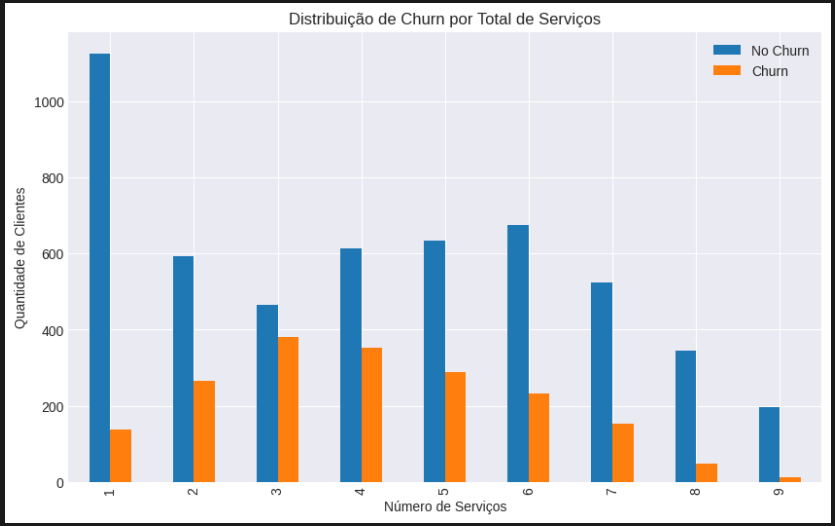
```python
print("=== FEATURE CRIADA: TotalServices ===")
print(df['TotalServices'].describe())

print("\n=== DISTRIBUICAO ===")
print(df['TotalServices'].value_counts().sort_index())

# Analisar relação com Churn
print("\n=== MEDIA DE SERVICOS POR CHURN ===")
print(df.groupby('Churn')['TotalServices'].mean())
```

# Feature 5: Agregação - Contar Serviços (cont.)

**Python**

```python
# Visualizar
fig, ax = plt.subplots(figsize=(10, 6))
df.groupby(['TotalServices', 'Churn']).size().unstack().plot(kind='bar',
    ax=ax)
ax.set_title('Distribuição de Churn por Total de Serviços')
ax.set_xlabel('Número de Serviços')
ax.set_ylabel('Quantidade de Clientes')
plt.legend(['No Churn', 'Churn'])
plt.show()
```

# Feature 5: Agregação - Contar Serviços (cont.)



# Agregação: Versão Otimizada

**Método mais eficiente com apply:**

**</> Python**

```python
# Alternativa: usar apply (mais eficiente)
def count_services(row):
    """Conta serviços ativos excluindo 'No' e variações"""
    count = 0
    for col in service_cols:
        value = str(row[col]).lower()
        if value not in ['no', 'no internet service', 'no phone service']:
            count += 1
    return count

df['TotalServices'] = df.apply(count_services, axis=1)
```

# Agregação: Versão Otimizada (cont.)

**Método mais eficiente com apply:**

**Python**

```python
# Ou ainda mais simples com sum:
df['TotalServices'] = df[service_cols].apply(
    lambda x: (~x.isin(['No', 'No internet service',
                        'No phone service'])).sum(),
    axis=1
)

print("Total de serviços calculado!")
print(df['TotalServices'].value_counts())
```

# Feature 6: Flags Booleanas

## Indicadores binários úteis

**</> Python**

```python
# Feature: Has Internet
df['HasInternet'] = (df['InternetService'] != 'No').astype(int)
# Feature: Has Phone
df['HasPhone'] = (df['PhoneService'] == 'Yes').astype(int)
# Feature: Senior with Dependents (combinação interessante)
df['SeniorWithDependents'] = (
    (df['SeniorCitizen'] == 1) &
    (df['Dependents'] == 'Yes')
).astype(int)
# Feature: Premium Customer (muitos serviços + contrato longo)
df['IsPremium'] = (
    (df['TotalServices'] >= 6) &
    (df['Contract'].isin(['One year', 'Two year']))
).astype(int)
```

# Flags: Por que São Úteis?

**Validando as flags criadas:**

**Python**

```python
# Analisar cada flag
flags = ['HasInternet', 'HasPhone', 'SeniorWithDependents', 'IsPremium']

print("=== ANALISE DE FLAGS ===\n")
for flag in flags:
    print(f"\n--- {flag} ---")

    # Distribuição
    print(f"Total com flag=1: {df[flag].sum()} ({df[flag].mean()*100:.1f
    }%)")
```

# Flags: Por que São Úteis?

**Validando as flags criadas:**

**Python**

```python
# Taxa de churn
churn_rate = df.groupby(flag)['Churn'].apply(
    lambda x: (x == 'Yes').mean()
)
print("Taxa de Churn:")
print(churn_rate)

# Diferença
diff = churn_rate[1] - churn_rate[0]
print(f"Diferença: {diff*100:.1f} pontos percentuais")
```

# Interações entre Variáveis

**O que são interações?**

- Efeito combinado de duas variáveis
- Uma variável modifica o efeito da outra
- Relação não é simplesmente aditiva

**Exemplo conceitual:**

- Variável A: Preço alto
- Variável B: Contrato curto
- Isoladamente: Ambos aumentam churn
- **Interação:** Combinação (preço alto + contrato curto) aumenta MUITO mais
- Efeito não é A + B, é A * B

**Como criar:**

- Multiplicação de variáveis
- Combinação de flags
- Categorias compostas

# Quando combinar variáveis (interações)?

**Sinais de que interação pode ser útil:**

**1. Conhecimento do domínio sugere**
- ”Idosos com dependentes”podem ter padrão específico
- ”Preço alto + contrato curto”é combinação arriscada
- Especialista identifica combinações relevantes

**2. Análise exploratória revela**
- Crosstabs mostram padrões interessantes
- Ex: Churn é 80% quando A=high E B=yes
- Mas apenas 20% quando só A=high ou só B=yes

**3. Hipóteses de negócio**
- ”Clientes jovens com múltiplas linhas têm perfil específico”
- ”Serviço de fibra + streaming indica uso intenso”
- Testar hipóteses com interações

# Quando NÃO combinar variáveis?

**Evite interações desnecessárias:**

**1. Sem justificativa clara**

- Não combine aleatoriamente
- ”Gênero * PaymentMethod”provavelmente não faz sentido
- Precisa haver raciocínio de negócio

**2. Variáveis já correlacionadas**

- TotalCharges * tenure é redundante (TC já é função de tenure)
- MonthlyCharges * AvgMonthlySpend é circular

**3. Explosão combinatória**

- Cuidado com variáveis de alta cardinalidade
- PaymentMethod (4) * InternetService (3) = 12 combinações
- Pode criar muitas features sem valor

# Feature 7: Interações entre Variáveis

**Criando interações úteis:**

**</> Python**

```python
# Interação 1: Contrato Curto + Preço Alto
# Hipótese: Combinação perigosa para churn
df['ShortContract_HighPrice'] = (
    (df['Contract'] == 'Month-to-month') &
    (df['MonthlyCharges'] > df['MonthlyCharges'].quantile(0.75))
).astype(int)

# Interação 2: Sem segurança + Idoso
# Hipótese: Idosos sem proteção podem estar insatisfeitos
df['Senior_NoSecurity'] = (
    (df['SeniorCitizen'] == 1) &
    (df['OnlineSecurity'] == 'No')
).astype(int)
```

# Feature 7: Interações entre Variáveis (cont.)

**Criando interações úteis:**

**</> Python**

```python
# Interação 3: Fibra + Streaming (usuário power)
df['FiberStreamer'] = (
    (df['InternetService'] == 'Fiber optic') &
    ((df['StreamingTV'] == 'Yes') | (df['StreamingMovies'] == 'Yes'))
).astype(int)
```

# Feature 7: Interações entre Variáveis (cont.)

**Validando as interações:**

**Python**

```python
# Analisar impacto das interações
interactions = ['ShortContract_HighPrice', 'Senior_NoSecurity', '
    FiberStreamer']

print("=== ANALISE DE INTERACOES ===\n")
for interaction in interactions:
    print(f"\n--- {interaction} ---")

    # Crosstab com Churn
    ct = pd.crosstab(df[interaction], df['Churn'], normalize='index')
    print(ct)
```

# Feature 7: Interações entre Variáveis (cont.)

**Validando as interações:**

**</> Python**

```python
    # Taxa de churn
    churn_rate = df[df[interaction] == 1]['Churn'].apply(
        lambda x: 1 if x == 'Yes' else 0
    ).mean()

    overall_rate = (df['Churn'] == 'Yes').mean()

    print(f"\nChurn Rate com {interaction}=1: {churn_rate*100:.1f}%")
    print(f"Churn Rate geral: {overall_rate*100:.1f}%")
    print(f"Diferença: {(churn_rate - overall_rate)*100:.1f} p.p.")
```

# Feature 7: Interações entre Variáveis (cont.)

**Validando as interações:**

**Python**

```python
# ...
# Churn Rate com FiberStreamer=1: 39.9%
# Churn Rate geral: 26.5%
# Diferença: 13.4 p.p.
```

# Feature 8: Extrair de Strings

**Extrair informação útil de textos:**

**Python**

```python
# Feature: Payment Type Category
# Extrair categoria principal do método de pagamento
def categorize_payment(payment_method):
    """Categorizar método de pagamento"""
    if pd.isna(payment_method):
        return 'Unknown'
    payment = str(payment_method).lower()
    if 'electronic' in payment or 'bank' in payment:
        return 'Automatic'
    elif 'credit' in payment or 'check' in payment:
        return 'Manual'
    else:
        return 'Other'
```

# Feature 8: Extrair de Strings (cont.)

**Extrair informação útil de textos:**

**</> Python**

```python
df['PaymentCategory'] = df['PaymentMethod'].apply(categorize_payment)
print("=== CATEGORIAS DE PAGAMENTO ===")
print(df['PaymentCategory'].value_counts())
```

# Feature 9: Features Temporais (conceitual)

**Conceito: Extrair informação de timestamps**

**</> Python**

```python
# EXEMPLO CONCEITUAL (dataset não tem datas reais)
# ESTE CÓDIGO NÃO VAI FUNCIONAR NO COLAB
# Se tivéssemos uma coluna 'SignupDate':

# df['SignupDate'] = pd.to_datetime(df['SignupDate'])

# Extrair componentes temporais
# df['SignupYear'] = df['SignupDate'].dt.year
# df['SignupMonth'] = df['SignupDate'].dt.month
# df['SignupDayOfWeek'] = df['SignupDate'].dt.dayofweek
# df['SignupQuarter'] = df['SignupDate'].dt.quarter
```

# Feature 9: Features Temporais (conceitual) (cont.)

**Conceito: Extrair informação de timestamps**

**</> Python**

```python
# EXEMPLO CONCEITUAL (dataset não tem datas reais)
# ESTE CÓDIGO NÃO VAI FUNCIONAR NO COLAB

# Features derivadas
# df['DaysSinceSignup'] = (pd.Timestamp.now() - df['SignupDate']).dt.days
# df['IsWeekend'] = df['SignupDayOfWeek'].isin([5, 6]).astype(int)
# df['IsEndOfMonth'] = (df['SignupDate'].dt.day > 25).astype(int)

# Sazonalidade
# df['IsSummerSignup'] = df['SignupMonth'].isin([12, 1, 2]).astype(int)
```

# Feature 9: Features Temporais (conceitual) (cont.)

**Usando 'tenure' como proxy temporal:**

**Python**

```python
# Já que temos tenure (meses), podemos extrair padrões temporais
# Converter tenure em anos / Categorizar por período
df['TenureYears'] = (df['tenure'] / 12).astype(int)
def categorize_tenure(months):
    if months <= 6:
        return 'Very New'
    elif months <= 12:
        return 'New'
    elif months <= 24:
        return 'Established'
    elif months <= 48:
        return 'Long-term'
    else:
        return 'Veteran'
```

# Feature 9: Features Temporais (conceitual) (cont.)

**Usando 'tenure' como proxy temporal:**

**Python**

```python
df['TenurePeriod'] = df['tenure'].apply(categorize_tenure)

print("=== DISTRIBUICAO POR PERIODO ===")
print(df['TenurePeriod'].value_counts())
```

# Features Temporais: Importância

**Por que features temporais importam?**

**Padrões temporais revelam insights:**

- Sazonalidade: churn pode aumentar em certos meses
- Dia da semana: comportamento diferente fim de semana
- Tempo decorrido: ”primeiros 3 meses”são críticos
- Tendências: mudanças ao longo do tempo

**Exemplos de aplicação:**

- E-commerce: compras aumentam em novembro/dezembro
- Telecom: cancelamentos após fim de promoção
- Bancos: transações suspeitas à noite
- Saúde: internações aumentam no inverno

# Features Temporais: Importância

**Nota Importante**

Sempre que houver timestamp, extraia componentes temporais!

# Feature 10: Percentis e Rankings

**Posição relativa no dataset:**

**</> Python**

```python
# Feature: Rank de TotalCharges
# Útil para identificar clientes extremos

# Percentil (0-100)
df['TotalCharges_Percentile'] = df['TotalCharges'].rank(pct=True) * 100

# Rank absoluto
df['TotalCharges_Rank'] = df['TotalCharges'].rank(ascending=False)

# Categorizar por percentil
df['ChargesLevel'] = pd.cut(df['TotalCharges_Percentile'],
                            bins=[0, 25, 50, 75, 100],
                            labels=['Bottom 25%', 'Q2', 'Q3', 'Top 25%'])
```

# Feature 10: Percentis e Rankings (cont.)

**Posição relativa no dataset:**

**</> Python**

```python
print("=== DISTRIBUICAO POR NIVEL ===")
print(df['ChargesLevel'].value_counts())

# Analisar churn por nível
print("\n=== CHURN POR NIVEL ===")
print(pd.crosstab(df['ChargesLevel'], df['Churn'], normalize='index'))
```

# Feature 11: Z-Score para Detecção de Anomalias

**Identificar valores extremos (outliers):**

**</> Python**

```python
# Feature: Z-score de MonthlyCharges
# Identifica clientes com gastos muito atípicos

from scipy import stats

# Calcular Z-score
df['MonthlyCharges_Zscore'] = stats.zscore(df['MonthlyCharges'])

# Flag para outliers (|Z| > 3)
df['IsOutlier_Charges'] = (abs(df['MonthlyCharges_Zscore']) > 3).astype(
    int)
```

# Feature 11: Z-Score para Detecção de Anomalias

**Identificar valores extremos (outliers):**

**</> Python**

```python
print("=== OUTLIERS IDENTIFICADOS ===")
print(f"Total de outliers: {df['IsOutlier_Charges'].sum()}")
print(f"Percentual: {df['IsOutlier_Charges'].mean()*100:.2f}%")

# Analisar outliers
print("\n=== CHURN ENTRE OUTLIERS ===")
print(pd.crosstab(df['IsOutlier_Charges'], df['Churn'], normalize='index')
    )
```

# Feature 12: Conhecimento do Domínio

**Features que requerem expertise do negócio:**

**</> Python**

```python
# Feature: High Risk Profile
# Baseado em conhecimento de que certos perfis cancelam mais

df['HighRiskProfile'] = (
    (df['Contract'] == 'Month-to-month') &  # Sem compromisso
    (df['tenure'] < 12) &                   # Cliente novo
    (df['TotalServices'] <= 2) &            # Poucos serviços
    (df['PaymentMethod'] == 'Electronic check')  # Método menos estável
).astype(int)

# Feature: Value Customer
# Perfil de cliente valioso (menos propenso a sair)
```

# Feature 12: Conhecimento do Domínio (cont.)

**Features que requerem expertise do negócio:**

**</> Python**

```python
df['ValueCustomer'] = (
    (df['Contract'].isin(['One year', 'Two year'])) &  # Contrato longo
    (df['TotalServices'] >= 4) &                       # Muitos serviços
    (df['tenure'] > 24) &                              # Cliente antigo
    (df['MonthlyCharges'] > df['MonthlyCharges'].median())  # Gasto alto
).astype(int)
```

# Feature 12: Conhecimento do Domínio (cont.)

**Validando features baseadas em domínio:**

**</> Python**

```python
# Analisar profiles criados
profiles = ['HighRiskProfile', 'ValueCustomer']

print("=== ANALISE DE PROFILES ===\n")
for profile in profiles:
    print(f"\n--- {profile} ---")

    # Quantidade
    count = df[profile].sum()
    pct = df[profile].mean() * 100
    print(f"Total: {count} ({pct:.1f}%)")
```

# Feature 12: Conhecimento do Domínio (cont.)

**Validando features baseadas em domínio:**

**</> Python**

```python
    # Churn rate
    churn_rate = df[df[profile] == 1]['Churn'].apply(
        lambda x: 1 if x == 'Yes' else 0
    ).mean()

    print(f"Churn Rate: {churn_rate*100:.1f}%")

    # Comparar com baseline
    baseline = (df['Churn'] == 'Yes').mean()
    print(f"Baseline: {baseline*100:.1f}%")
    print(f"Diferença: {(churn_rate - baseline)*100:.1f} p.p.")
```

# Conhecimento do Domínio: Essencial

**Por que expertise importa:**

**Dados sozinhos não bastam:**

- Padrões estatísticos ≠ causação
- Correlação espúria é comum
- Contexto de negócio valida hipóteses

**Como obter conhecimento do domínio:**

- Conversar com especialistas
- Estudar literatura do setor
- Analisar casos históricos
- Testar hipóteses de negócio
- Iterar com stakeholders

# Conhecimento do Domínio: Essencial

**Por que expertise importa:**

**Melhores features vêm de:**

- 50% conhecimento dos dados
- 50% conhecimento do negócio

**Nota Importante**

Cientista de dados efetivo = Técnica + Domínio!

# Validar Features: Correlação com Target

**Quais features são mais úteis?**

**</> Python**

```python
# Converter Churn para binário
df['Churn_binary'] = (df['Churn'] == 'Yes').astype(int)

# Selecionar apenas features numéricas
numeric_features = df.select_dtypes(include=['int64', 'float64']).columns.
    tolist()

# Remover colunas que não são features
exclude = ['Churn_binary', 'customerID']
numeric_features = [f for f in numeric_features if f not in exclude]
```

# Validar Features: Correlação com Target (cont.)

**Quais features são mais úteis?**

**Python**

```python
# Calcular correlações com Churn
correlations = df[numeric_features + ['Churn_binary']].corr()['Churn_binary'].drop('Churn_binary')

# Ordenar por valor absoluto
correlations_abs = correlations.abs().sort_values(ascending=False)

print("=== TOP 15 FEATURES MAIS CORRELACIONADAS ===")
print(correlations_abs.head(15))
```

# Visualizar Importância das Features

**Python**

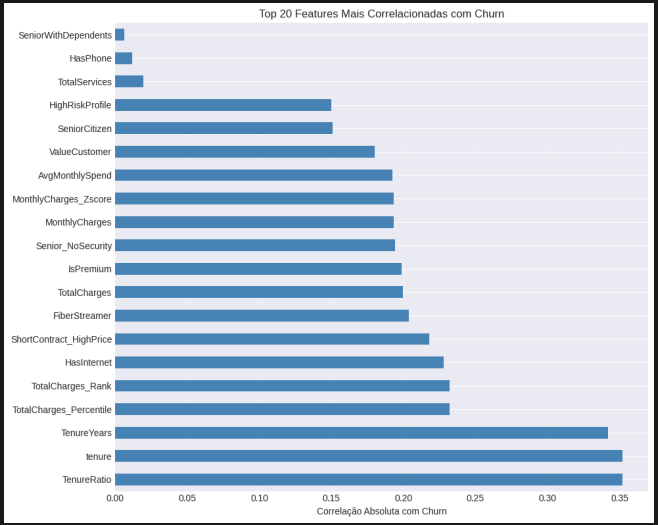
```python
# Plotar correlações
import matplotlib.pyplot as plt
import seaborn as sns

fig, ax = plt.subplots(figsize=(10, 8))

# Top 20 features
top_features = correlations_abs.head(20)

# Barplot horizontal
top_features.plot(kind='barh', ax=ax, color='steelblue')
ax.set_xlabel('Correlação Absoluta com Churn')
ax.set_title('Top 20 Features Mais Correlacionadas com Churn')
ax.grid(True, alpha=0.3, axis='x')
```

# Visualizar Importância das Features (cont.)

**Python**

```python
plt.tight_layout()
plt.show()

# Insights
print("\n=== INSIGHTS ===")
print("- Features criadas aparecem no top?")
print("- São melhores que features originais?")
print("- Faz sentido do ponto de vista de negócio?")
```

# Visualizar Importância das Features (cont.)



# Boas Práticas de Feature Engineering

**1. Documente TUDO**

- Raciocínio de cada feature
- Como foi calculada
- Por que acredita que ajuda
- Código comentado

**2. Mantenha simplicidade**

- Comece com features óbvias
- Adicione complexidade gradualmente
- Teste impacto incremental
- Descarte features que não agregam

# Boas Práticas de Feature Engineering (cont.)

**3. Validação contínua**

- Verifique correlação com target
- Análise visual (distribuições, boxplots)
- Faz sentido para o negócio?
- Não adicione features ”porque sim”

**4. Atenção à interpretabilidade**

- Features devem ser explicáveis
- Stakeholders precisam entender
- Muito complexo = difícil validar
- Preferir clareza a sofisticação

# Boas Práticas de Feature Engineering (cont.)

**5. Cuidado com leakage**

- Não use informação ”do futuro”
- Features devem ser calculáveis em produção
- Ex: não use ”cancelou_mes_seguinte”para prever churn atual

**6. Itere e refine**

- Feature engineering é iterativo
- Primeira versão raramente é perfeita
- Aprenda com análises
- Descarte o que não funciona

# Resumo: Features Criadas

**Total de features novas no dataset:**

**Python**

```python
# Listar todas as features criadas
created_features = [
    'AvgMonthlySpend', 'ChargeDiff', 'TenureRatio',
    'ChargesBin_Equal', 'ChargesBin_Quantile',
    'TotalServices', 'HasInternet', 'HasPhone',
    'SeniorWithDependents', 'IsPremium',
    'ShortContract_HighPrice', 'Senior_NoSecurity', 'FiberStreamer',
    'PaymentCategory', 'TenureYears', 'TenurePeriod',
    'TotalCharges_Percentile', 'ChargesLevel',
    'MonthlyCharges_Zscore', 'IsOutlier_Charges',
    'HighRiskProfile', 'ValueCustomer'
]
```

# Resumo: Features Criadas (cont.)

**Total de features novas no dataset:**

**</> Python**

```python
print(f"=== TOTAL DE FEATURES CRIADAS: {len(created_features)} ===")
print("\nColunas no dataset:")
print(f"Original: {len(df_original.columns)}")
print(f"Atual: {len(df.columns)}")
print(f"Features adicionadas: {len(df.columns) - len(df_original.columns)}
    ")
# === TOTAL DE FEATURES CRIADAS: 22 ===
#
# Colunas no dataset:
# Original: 21
# Atual: 46
# Features adicionadas: 25
```

# Bloco 2

Encoding de Variáveis Categóricas

# Bloco 2: Encoding de Variáveis Categóricas

**Conceito:**

- Por que precisamos converter categorias em números
- Tipos de variáveis categóricas
- Estratégias diferentes para cada tipo

**Técnicas:**

- Label Encoding (variáveis ordinais)
- One-Hot Encoding (variáveis nominais)
- Binary Encoding (variáveis binárias)
- Estratégias para alta cardinalidade

**Objetivo:**

- Permitir análises quantitativas
- Calcular correlações
- Criar visualizações numéricas
- Preparar dados para comparações

# Por que Encoding Importa?

**Limitações de variáveis categóricas (texto):**

**O que NÃO podemos fazer com texto:**

- ✗ Calcular correlações
- ✗ Fazer operações matemáticas
- ✗ Criar certos tipos de visualização
- ✗ Comparar quantitativamente
- ✗ Usar em análises estatísticas avançadas

**O que PODEMOS fazer após encoding:**

- ✓ Correlacionar com outras variáveis
- ✓ Análises estatísticas completas
- ✓ Heatmaps de correlação
- ✓ Comparações numéricas diretas
- ✓ Normalização e padronização

# Por que Encoding Importa?

**Nota Importante**

Encoding = Traduzir categorias para linguagem numérica

# Tipos de Variáveis Categóricas

**Importante: Tipo determina estratégia de encoding!**

**1. Nominais (sem ordem natural):**
- Exemplos: cor, cidade, método de pagamento
- Valores são apenas ”diferentes”, não ”maiores”ou ”menores”
- Estratégia: One-Hot Encoding

**2. Ordinais (com ordem natural):**
- Exemplos: educação (fundamental < médio < superior)
- Valores têm hierarquia clara
- Estratégia: Label Encoding (preservar ordem)

**3. Binárias (apenas 2 valores):**
- Exemplos: Yes/No, True/False, Male/Female
- Caso especial (mais simples)
- Estratégia: 0/1 direto

# Identificar Tipo no Dataset Telco Churn

**Classificando nossas variáveis:**

**Binárias (Yes/No):**

- gender, Partner, Dependents, PhoneService
- PaperlessBilling, SeniorCitizen

**Ordinais (têm ordem):**

- **Contract:** Month-to-month < One year < Two year
- (Ordem de compromisso/duração)

**Nominais (sem ordem):**

- **PaymentMethod:** Electronic check, Mailed check, Bank transfer, Credit card
- **InternetService:** DSL, Fiber optic, No
- **MultipleLines, OnlineSecurity, etc:** Yes, No, No service

# Label Encoding: Para Ordinais

**Quando usar:**

- Variável tem ordem natural clara
- Queremos preservar essa ordem
- Exemplos: níveis de educação, tamanho (S, M, L), satisfação (baixa, média, alta)

**Como funciona:**

- Atribuir números inteiros seguindo a ordem
- 0, 1, 2, 3, ...
- Ordem numérica = ordem categórica

# Label Encoding: Para Ordinais

**Exemplo - Contract:**

| Original | Encoded |
|---|---|
| Month-to-month | 0 |
| One year | 1 |
| Two year | 2 |

**Interpretação:** 2 > 1 > 0 reflete compromisso crescente

# Label Encoding: Implementação

## Método 1: Mapeamento manual (recomendado para ordinais)

**</> Python**

```python
# Criar dicionário de mapeamento
contract_mapping = {
    'Month-to-month': 0,
    'One year': 1,
    'Two year': 2
}

# Aplicar mapeamento
df['Contract_encoded'] = df['Contract'].map(contract_mapping)
```

# Label Encoding: Implementação (cont.)

## Método 1: Mapeamento manual (recomendado para ordinais)

**</> Python**

```python
# Verificar resultado
print("=== LABEL ENCODING: Contract ===")
print("\nValores originais e encodados:")
print(df[['Contract', 'Contract_encoded']].drop_duplicates().sort_values('
    Contract_encoded'))
print("\nDistribuição:")
print(df['Contract_encoded'].value_counts().sort_index())
```

# Label Encoding: Implementação (cont.)

## Método 1: Mapeamento manual (recomendado para ordinais)

**</> Python**

```
# === LABEL ENCODING: Contract ===
#
# Valores originais e encodados:
#          Contract  Contract_encoded
# 0   Month-to-month                0
# 1         One year                1
# 11        Two year                2
#
# Distribuição:
# Contract_encoded
# 0    3875
# 1    1473
# 2    1695
```

# Label Encoding - `.cat.codes`

Encoding Stateless Nativo do Pandas:

- **Stateless:** Não mantém objeto com mapeamento persistente
- **Operação ad-hoc:** Executa encoding no DataFrame atual
- **Ordem automática:** Por padrão usa ordem alfabética/lexicográfica
- **Type-safe:** Cria um dtype Categorical eficiente

**Nota Importante**

**Ponto-chave:** .cat.codes é uma *propriedade* de objetos Categorical do pandas, não um transformer reutilizável.

# Label Encoding - `.cat.codes`: Comportamento

Vantagens:

- Performance otimizada
- Memória eficiente
- Integração natural com pandas
- Suporta ordem customizada
- Dtype explícito (ordered)

Desvantagens:

- Não persiste mapeamento
- Inconsistência sem cuidado
- Requer categories explícitas
- Não integra com sklearn
- Difícil de serializar

**⚠ Atenção**

**Armadilha comum:** Usar astype('category') sem definir categories pode gerar encodings diferentes em train vs test!

# Label Encoding - `.cat.codes`: Implementação

**Método 2: Mapeamento automático (usando cat.codes)**

**Python**

```python
# Definir ordem explícita
df['Contract'] = pd.Categorical(
    df['Contract'],
    categories=['Month-to-month', 'One year', 'Two year'],
    ordered=True
)
df['Contract_encoded'] = df['Contract'].cat.codes
# 0, 1, 2 na ordem definida
```

# Label Encoding - Sklearn LabelEncoder

## Encoding Stateful e Reutilizável:

- **Stateful:** Mantém objeto com mapeamento interno (classes_)
- **Fit/Transform pattern:** Aprende no treino, aplica no teste
- **Ordem alfabética:** Sempre ordena classes alfabeticamente
- **Serializável:** Pode ser salvo e carregado (joblib)

**Nota Importante**

**Ponto-chave:** LabelEncoder é um *transformer sklearn* que armazena estado e pode ser reutilizado em novos dados.

# Label Encoding - Sklearn LabelEncoder: Comportamento

Vantagens:

- Consistência garantida
- Fit/transform separation
- Erro explícito em unknowns
- Fácil serialização
- Inverse transform nativo
- Ideal para target encoding

Limitações:

- Apenas 1D arrays
- Ordem alfabética fixa
- Não integra diretamente em ColumnTransformer
- Menos controle sobre ordem
- Sem ordered categorical

**Nota Importante**

Para features em pipelines, usar OrdinalEncoder (2D). LabelEncoder é ideal para target (y).

# Label Encoding - Sklearn LabelEncoder: Implementação

## Método 3: LabelEncoder do sklearn

**</> Python**

```python
from sklearn.preprocessing import LabelEncoder

# Criar encoder
le = LabelEncoder()

# Fit e transform
df['Contract_encoded_sklearn'] = le.fit_transform(df['Contract'])

# Ver mapeamento criado
print("=== SKLEARN LABEL ENCODER ===")
print("Classes encontradas:", le.classes_)
print("Mapeamento:")
for i, label in enumerate(le.classes_):
    print(f"  {label} -> {i}")
```

# Label Encoding - Sklearn LabelEncoder (cont.)

## Método 3: LabelEncoder do sklearn

**</> Python**

```python
# === SKLEARN LABEL ENCODER ===
# Classes encontradas: ['Month-to-month' 'One year' 'Two year']
# Mapeamento:
#   Month-to-month -> 0
#   One year -> 1
#   Two year -> 2

# ATENÇÃO: sklearn mapeia alfabeticamente!
# Pode não preservar ordem desejada para ordinais
# Por isso, mapeamento manual é preferível para ordinais
```

# Comparação: Estado e Persistência

cat.codes (Stateless)

- Não mantém objeto
- Mapeamento implícito
- Requer categories externas
- Serialização manual
- Recriação necessária

**Implicação:** Desenvolvedor é responsável por manter consistência entre treino e produção.

LabelEncoder (Stateful)

- Mantém objeto encoder
- Mapeamento em classes_
- Autocontido
- joblib.dump() direto
- Reutilização simples

**Implicação:** Encoder garante consistência automaticamente via transform().

# Comparação: Controle de Ordem

## cat.codes: Flexibilidade

**Opções de ordenação:**

- Alfabética
- Por frequência
- Semântica customizada
- Qualquer ordem desejada

**Exemplo de uso:**

- Ordenar por importância
- Ordenar por risco
- Ordem de negócio

## LabelEncoder: Consistência

**Ordem fixa:**

- Sempre alfabética
- Determinística
- Não customizável
- Previsível

**Trade-off:**

- Menos flexibilidade
- Mais consistência
- Menos erros humanos

**Decisão:** Se ordem customizada é crítica → cat.codes. Se consistência é crítica → LabelEncoder/OrdinalEncoder.

# Comparação: Performance e Memória

Aspectos de eficiência computacional:

**cat.codes**

- Dtype Categorical eficiente
- Economia de memória
- Operações otimizadas
- Cache-friendly
- Ideal para datasets grandes

Vantagem: 50-90% menos memória com muitas categorias repetidas

**LabelEncoder**

- Retorna array numpy padrão
- Memória convencional
- Performance adequada
- Overhead mínimo
- Compatibilidade total

Vantagem: Simplicidade e integração universal

**Nota Importante**

Na prática, diferença de performance é marginal.

# Tabela Comparativa Resumida

| Aspecto | cat.codes | LabelEncoder |
|---|---|---|
| Estado | Stateless | Stateful |
| Consistência train/test | Manual | Automática |
| Valores desconhecidos | -1 (silencioso) | ValueError |
| Ordem | Customizável | Alfabética fixa |
| Serialização | Manual | joblib.dump() |
| Pipeline sklearn | Não | Sim (1D) |
| Inverse transform | Manual | Nativo |
| Arrays 2D | Não | Não (use OrdinalEncoder) |
| Memória | Eficiente | Padrão |
| Dtype | Categorical | numpy array |

**Nota Importante**

Verde indica vantagem clara. Nenhuma ferramenta é superior em todos aspectos.

# Validar Label Encoding

**Verificar se encoding faz sentido:**

**</> Python**

```python
# Analisar relação entre encoded e Churn
print("=== VALIDACAO: Contract_encoded vs Churn ===")

# Média de churn por nível de contract
churn_by_contract = df.groupby('Contract_encoded')['Churn'].apply(
    lambda x: (x == 'Yes').mean()
)
print("\nTaxa de Churn por Contract (encoded):")
print(churn_by_contract)

# Correlação com Churn_binary
from scipy.stats import pearsonr
corr, pval = pearsonr(df['Contract_encoded'], df['Churn_binary'])
print(f"\nCorrelação com Churn: {corr:.3f} (p={pval:.4f})")
```

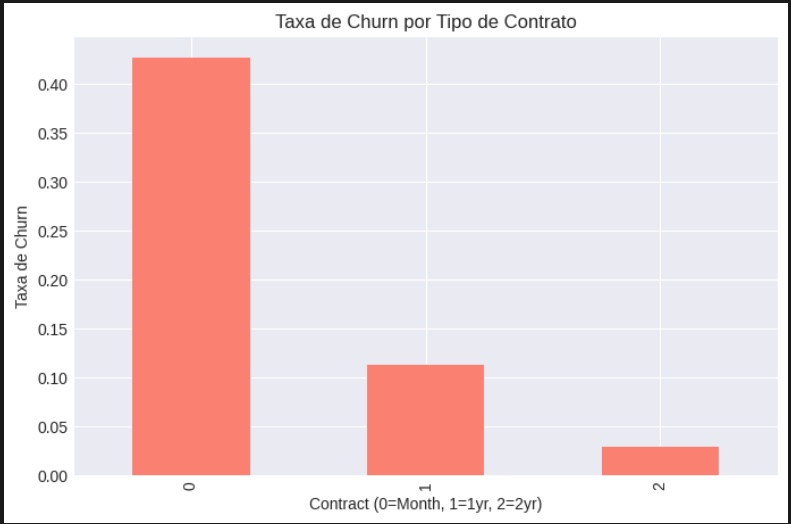
# Validar Label Encoding (cont.)

**Verificar se encoding faz sentido:**

**</> Python**

```python
# Visualizar
import matplotlib.pyplot as plt
churn_by_contract.plot(kind='bar', color='salmon', figsize=(8, 5))
plt.title('Taxa de Churn por Tipo de Contrato')
plt.xlabel('Contract (0=Month, 1=1yr, 2=2yr)')
plt.ylabel('Taxa de Churn')
plt.show()
```

# Validar Label Encoding (cont.)



# One-Hot Encoding: Para Nominais

**Quando usar:**

- Variável NÃO tem ordem natural
- Categorias são simplesmente ”diferentes”
- Exemplos: cor, cidade, método de pagamento

**Como funciona:**

- Criar uma coluna binária para CADA categoria
- 1 se categoria está presente, 0 caso contrário
- Também chamado de ”dummy variables”

# One-Hot Encoding: Para Nominais

**Exemplo - InternetService (3 categorias):**

| Original | InternetService_DSL | InternetService_Fiber | InternetService_No |
|---|---|---|---|
| DSL | 1 | 0 | 0 |
| Fiber optic | 0 | 1 | 0 |
| No | 0 | 0 | 1 |

**Resultado:** 3 colunas binárias ao invés de 1 categórica

# One-Hot: Por que NÃO usar Label Encoding?

**Exemplo: PaymentMethod**

- Categorias: Electronic check, Mailed check, Bank transfer, Credit card
- Se usarmos Label: 0, 1, 2, 3

**O que está errado?**

- Números implicam ordem: 3 > 2 > 1 > 0
- Mas não há ordem real entre métodos de pagamento!
- Credit card (3) não é ”maior”que Electronic check (0)
- Análises podem interpretar erroneamente como ordinal
- Correlações ficam sem sentido

**Solução: One-Hot**

- Cada método vira coluna binária independente
- Sem implicação de ordem
- Cada coluna pode ser correlacionada separadamente

# One-Hot Encoding: Implementação

**Método: pd.get_dummies()**

**Python**

```python
# One-Hot Encoding de InternetService
print("=== ONE-HOT ENCODING: InternetService ===")

# Antes
print("Coluna original:")
print(df['InternetService'].value_counts())

# Aplicar One-Hot
internet_dummies = pd.get_dummies(df['InternetService'],
                                  prefix='InternetService',
                                  dtype=int)
```

# One-Hot Encoding: Implementação

**Método: pd.get_dummies()**

**</> Python**

```python
# Visualizar
print("\nColunas criadas:")
print(internet_dummies.columns.tolist())

print("\nExemplo de dados:")
print(internet_dummies.head())

# Adicionar ao dataframe
df = pd.concat([df, internet_dummies], axis=1)

print(f"\nShape antes: {df.shape[0]} x {df.shape[1] - len(internet_dummies.columns)}")
print(f"Shape depois: {df.shape}")
```

# One-Hot: Múltiplas Colunas de Uma Vez

**</> Python**

```python
# Aplicar One-Hot a múltiplas colunas nominais
nominal_cols = ['InternetService', 'PaymentMethod']

# Método 1: Uma por vez (mais controle)
for col in nominal_cols:
    dummies = pd.get_dummies(df[col], prefix=col, dtype=int)
    df = pd.concat([df, dummies], axis=1)

# Método 2: Todas de uma vez (mais simples)
df_encoded = pd.get_dummies(df,
                            columns=nominal_cols,
                            prefix=nominal_cols,
                            dtype=int,
                            drop_first=False)  # Veremos a seguir
```

# One-Hot: Múltiplas Colunas de Uma Vez (cont.)

**</> Python**

```python
print("=== COLUNAS CRIADAS ===")
new_cols = [col for col in df_encoded.columns if any(nom in col for nom in
    nominal_cols)]
print(f"Total: {len(new_cols)} novas colunas")
print(new_cols)

# === COLUNAS CRIADAS ===
# Total: 17 novas colunas
# ['InternetService_DSL', 'InternetService_Fiber optic',
# 'InternetService_No', 'InternetService_DSL',
# 'InternetService_Fiber optic', 'InternetService_No', ...
```

# One-Hot com drop_first

**Problema:**

- Se temos InternetService com 3 categorias
- One-Hot cria 3 colunas: DSL, Fiber, No
- Mas: se DSL=0 e Fiber=0, então necessariamente No=1
- Informação redundante! (colinearidade perfeita)

**Solução: drop_first=True**

- Remove primeira categoria
- Serve como referência (baseline)
- N categorias $\rightarrow$ N-1 colunas

**Exemplo:**

- InternetService: 3 categorias $\rightarrow$ 2 colunas
- Se Fiber=0 e No=0, então é DSL (implícito)
- DSL é a categoria de referência

# One-Hot com drop_first: Implementação

**Python**

```python
# Comparar com e sem drop_first
print("=== COMPARACAO: drop_first ===\n")

# Sem drop_first (3 colunas)
dummies_full = pd.get_dummies(df['InternetService'],
                              prefix='Internet',
                              dtype=int,
                              drop_first=False)
print(f"Sem drop_first: {dummies_full.shape[1]} colunas")
print(dummies_full.columns.tolist())

# ['Internet_DSL', 'Internet_Fiber optic', 'Internet_No']
```

**Quando usar drop_first?**

# One-Hot com drop_first: Implementação (cont.)

**Python**

```python
# Com drop_first (2 colunas)
dummies_dropped = pd.get_dummies(df['InternetService'],
                                 prefix='Internet',
                                 dtype=int,
                                 drop_first=True)
print(f"\nCom drop_first: {dummies_dropped.shape[1]} colunas")
print(dummies_dropped.columns.tolist())
# ['Internet_Fiber optic', 'Internet_No']
print("\nCategoria de referência (implícita): DSL")
```

# One-Hot com drop_first: Implementação (cont.)

**Quando usar drop_first?**

- Para análise exploratória: tanto faz
- Para ML (futuro): geralmente sim (evita multicolinearidade)

# One-Hot: Vantagens e Desvantagens

**Vantagens:**

- ✓ Não impõe ordem artificial
- ✓ Cada categoria é independente
- ✓ Interpretação clara (1 = tem, 0 = não tem)
- ✓ Funciona para qualquer tipo de categoria
- ✓ Permite correlações separadas

**Desvantagens:**

- ✗ Aumenta dimensionalidade (curse of dimensionality)
- ✗ N categorias $\rightarrow$ N (ou N-1) colunas
- ✗ Problemático para alta cardinalidade (muitas categorias)
- ✗ Dataset fica ”esparso” (muitos zeros)

# One-Hot: Vantagens e Desvantagens

**⚠ Atenção**

Para variável com 100 categorias: One-Hot cria 100 colunas!
Veremos alternativas para alta cardinalidade.

# Label vs One-Hot: Quando usar cada um

| Aspecto | Label Encoding | One-Hot Encoding |
|---|---|---|
| Tipo de variável | Ordinal | Nominal |
| Ordem importa? | Sim | Não |
| Colunas criadas | 1 | N (ou N-1) |
| Espaço usado | Eficiente | Pode explodir |
| Interpretação | Ordem numérica | Presença/ausência |
| Exemplo | Contract, Educação, Tamanho | PaymentMethod, Cidade, Cor |

**Nota Importante**

Regra de ouro: Se tem ordem natural clara $\rightarrow$ Label
Se não tem ordem $\rightarrow$ One-Hot

# Fluxograma: Escolher Encoding

**Árvore de decisão:**

**Variável tem ordem natural?**
↓ SIM ↓ NÃO
**Label Encoding** **Quantas categorias?**
↓ 2 ↓ < 20 ↓ > 20
**0/1** **One-Hot** **Alternativas**

**Alternativas para alta cardinalidade (20):**

- Frequency Encoding
- Target Encoding (cuidado com leakage!)
- Agrupar categorias raras
- Embeddings (deep learning - futuro)

# Caso Prático: Pipeline de Encoding Telco

**Aplicar encoding sistemático ao dataset:**

**Python**

```python
# Criar cópia para trabalhar
df_encoded = df.copy()

print("=== PIPELINE DE ENCODING ===\n")
# 1. ORDINAIS: Label Encoding
print("1. Encoding ORDINAIS (Label)...")
ordinal_mappings = {
    'Contract': {
        'Month-to-month': 0,
        'One year': 1,
        'Two year': 2
    }
}
```

# Caso Prático: Pipeline de Encoding Telco

**Aplicar encoding sistemático ao dataset:**

**</> Python**

```python
for col, mapping in ordinal_mappings.items():
    df_encoded[f'{col}_encoded'] = df_encoded[col].map(mapping)
    print(f"     {col} encoded")
```

# Caso Prático: Pipeline de Encoding (cont.)

**Python**

```python
# 2. BINÁRIAS: Conversão 0/1
print("\n2. Encoding BINÁRIAS (0/1)...")
binary_cols = ['gender', 'Partner', 'Dependents',
               'PhoneService', 'PaperlessBilling']

for col in binary_cols:
    # Assumindo que 'Yes'/'Male' = 1, 'No'/'Female' = 0
    df_encoded[f'{col}_encoded'] = df_encoded[col].map({
        'Yes': 1, 'No': 0,
        'Male': 1, 'Female': 0
    }).fillna(df_encoded[col].map({
        1: 1, 0: 0  # Se já for numérico (SeniorCitizen)
    }))
    print(f"    {col} encoded")
```

# Caso Prático: Pipeline de Encoding (cont.)

**</> Python**

```python
# 3. NOMINAIS: One-Hot
print("\n3. Encoding NOMINAIS (One-Hot)...")
nominal_cols = ['PaymentMethod', 'InternetService']

df_encoded = pd.get_dummies(df_encoded,
                            columns=nominal_cols,
                            prefix=nominal_cols,
                            drop_first=True,
                            dtype=int)
print(f"    {len(nominal_cols)} colunas expandidas")
```

# Variáveis Binárias: Caso Especial

**Tipos de binárias:**

- Yes/No
- True/False
- Male/Female
- 1/0

**Como encodar:**

- Opção 1: 0 e 1 (mais comum e recomendado)
- Opção 2: -1 e 1 (às vezes usado)
- Opção 3: One-Hot (desnecessário e redundante!)

**Recomendação:**

- Use 0 para ”negativo/ausente/feminino”
- Use 1 para ”positivo/presente/masculino”
- Não precisa One-Hot (seria criar 2 colunas redundantes)
- Mais eficiente: 1 coluna ao invés de 2

# Variáveis Binárias: Caso Especial

**Nota Importante**

Binárias: encoding mais simples possível (0/1 direto)

# Encoding de Binárias: Implementação

**Python**

```python
# Método 1: map() - mais explícito
df['gender_encoded'] = df['gender'].map({'Male': 1, 'Female': 0})

# Método 2: operador booleano - mais conciso
df['Partner_encoded'] = (df['Partner'] == 'Yes').astype(int)

# Método 3: replace() - similar ao map
df['Dependents_encoded'] = df['Dependents'].replace({'Yes': 1, 'No': 0})
```

# Encoding de Binárias: Implementação

**</> Python**

```python
# Verificar
print("=== ENCODING DE BINÁRIAS ===")
for col in ['gender', 'Partner', 'Dependents']:
    encoded_col = f'{col}_encoded'
    print(f"\n{col}:")
    print(df[[col, encoded_col]].drop_duplicates())

# Validar: não deve haver valores além de 0 e 1
print("\nValores únicos (devem ser apenas 0 e 1):")
print(df[[c for c in df.columns if '_encoded' in c]].nunique())
```

# Alta Cardinalidade: O Problema

**O que é alta cardinalidade?**

- Variável categórica com MUITAS categorias únicas
- Exemplos: CEP (milhares), UserID, ProductID

**Por que é problema para One-Hot?**

- Explosão dimensional
- 1000 categorias $\rightarrow$ 1000 colunas!
- Dataset fica enorme e esparso
- 99% dos valores são zeros
- Dificulta análise e visualização
- Consome muita memória

# Alta Cardinalidade: O Problema

**Threshold comum:**

- $<$ 10 categorias: One-Hot sem problemas
- 10-20 categorias: One-Hot ainda OK
- $>$ 20 categorias: Considerar alternativas
- $>$ 50 categorias: Definitivamente usar alternativas

# Soluções para Alta Cardinalidade

**Alternativas ao One-Hot:**

**1. Agrupar categorias raras**
- Categorias com frequência < 5% $\rightarrow$ ”Other”
- Reduz número de categorias
- Mantém informação das principais

**2. Frequency Encoding**
- Substituir categoria por sua frequência
- Ex: ”SP”aparece 30% das vezes $\rightarrow$ 0.30
- Uma coluna numérica ao invés de N binárias

**3. Target Encoding (avançado)**
- Substituir categoria pela média do target naquela categoria
- Ex: Churn médio em ”SP” = 0.25 $\rightarrow$ 0.25
- **Cuidado:** risco de data leakage!
- Mais usado em ML

# Solução 1: Agrupar Categorias Raras

**Python**

```python
# Exemplo: Agrupar métodos de pagamento raros
def group_rare_categories(series, threshold=0.05, other_label='Other'):
    """
    Agrupa categorias com frequência < threshold em 'Other'

    Parameters:
    -----------
    series : pd.Series
        Coluna categórica
    threshold : float
        Frequência mínima (default: 5%)
    other_label : str
        Label para categorias raras
    """
```

# Solução 1: Agrupar Categorias Raras (cont.)

**Python**

```python
    # Calcular frequências
    freq = series.value_counts(normalize=True)

    # Identificar categorias raras
    rare_categories = freq[freq < threshold].index.tolist()

    # Substituir raras por 'Other'
    return series.replace(rare_categories, other_label)
```

# Solução 1: Agrupar Categorias Raras (cont.)

**Python**

```python
# Aplicar agrupamento
print("=== AGRUPAMENTO DE CATEGORIAS RARAS ===")

# Antes
print("\nANTES - PaymentMethod:")
print(df['PaymentMethod'].value_counts(normalize=True))

# Agrupar
df['PaymentMethod_grouped'] = group_rare_categories(
    df['PaymentMethod'],
    threshold=0.10  # Agrupar se < 10%
)
```

# Solução 1: Agrupar Categorias Raras (cont.)

**Python**

```python
# Depois
print("\nDEPOIS - PaymentMethod_grouped:")
print(df['PaymentMethod_grouped'].value_counts(normalize=True))

# Agora podemos aplicar One-Hot com menos colunas
dummies = pd.get_dummies(df['PaymentMethod_grouped'],
                         prefix='Payment',
                         drop_first=True)
print(f"\nColunas criadas: {len(dummies.columns)}")
```

# Solução 2: Frequency Encoding

**</> Python**

```python
# Frequency Encoding: substituir por frequência
def frequency_encoding(series):
    """
    Codifica categorias por sua frequência relativa

    Returns:
    --------
    pd.Series com frequências
    """
    freq = series.value_counts(normalize=True)
    return series.map(freq)
```

**Vantagem:** 1 coluna numérica ao invés de N binárias

# Solução 2: Frequency Encoding

**Python**

```python
# Aplicar
print("=== FREQUENCY ENCODING ===")

df['PaymentMethod_freq'] = frequency_encoding(df['PaymentMethod'])

print("\nMapeamento criado:")
print(df[['PaymentMethod', 'PaymentMethod_freq']].drop_duplicates().
    sort_values('PaymentMethod_freq', ascending=False))

print("\nEstatísticas da feature encodada:")
print(df['PaymentMethod_freq'].describe())
```

**Vantagem:** 1 coluna numérica ao invés de N binárias

# Comparar Estratégias de Encoding

**Análise comparativa:**

**Python**

```python
# Comparar diferentes encodings de PaymentMethod
print("=== COMPARACAO DE ENCODINGS ===\n")

# 1. One-Hot (baseline)
onehot = pd.get_dummies(df['PaymentMethod'], prefix='PM')
print(f"One-Hot: {onehot.shape[1]} colunas")

# 2. One-Hot com agrupamento
grouped_onehot = pd.get_dummies(df['PaymentMethod_grouped'], prefix='PM')
print(f"One-Hot + Grouped: {grouped_onehot.shape[1]} colunas")

# 3. Frequency Encoding
print(f"Frequency: 1 coluna")
```

# Comparar Estratégias de Encoding (cont.)

**Análise comparativa:**

**Python**

```python
# 4. Label (se forçarmos - não recomendado para nominal!)
label_enc = pd.factorize(df['PaymentMethod'])[0]
print(f"Label (NÃO RECOMENDADO para nominal): 1 coluna")

print("\n=== RECOMENDACAO ===")
print("Para PaymentMethod (nominal, 4 categorias):")
print(" - One-Hot é apropriado (apenas 4 colunas)")
```

# Comparar Estratégias de Encoding (cont.)

**Análise comparativa:**

**Python**

```python
# === COMPARACAO DE ENCODINGS ===
#
# One-Hot: 4 colunas
# One-Hot + Grouped: 4 colunas
# Frequency: 1 coluna
# Label (NÃO RECOMENDADO para nominal): 1 coluna
#
# === RECOMENDACAO ===
# Para PaymentMethod (nominal, 4 categorias):
# - One-Hot é apropriado (apenas 4 colunas)

```

# Cuidados com Encoding

**Armadilhas comuns:**

**1. Usar Label para nominais**

- ✗ Impõe ordem artificial
- ✗ Correlações sem sentido
- Exemplo: PaymentMethod como 0,1,2,3 sugere ordem que não existe

**2. Esquecer de encodar todas as categóricas**

- Deixar alguma coluna object atrapalha análises
- Verificar: `df.select_dtypes('object')`

# Cuidados com Encoding

**Armadilhas comuns:**

**3. Perder track das colunas originais**

- Documentar qual encoding foi usado onde
- Manter mapeamento para interpretação posterior
- Comentar código claramente

**4. Não validar resultado**

- Sempre verificar distribuições
- Checar valores únicos
- Garantir que faz sentido para o negócio

# Verificar Resultado do Encoding

**Checklist de validação:**

**</> Python**

```python
# Após encoding completo
print("=== VALIDACAO DO ENCODING ===\n")

# 1. Verificar se ainda há colunas object
object_cols = df_encoded.select_dtypes(include='object').columns.tolist()
# Remover colunas que devem ser object (IDs, etc)
object_cols = [c for c in object_cols if c not in ['customerID', 'Churn']]

print(f"1. Colunas object restantes: {len(object_cols)}")
if len(object_cols) > 0:
    print(f"   ATENÇÃO: {object_cols}")
else:
    print("   Todas as features são numéricas")
# 1. Colunas object restantes: 15 ...
```

# Verificar Resultado do Encoding (cont.)

**Checklist de validação:**

**Python**

```python
# 2. Verificar shape
print(f"\n2. Shape:")
print(f"   Original: {df.shape}")
print(f"   Encoded: {df_encoded.shape}")
print(f"   Colunas adicionadas: {df_encoded.shape[1] - df.shape[1]}")
```

# Verificar Resultado do Encoding (cont.)

**</> Python**

```python
# 3. Verificar valores únicos das encoded
print("\n3. Valores únicos por coluna encoded:")
encoded_cols = [c for c in df_encoded.columns if '_encoded' in c or
                any(nom in c for nom in ['PaymentMethod_', '
    InternetService_'])]

for col in encoded_cols[:10]:  # Primeiras 10
    n_unique = df_encoded[col].nunique() #ATENÇÃO
    unique_vals = sorted(df_encoded[col].unique())
    print(f"   {col}: {n_unique} valores  $\rightarrow$  {unique_vals}")
```

# Verificar Resultado do Encoding (cont.)

**Python**

```python
# 4. Correlação entre features encodadas e target
print("\n4. Correlação com Churn:")
correlations = df_encoded[encoded_cols].corrwith(df_encoded['Churn_binary'])
top_corr = correlations.abs().sort_values(ascending=False).head(10)
print(top_corr)

print("\n Validação completa!")
```

# Função Reutilizável: Pipeline de Encoding

**</> Python**

```python
def encode_telco_churn(df):
    """
    Pipeline completo de encoding para dataset Telco Churn

    Parameters:
    -----------
    df : DataFrame
        Dataset original

    Returns:
    --------
    DataFrame encodado
    """
    df_enc = df.copy()
```

# Função Reutilizável: Pipeline (cont.)

**Python**

```python
    # 1. Ordinais (Label)
    ordinal_maps = {
        'Contract': {'Month-to-month': 0, 'One year': 1, 'Two year': 2}
    }
    for col, mapping in ordinal_maps.items():
        df_enc[f'{col}_encoded'] = df_enc[col].map(mapping)
```

# Função Reutilizável: Pipeline (cont.)

**Python**

```python
    # 2. Binárias (0/1)
    binary_cols = ['gender', 'Partner', 'Dependents',
                   'PhoneService', 'PaperlessBilling']
    for col in binary_cols:
        df_enc[f'{col}_encoded'] = (df_enc[col] == 'Yes').astype(int) \
            if df_enc[col].dtype == 'object' else df_enc[col]

    # SeniorCitizen já é 0/1
    df_enc['SeniorCitizen_encoded'] = df_enc['SeniorCitizen']
```

# Função Reutilizável: Pipeline (cont.)

**Python**

```python
    # 3. Nominais (One-Hot)
    nominal_cols = ['PaymentMethod', 'InternetService']
    df_enc = pd.get_dummies(df_enc, columns=nominal_cols,
                            prefix=nominal_cols, drop_first=True, dtype=
    int)

    # 4. Target binário
    df_enc['Churn_binary'] = (df_enc['Churn'] == 'Yes').astype(int)

    return df_enc
```

# Usar Pipeline de Encoding

**</> Python**

```python
# Aplicar pipeline completo
print("=== APLICANDO PIPELINE DE ENCODING ===\n")

# Encoding
df_final = encode_telco_churn(df)

print(f"Shape original: {df.shape}")
print(f"Shape final: {df_final.shape}")

# Listar features numéricas
numeric_features = df_final.select_dtypes(include=['int64', 'float64']).
    columns
print(f"\nTotal de features numéricas: {len(numeric_features)}")
```

# Usar Pipeline de Encoding (cont.)

**Python**

```python
# Verificar se há categorical restante (exceto IDs e target original)
cat_cols = df_final.select_dtypes('object').columns
cat_cols = [c for c in cat_cols if c not in ['customerID', 'Churn']]
print(f"Colunas categóricas restantes: {len(cat_cols)}")

if len(cat_cols) == 0:
    print("\n Encoding completo! Todos os dados são numéricos.")
else:
    print(f"\n Atenção: ainda há colunas categóricas: {cat_cols}")
```

# Análise Após Encoding

**Exploração com dados encodados:**

**Python**

```python
# Matriz de correlação
import seaborn as sns
import matplotlib.pyplot as plt

# Selecionar features relevantes
features_to_analyze = [
    'tenure', 'MonthlyCharges', 'TotalCharges',
    'Contract_encoded', 'TotalServices',
    'InternetService_Fiber optic', 'InternetService_No',
    'PaymentMethod_Electronic check',
    'Churn_binary'
]
```

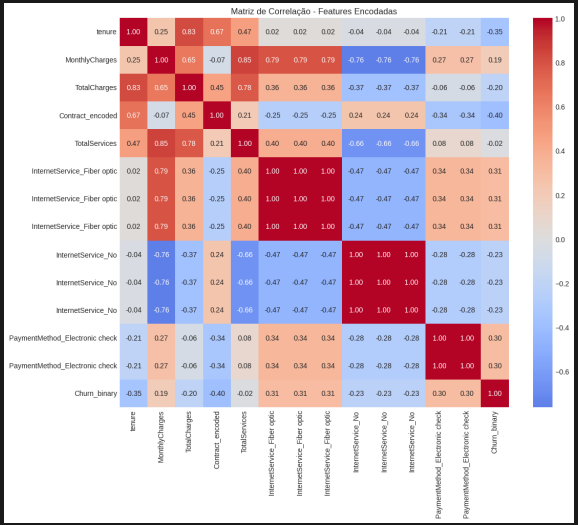
# Análise Após Encoding (cont.)

**Exploração com dados encodados:**

**Python**

```python
# Heatmap
fig, ax = plt.subplots(figsize=(12, 10))
corr_matrix = df_final[features_to_analyze].corr()
sns.heatmap(corr_matrix, annot=True, fmt='.2f', cmap='coolwarm',
            center=0, square=True, ax=ax)
ax.set_title('Matriz de Correlação - Features Encodadas')
plt.tight_layout()
plt.show()
```

# Análise Após Encoding (cont.)



# Resumo do Bloco 2

**O que aprendemos:**

**Conceitos:**

- Por que encoding é necessário
- Tipos de variáveis categóricas
- Escolher estratégia apropriada

**Técnicas:**

- Label Encoding (ordinais)
- One-Hot Encoding (nominais)
- Binary Encoding (binárias)
- Soluções para alta cardinalidade

# Resumo do Bloco 2

**O que aprendemos:**

**Habilidades:**

- Identificar tipo de variável
- Aplicar encoding correto
- Criar pipeline reproduzível
- Validar resultados
- Analisar dados encodados

# Bloco 3

Normalização e Padronização

# Introdução

**Problema:**

- Variáveis numéricas em escalas muito diferentes
- Dificulta comparações e visualizações
- Como tornar variáveis comparáveis?

**Técnicas:**

- Min-Max Scaling (normalização [0,1])
- Standardization (Z-score)
- Robust Scaling (resistente a outliers)
- Transformações matemáticas (log, sqrt)

**Objetivo:**

- Comparações justas entre variáveis
- Visualizações mais claras
- Identificar importância relativa
- Facilitar interpretação

# O Problema das Escalas Diferentes

**Exemplo concreto - Dataset Telco Churn:**

| Variável | Mínimo | Máximo |
|---|---|---|
| tenure (meses) | 0 | 72 |
| MonthlyCharges (R$) | 18 | 118 |
| TotalCharges (R$) | 18 | 8684 |

**Problemas:**

- TotalCharges varia em 8000+ unidades
- tenure varia apenas em 72 unidades
- Em gráficos: escala de TotalCharges domina visualmente
- Difícil comparar: ”tenure=36 é mais importante que TotalCharges=5000?”
- Escala diferente $\neq$ Importância diferente

# O Problema das Escalas Diferentes

**Exemplo concreto - Dataset Telco Churn:**

**Solução:**

- Colocar todas em escala comum
- Preservar relações dentro de cada variável
- Permitir comparações justas

# Benefícios da Normalização

**Para análise exploratória:**

**1. Visualizações mais claras**

- Múltiplas variáveis no mesmo gráfico
- Escalas compatíveis
- Comparações visuais diretas

**2. Comparabilidade**

- Identificar qual variável varia mais
- Comparar desvios padrão
- Entender distribuições relativamente

# Benefícios da Normalização

**Para análise exploratória:**

**3. Interpretação de correlações**

- Correlações mais significativas
- Coeficientes comparáveis
- Heatmaps mais informativos

**4. Preparação para análises avançadas**

- Clustering (agrupamento)
- PCA (análise de componentes principais)
- E futuramente: Machine Learning

# Três Técnicas Principais

**1. Min-Max Scaling (Normalização)**

- Escala para [0, 1]
- $X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}}$
- **Use quando:** Quer todos os valores entre 0 e 1
- **Sensível a:** Outliers

# Três Técnicas Principais

**2. Standardization (Z-score, Padronização)**

- Média = 0, Desvio Padrão = 1
- $X_{scaled} = \frac{X - \mu}{\sigma}$
- **Use quando:** Quer centralizar em zero
- **Sensível a:** Outliers moderados

# Três Técnicas Principais

**3. Robust Scaling**

- Usa mediana e IQR
- $X_{scaled} = \frac{X - median}{IQR}$
- **Use quando:** Tem MUITOS outliers
- **Resistente a:** Valores extremos

# Min-Max Scaling: Normalização para [0, 1]

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

**Exemplo manual:**

- Dados: [10, 20, 30, 40, 50]
- Min = 10, Max = 50, Range = 40
- 10 $\rightarrow$ (10-10)/40 = 0.0
- 30 $\rightarrow$ (30-10)/40 = 0.5
- 50 $\rightarrow$ (50-10)/40 = 1.0
- Resultado: [0.0, 0.25, 0.5, 0.75, 1.0]

**Propriedades:**

- Menor valor $\rightarrow$ 0
- Maior valor $\rightarrow$ 1
- Preserva forma da distribuição
- Interpretação: ”posição relativa no range”

# Min-Max: Implementação com Sklearn

**</> Python**

```python
from sklearn.preprocessing import MinMaxScaler
import numpy as np

# Selecionar colunas numéricas para normalizar
numeric_cols = ['tenure', 'MonthlyCharges', 'TotalCharges']

# Criar scaler
scaler = MinMaxScaler()
scaled_array = scaler.fit_transform(df[numeric_cols])

# Gerar novas colunas para features normalizadas
normalized_cols = [col + '_minmax' for col in numeric_cols]
for idx, col in enumerate(normalized_cols):
    df[col] = scaled_array[:, idx]
```

# Min-Max: Implementação com Sklearn (cont.)

**Python**

```python
# Aplicar fit and transform
df[normalized_cols] = scaler.fit_transform(df[numeric_cols])

# Alternativa: atribuição multicoluna
# scaled_df = pd.DataFrame(scaler.fit_transform(df[numeric_cols]), columns=normalized_cols, index=df.index)
# df = pd.concat([df, scaled_df], axis=1)

# Verificar resultado
print("=== MIN-MAX SCALING ===")
print("\nANTES:")
print(df[numeric_cols].describe())

print("\nDEPOIS:")
print(df[[c + '_minmax' for c in numeric_cols]].describe())
```

# Min-Max: Verificar Transformação

**Python**

```python
# Comparar valores originais e normalizados
print("=== COMPARACAO: Valores Originais vs Normalizados ===\n")

for col in numeric_cols:
    orig_col = col
    norm_col = col + '_minmax'

    print(f"\n{col}:")
    print(f"  Original - Min: {df[orig_col].min():.2f}, Max: {df[orig_col].max():.2f}")
    print(f"  Normalized - Min: {df[norm_col].min():.2f}, Max: {df[norm_col].max():.2f}")
```

# Min-Max: Verificar Transformação (cont.)

**Python**

```python
    # Exemplos
    sample_idx = df.sample(3).index
    print(f"  Exemplos:")
    for idx in sample_idx:
        orig_val = df.loc[idx, orig_col]
        norm_val = df.loc[idx, norm_col]
        print(f"    {orig_val:.2f}  -  {norm_val:.3f}")
```

# Min-Max: Vantagens e Desvantagens

**Vantagens:**

- ✓ Range fixo e previsível [0, 1]
- ✓ Fácil de interpretar: 0 = mínimo, 1 = máximo
- ✓ Preserva exatamente a forma da distribuição
- ✓ Bom para visualizações
- ✓ Útil quando há limites naturais nos dados

**Desvantagens:**

- ✗ Muito sensível a outliers
- ✗ Um único valor extremo ”comprime”todos os outros
- ✗ Dados novos podem ficar fora de [0, 1]
- ✗ Exemplo: se novo cliente tem tenure=100 (maior que max=72), valor normalizado seria > 1

# Min-Max: Vantagens e Desvantagens

**Quando usar:**

- Dados sem outliers extremos
- Quando precisa de range específico
- Dados com limites naturais (ex: proporções, percentuais)

# Standardization (Z-score): Detalhes

**Padronização: média 0, desvio padrão 1**

$$X_{std} = \frac{X - \mu}{\sigma}$$

onde $\mu$ = média, $\sigma$ = desvio padrão

**Exemplo manual:**

- Dados: [10, 20, 30, 40, 50]
- Média = 30, Desvio Padrão = 15.8
- 10 $\rightarrow$ (10-30)/15.8 = -1.27
- 30 $\rightarrow$ (30-30)/15.8 = 0.0
- 50 $\rightarrow$ (50-30)/15.8 = 1.27
- Resultado: [-1.27, -0.63, 0.0, 0.63, 1.27]

# Standardization (Z-score): Detalhes

**Exemplo manual:**

- Dados: [10, 20, 30, 40, 50]
- Média = 30, Desvio Padrão = 15.8
- 10 $\rightarrow$ (10-30)/15.8 = -1.27
- 30 $\rightarrow$ (30-30)/15.8 = 0.0
- 50 $\rightarrow$ (50-30)/15.8 = 1.27
- Resultado: [-1.27, -0.63, 0.0, 0.63, 1.27]

**Interpretação Z-score:**

- 0 = valor na média
- +1 = um desvio padrão acima da média
- -2 = dois desvios padrão abaixo da média
- $|Z| > 3$ = outlier extremo (regra geral)

# Standardization: Implementação

**Python**

```python
from sklearn.preprocessing import StandardScaler

# Criar scaler
scaler = StandardScaler()
std_cols = [c + '_std' for c in numeric_cols]

# Aplicar fit_transform
scaled_array = scaler.fit_transform(df[numeric_cols])
for i, col in enumerate(std_cols):
    df[col] = scaled_array[:, i]
```

# Standardization: Implementação (cont.)

**Python**

```python
# Verificar resultado
print("=== STANDARDIZATION (Z-SCORE) ===")
print("\nANTES:")
print(df[numeric_cols].describe())

print("\nDEPOIS:")
print(df[std_cols].describe())

print("\n Média  0, Desvio Padrão  1")
```

# Standardization: Interpretação

**Python**

```python
# Interpretar Z-scores
print("=== INTERPRETACAO DE Z-SCORES ===\n")

# Exemplo: tenure
tenure_std = df['tenure_std']

print("Distribuição dos Z-scores (tenure):")
print(f"  Entre -1 e +1: {((tenure_std >= -1) & (tenure_std <= 1)).mean()*100:.1f}%")
print(f"  Entre -2 e +2: {((tenure_std >= -2) & (tenure_std <= 2)).mean()*100:.1f}%")
print(f"  Outliers (|Z| > 3): {(abs(tenure_std) > 3).sum()} clientes")
```

# Standardization: Interpretação (cont.)

**Python**

```python
# Identificar extremos
print("\n=== CLIENTES COM TENURE EXTREMO ===")
print("Muito baixo (Z < -2):")
print(df[tenure_std < -2][['tenure', 'tenure_std', 'Churn']].head())

print("\nMuito alto (Z > 2):")
print(df[tenure_std > 2][['tenure', 'tenure_std', 'Churn']].head())
```

# Standardization: Vantagens e Desvantagens

**Vantagens:**

- ✓ Não está limitado a range específico
- ✓ Interpretação estatística clara (desvios padrão)
- ✓ Funciona bem com dados que seguem distribuição normal
- ✓ Facilita identificação de outliers
- ✓ Dados novos não ”quebram”a escala
- ✓ Muito usado em análises estatísticas

**Desvantagens:**

- ✗ Ainda sensível a outliers (usa média e desvio padrão)
- ✗ Valores não estão em range fixo
- ✗ Menos intuitivo para não-estatísticos
- ✗ Requer distribuição aproximadamente normal para melhor efeito

# Standardization: Vantagens e Desvantagens

**Quando usar:**

- Dados aproximadamente normais
- Quando interpretação estatística importa
- Outliers moderados (não extremos)
- Análises que assumem normalidade

# Robust Scaling: Para Outliers

**Escala resistente a valores extremos**

$$X_{robust} = \frac{X - median}{IQR}$$

onde IQR = Q3 - Q1 (intervalo interquartil)

**Por que é robusto?**

- Usa mediana (não média)
- Usa IQR (não desvio padrão)
- Ambos são resistentes a outliers
- Outliers não ”inflam” a escala

# Robust Scaling: Para Outliers

**Exemplo:**

- Dados: [10, 20, 30, 40, 1000] (1000 é outlier)
- Mediana = 30, Q1 = 20, Q3 = 40, IQR = 20
- 10 $\rightarrow$ (10-30)/20 = -1.0
- 30 $\rightarrow$ (30-30)/20 = 0.0
- 1000 $\rightarrow$ (1000-30)/20 = 48.5
- Outlier não afeta escala dos valores normais!

# Robust Scaling: Implementação

**Python**

```python
from sklearn.preprocessing import RobustScaler

# Criar scaler
scaler = RobustScaler()

# Aplicar fit_transform
robust_cols = [col + '_robust' for col in numeric_cols]
scaled_df = pd.DataFrame(scaler.fit_transform(df[numeric_cols]), columns=
    robust_cols, index=df.index)
df = pd.concat([df, scaled_df], axis=1)
```

# Robust Scaling: Implementação (cont.)

**Python**

```python
# Verificar resultado
print("=== ROBUST SCALING ===")
print("\nANTES:")
print(df[numeric_cols].describe())

print("\nDEPOIS:")
robust_cols = [c + '_robust' for c in numeric_cols]
print(df[robust_cols].describe())

print("\ n Mediana  0, IQR-based scaling")
```

# Comparação: Min-Max vs Standard vs Robust

**Python**

```python
print("=== COMPARACAO DAS TECNICAS ===\n")
scaling_methods = {
    'Original': 'tenure',
    'Min-Max': 'tenure_minmax',
    'Standard': 'tenure_std',
    'Robust': 'tenure_robust'
}
for name, col in scaling_methods.items():
    if col in df.columns:
        print(f"\n{name}:")
        print(f"  Min: {df[col].min():.2f}")
        print(f"  Median: {df[col].median():.2f}")
        print(f"  Mean: {df[col].mean():.2f}")
        print(f"  Max: {df[col].max():.2f}")
        print(f"  Std: {df[col].std():.2f}")
```

# Visualizar Efeito do Scaling

**Python**

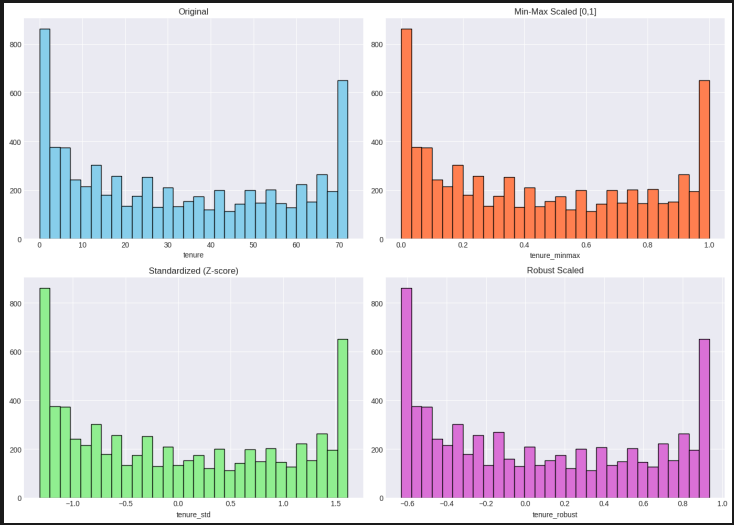
```python
import matplotlib.pyplot as plt
import seaborn as sns
fig, axes = plt.subplots(2, 2, figsize=(14, 10))
# Original
df['tenure'].hist(bins=30, ax=axes[0,0], color='skyblue', edgecolor='black')
axes[0,0].set_title('Original')
axes[0,0].set_xlabel('tenure')

# Min-Max
df['tenure_minmax'].hist(bins=30, ax=axes[0,1], color='coral', edgecolor='black')
axes[0,1].set_title('Min-Max Scaled [0,1]')
axes[0,1].set_xlabel('tenure_minmax')
```

# Visualizar Efeito do Scaling (cont.)

**Python**

```python
# Standardized
df['tenure_std'].hist(bins=30, ax=axes[1,0], color='lightgreen', edgecolor='black')
axes[1,0].set_title('Standardized (Z-score)')
axes[1,0].set_xlabel('tenure_std')

# Robust
df['tenure_robust'].hist(bins=30, ax=axes[1,1], color='orchid', edgecolor='black')
axes[1,1].set_title('Robust Scaled')
axes[1,1].set_xlabel('tenure_robust')

plt.tight_layout()
plt.show()
```

# Visualizar Efeito do Scaling (cont.)



# Quando NÃO Normalizar

**Situações onde scaling não é necessário ou até prejudicial:**

**1. Variáveis já estão na mesma escala**

- Ex: proporções (todas entre 0 e 1)
- Ex: percentuais (todas entre 0 e 100)
- Scaling seria redundante

**2. Escala original tem significado importante**

- Ex: temperatura em Celsius tem interpretação clara
- Ex: valor em reais é mais interpretável que normalizado
- Para apresentações de negócio: manter original

# Quando NÃO Normalizar

**Situações onde scaling não é necessário ou até prejudicial:**

**3. Análises que não dependem de escala**

- Árvores de decisão (splits são ordinais)
- Correlações (já são normalizadas [-1, 1])
- Análises descritivas simples

**4. Interpretabilidade é prioridade**

- Stakeholders não-técnicos
- Valores originais mais fáceis de explicar
- Relatórios de negócio

# Boas Práticas: Separar Dados para Validação

**Conceito importante:**

**Problema:**

- Quando normalizamos, calculamos parâmetros (min, max, média, etc)
- Esses parâmetros são calculados de TODO o dataset
- Se no futuro tivermos dados novos, como normalizar?
- Precisamos usar os MESMOS parâmetros!

**Solução: Fit vs Transform**

- `fit()`: Aprende parâmetros (min, max, média, etc)
- `transform()`: Aplica transformação usando parâmetros aprendidos
- `fit_transform()`: Faz ambos de uma vez

# Boas Práticas: Separar Dados para Validação

**Boa prática:**

- Separar dados em conjunto principal e validação
- Fit apenas no conjunto principal
- Transform em ambos usando os mesmos parâmetros
- Simula como seria com dados futuros/novos

# Implementar Separação de Dados

**</> Python**

```python
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
# 1. Separar dados (80% principal, 20% validação)
df_main, df_val = train_test_split(df, test_size=0.2, random_state=42)
print(f"Conjunto principal: {len(df_main)} registros")
print(f"Conjunto validação: {len(df_val)} registros")

# 2. Fit scaler APENAS no conjunto principal
scaler = StandardScaler()
scaler.fit(df_main[numeric_cols])
print("\n=== PARAMETROS APRENDIDOS (do conjunto principal) ===")
print(f"Médias: {scaler.mean_}")
print(f"Desvios: {scaler.scale_}")
```

# Aplicar Transformação aos Dois Conjuntos

**Python**

```python
# 3. Transform em AMBOS os conjuntos (usando mesmos parâmetros)
df_main_scaled = scaler.transform(df_main[numeric_cols])
df_val_scaled = scaler.transform(df_val[numeric_cols])
# Converter de volta para DataFrame
df_main[std_cols] = df_main_scaled
df_val[std_cols] = df_val_scaled

# 4. Verificar que validação usa mesmos parâmetros
print("=== VERIFICACAO ===")
print("\nConjunto Principal:")
print(f"  Média tenure_std: {df_main['tenure_std'].mean():.3f}  ( 0)")
print(f"  Std tenure_std: {df_main['tenure_std'].std():.3f}  ( 1)")
print("\nConjunto Validação:")
print(f"  Média tenure_std: {df_val['tenure_std'].mean():.3f} (pode   0)")
print(f"  Std tenure_std: {df_val['tenure_std'].std():.3f} (pode   1)")
print("\ n Validação usa parâmetros do principal (correto!)")
```

# Por que Separar é Importante?

**Simula cenário real:**

**Analogia:**

- Conjunto principal = dados históricos que temos hoje
- Conjunto validação = dados futuros que chegarão amanhã
- Quando dados novos chegarem, não podemos ”re-calcular”min/max/média
- Precisamos usar os parâmetros que já tínhamos

**Exemplo prático:**

- Hoje: normalizamos tenure com min=0, max=72
- Amanhã: chega cliente com tenure=80 (novo máximo!)
- Usamos: (80-0)/72 = 1.11 (>1 é OK!)
- NÃO usamos: re-calcular com novo max=80
- Por quê? Consistência e comparabilidade

# Por que Separar é Importante?

**Relevância para ML (futuro):**

- Em ML, isso é CRÍTICO (evita ”data leakage”)
- Mesma boa prática que usamos aqui
- Você já estará familiarizado quando chegar lá!

# Transformações Matemáticas

**Para corrigir distribuições assimétricas:**

**</> Python**

```python
import numpy as np
# 1. Log Transform (para assimetria à direita)
# Útil para dados exponenciais ou com cauda longa
df['TotalCharges_log'] = np.log1p(df['TotalCharges'])
# log1p = log(1+x), evita log(0)
# 2. Square Root (menos agressivo que log)
df['TotalCharges_sqrt'] = np.sqrt(df['TotalCharges'])
# 3. Inverso (para assimetria à esquerda)
# Cuidado: inverte ordem!
df['tenure_inverse'] = 1 / (df['tenure'] + 1)  # +1 evita divisão por 0
print("=== TRANSFORMACOES MATEMATICAS ===")
print("\nOriginal vs Transformada:")
print(df[['TotalCharges', 'TotalCharges_log', 'TotalCharges_sqrt']].
    describe())
```

# Quando Usar Transformações Matemáticas

**Python**

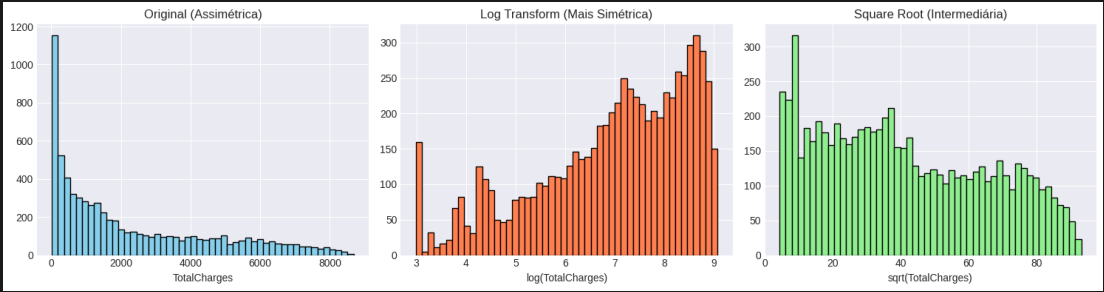
```python
# Visualizar efeito da transformação
import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, 3, figsize=(15, 4))

# Original (assimétrica)
df['TotalCharges'].hist(bins=50, ax=axes[0], color='skyblue', edgecolor='
    black')
axes[0].set_title('Original (Assimétrica)')
axes[0].set_xlabel('TotalCharges')
```

# Quando Usar Transformações Matemáticas (cont.)

**Python**

```python
# Log (mais simétrica)
df['TotalCharges_log'].hist(bins=50, ax=axes[1], color='coral', edgecolor='black')
axes[1].set_title('Log Transform (Mais Simétrica)')
axes[1].set_xlabel('log(TotalCharges)')

# Sqrt (intermediária)
df['TotalCharges_sqrt'].hist(bins=50, ax=axes[2], color='lightgreen', edgecolor='black')
axes[2].set_title('Square Root (Intermediária)')
axes[2].set_xlabel('sqrt(TotalCharges)')

plt.tight_layout()
plt.show()
```

# Quando Usar Transformações Matemáticas (cont.)



# Transformações: Quando Usar Cada Uma

**Guia de decisão:**

**Log Transform:**

- Distribuição muito assimétrica à direita
- Dados exponenciais (renda, população)
- Range muito grande (vários orders de magnitude)
- **Cuidado:** Requer valores > 0

**Square Root:**

- Assimetria moderada à direita
- Menos agressivo que log
- Dados de contagem (número de eventos)
- Aceita zero

# Transformações: Quando Usar Cada Uma

**Box-Cox (avançado):**

- Encontra transformação ótima automaticamente
- Família de transformações parametrizada
- Requer valores > 0
- Mais usado em análises estatísticas formais

# Pipeline Completo de Normalização

**Python**

```python
def normalize_features(df, numeric_cols, method='standard',
                       fit_data=None, return_scaler=False):
    """
    Parâmetros:
    -----------
    df : DataFrame
        Dados a normalizar
    numeric_cols : list
        Colunas numéricas
    method : str
        'minmax', 'standard', ou 'robust'
```

# Pipeline Completo (cont.)

**Python**

```
    fit_data : DataFrame, optional
        Se fornecido, fit nos fit_data e transform no df
    return_scaler : bool
        Se True, retorna (df_normalized, scaler)
    """
    from sklearn.preprocessing import MinMaxScaler, StandardScaler,
    RobustScaler

    # Escolher scaler
    scalers = {
        'minmax': MinMaxScaler(),
        'standard': StandardScaler(),
        'robust': RobustScaler()
    }
    scaler = scalers[method]
```

# Pipeline Completo (cont.)

**Python**

```python
    # Fit e transform
    if fit_data is not None:
        # Fit em fit_data, transform em df
        scaler.fit(fit_data[numeric_cols])
    else:
        # Fit e transform no mesmo df
        scaler.fit(df[numeric_cols])

    # Transform
    df_normalized = df.copy()
    df_normalized[numeric_cols] = scaler.transform(df[numeric_cols])

    if return_scaler:
        return df_normalized, scaler
    return df_normalized
```

# Pipeline Completo (cont.)

**Python**

```python
# Uso
df_normalized = normalize_features(df, numeric_cols, method='standard')
print("Normalização aplicada!")
```

# Análise Exploratória com Dados Normalizados

**Python**

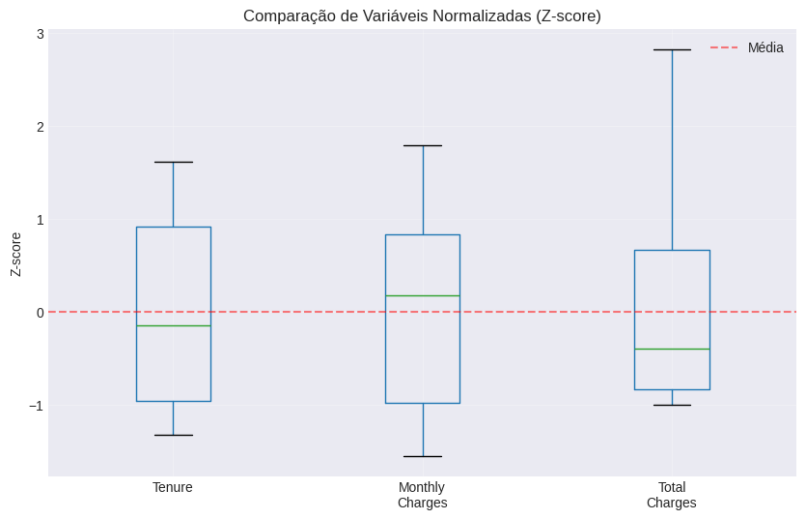
```python
# Comparar múltiplas variáveis no mesmo gráfico
import matplotlib.pyplot as plt

# Selecionar variáveis normalizadas
normalized_features = ['tenure_std', 'MonthlyCharges_std', '
    TotalCharges_std']
```

# Análise Exploratória com Dados Normalizados (cont.)

**</> Python**

```python
# Boxplot comparativo
fig, ax = plt.subplots(figsize=(10, 6))
df[normalized_features].boxplot(ax=ax)
ax.set_title('Comparação de Variáveis Normalizadas (Z-score)')
ax.set_ylabel('Z-score')
ax.set_xticklabels(['Tenure', 'Monthly\nCharges', 'Total\nCharges'])
ax.axhline(y=0, color='r', linestyle='--', alpha=0.5, label='Média')
ax.grid(True, alpha=0.3)
plt.legend()
plt.show()
print(" Agora podemos comparar variáveis diretamente!")
print("Todas na mesma escala (desvios padrão)")
```

# Análise Exploratória com Dados Normalizados (cont.)



# Heatmap com Variáveis Normalizadas

**Python**

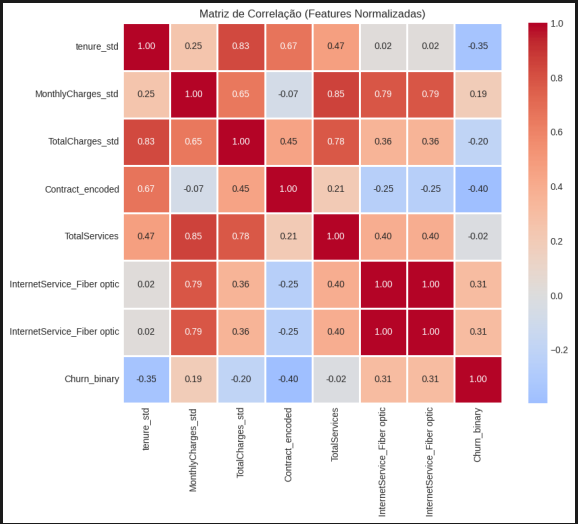
```python
# Matriz de correlação com features normalizadas
import seaborn as sns

features_for_heatmap = [
    'tenure_std', 'MonthlyCharges_std', 'TotalCharges_std',
    'Contract_encoded', 'TotalServices',
    'InternetService_Fiber optic',
    'Churn_binary'
]

# Calcular correlações
corr_matrix = df[features_for_heatmap].corr()
```

# Heatmap com Variáveis Normalizadas (cont.)

**Python**

```python
# Heatmap
fig, ax = plt.subplots(figsize=(10, 8))
sns.heatmap(corr_matrix, annot=True, fmt='.2f', cmap='coolwarm',
            center=0, square=True, linewidths=1, ax=ax)
ax.set_title('Matriz de Correlação (Features Normalizadas)')
plt.tight_layout()
plt.show()

print("Correlações mais interpretáveis com dados normalizados")
```

# Heatmap com Variáveis Normalizadas (cont.)



# Resumo do Bloco 3

**O que aprendemos:**

**Conceitos:**

- Por que normalizar (comparabilidade, visualização)
- Diferenças entre técnicas de scaling
- Quando usar cada técnica
- Quando NÃO normalizar

**Técnicas:**

- Min-Max Scaling (range fixo [0,1])
- Standardization (Z-score, média=0, std=1)
- Robust Scaling (resistente a outliers)
- Transformações matemáticas (log, sqrt)

# Resumo do Bloco 3

**Boas Práticas:**

- Separar dados para validação
- Fit vs Transform (parâmetros consistentes)
- Validar resultado visualmente
- Documentar escolhas

# Checklist de Normalização

**Antes de normalizar, pergunte:**

1. ☐ Variáveis estão em escalas muito diferentes?
   - Se não: talvez não precise normalizar
2. ☐ Qual objetivo da análise?
   - Comparação visual → Min-Max
   - Análise estatística → Standardization
   - Muitos outliers → Robust
3. ☐ Há outliers extremos?
   - Sim → Robust ou tratar outliers primeiro
   - Não → Min-Max ou Standard
4. ☐ Interpretabilidade é crítica?
   - Sim → Considerar manter valores originais
   - Para stakeholders: valores originais
   - Para análise técnica: pode normalizar
5. ☐ Tenho dados que simulariam dados futuros?
   - Sim → Separar e validar processo
   - Sempre boa prática!

# Bloco 4

Pipeline Completo

# O que vamos fazer

**Integração:**

- Juntar Feature Engineering + Encoding + Normalização
- Criar função pipeline end-to-end
- Aplicar ao dataset Telco Churn

**Análise Avançada:**

- Explorar dados completamente transformados
- Identificar insights que não eram visíveis antes
- Correlações, visualizações, comparações

**Fechamento:**

- Preview: conexão com Machine Learning
- Checklist de boas práticas
- Próximos passos no curso

# Pipeline Completo: Estrutura

**DADOS BRUTOS**
*Dataset original (bagunça)*
↓
**1. FEATURE ENGINEERING**
*Criar variáveis derivadas*
↓
**2. ENCODING**
*Categorias → números*
↓
**3. NORMALIZAÇÃO**
*Ajustar escalas*
↓
**DADOS PREPARADOS**
*Prontos para análise avançada*

# Pipeline: Função Completa

**Python**

```python
def transform_telco_churn(df, fit_scaler=True, scaler=None):
    """
    Pipeline completo de transformação para Telco Churn

    Parameters:
    -----------
    df : DataFrame
        Dataset bruto
    fit_scaler : bool
        Se True, fit novo scaler. Se False, usa scaler fornecido
    scaler : sklearn scaler, optional
        Scaler pré-treinado (para dados novos)
```

# Pipeline: Função Completa (cont.)

**Python**

```python
    Returns:
    --------
    df_transformed, scaler
    """
    from sklearn.preprocessing import StandardScaler

    df_transformed = df.copy()

    # ===== 1. FEATURE ENGINEERING =====
    print("1. Feature Engineering...")
```

# Pipeline: Função Completa (cont.)

**Python**

```python
    # Operações aritméticas
    df_transformed['AvgMonthlySpend'] = (
        df_transformed['TotalCharges'] / df_transformed['tenure']
    )
    df_transformed['AvgMonthlySpend'].replace([np.inf, -np.inf], np.nan,
    inplace=True)
    df_transformed.loc[df_transformed['tenure'] == 0, 'AvgMonthlySpend'] =
     \
        df_transformed.loc[df_transformed['tenure'] == 0, 'MonthlyCharges'
    ]
```

# Pipeline: Função Completa (cont.)

**Python**

```python
    # Agregações
    service_cols = ['PhoneService', 'MultipleLines', 'InternetService',
                    'OnlineSecurity', 'OnlineBackup', 'DeviceProtection',
                    'TechSupport', 'StreamingTV', 'StreamingMovies']
    df_transformed['TotalServices'] = 0
    for col in service_cols:
        df_transformed['TotalServices'] += (
            ~df_transformed[col].isin(['No', 'No internet service',
                                       'No phone service'])
        ).astype(int)
```

# Pipeline: Função Completa (cont.)

**Python**

```python
    # Flags
    df_transformed['IsPremium'] = (
        (df_transformed['TotalServices'] >= 6) &
        (df_transformed['Contract'].isin(['One year', 'Two year']))
    ).astype(int)
```

# Pipeline: Função Completa (cont.)

**Python**

```python
    # ===== 2. ENCODING =====
    print("2. Encoding...")

    # Ordinais (Label)
    contract_map = {'Month-to-month': 0, 'One year': 1, 'Two year': 2}
    df_transformed['Contract_encoded'] = df_transformed['Contract'].map(
    contract_map)

    # Binárias (0/1)
    binary_cols = ['gender', 'Partner', 'Dependents',
                   'PhoneService', 'PaperlessBilling']
    for col in binary_cols:
        df_transformed[f'{col}_encoded'] = (
            df_transformed[col] == 'Yes'
        ).astype(int)
```

# Pipeline: Função Completa (cont.)

**Python**

```python
    df_transformed['SeniorCitizen_encoded'] = df_transformed['
    SeniorCitizen']

    # Nominais (One-Hot)
    nominal_cols = ['PaymentMethod', 'InternetService']
    df_transformed = pd.get_dummies(df_transformed, columns=nominal_cols,
                                    prefix=nominal_cols, drop_first=True,
    dtype=int)
```

# Pipeline: Função Completa (cont.)

**Python**

```python
    # ===== 3. NORMALIZAÇÃO =====
    print("3. Normalização...")

    # Colunas numéricas para normalizar
    numeric_to_scale = ['tenure', 'MonthlyCharges', 'TotalCharges',
                        'AvgMonthlySpend', 'TotalServices']

    if fit_scaler:
        # Criar e treinar novo scaler
        scaler = StandardScaler()
        df_transformed[numeric_to_scale] = scaler.fit_transform(
            df_transformed[numeric_to_scale]
        )
```

# Pipeline: Função Completa (cont.)

**Python**

```python
    else:
        # Usar scaler existente
        if scaler is None:
            raise ValueError("Forneça scaler quando fit_scaler=False")
        df_transformed[numeric_to_scale] = scaler.transform(
            df_transformed[numeric_to_scale]
        )

    # Target binário
    df_transformed['Churn_binary'] = (df_transformed['Churn'] == 'Yes').
    astype(int)

    print("  Pipeline completo!")
    return df_transformed, scaler
```

# Aplicar Pipeline ao Dataset

**Python**

```python
# Aplicar pipeline completo
print("=== APLICANDO PIPELINE COMPLETO ===\n")

# Transformar
df_final, trained_scaler = transform_telco_churn(df, fit_scaler=True)

print(f"\nShape original: {df.shape}")
print(f"Shape final: {df_final.shape}")
print(f"Colunas adicionadas: {df_final.shape[1] - df.shape[1]}")
```

# Aplicar Pipeline ao Dataset (cont.)

**Python**

```python
# Verificar tipos
print("\n=== TIPOS DE DADOS ===")
print(f"Numéricas: {len(df_final.select_dtypes(include='number').columns)}
    ")
print(f"Categóricas: {len(df_final.select_dtypes(include='object').columns
    )}")

# Listar features finais
numeric_features = df_final.select_dtypes(include='number').columns
print(f"\nTotal de features numéricas: {len(numeric_features)}")
print("\ n Dataset completamente transformado!")
```

# Validar Pipeline

**Checklist de validação:**

**Python**

```python
# Validações pós-pipeline
print("=== VALIDACAO DO PIPELINE ===\n")

# 1. Verificar missing values
print("1. Missing values:")
missing = df_final.isnull().sum()
print(f"   Total: {missing.sum()}")
if missing.sum() > 0:
    print("   Colunas com missing:")
    print(missing[missing > 0])
```

# Validar Pipeline (cont.)

**Python**

```python
# 2. Verificar ranges das normalizadas
print("\n2. Ranges das variáveis normalizadas:")
for col in ['tenure', 'MonthlyCharges', 'TotalCharges']:
    print(f"   {col}: [{df_final[col].min():.2f}, {df_final[col].max():.2f}]")
    print(f"      Média: {df_final[col].mean():.2f}, Std: {df_final[col].std():.2f}")

# 3. Verificar encoding
print("\n3. Encoding:")
encoded_cols = [c for c in df_final.columns if '_encoded' in c or
                'PaymentMethod_' in c or 'InternetService_' in c]
print(f"   Colunas encodadas: {len(encoded_cols)}")
```

# Validar Pipeline (cont.)

**Python**

```python
# 4. Verificar features criadas
print("\n4. Features criadas:")
created = ['AvgMonthlySpend', 'TotalServices', 'IsPremium']
for feat in created:
    if feat in df_final.columns:
        print(f"    {feat}")
    else:
        print(f"    {feat} - FALTANDO!")
```

# Validar Pipeline (cont.)

**Python**

```python
# 5. Verificar se há categóricas restantes
print("\n5. Categóricas restantes (exceto IDs e target):")
cat_cols = df_final.select_dtypes('object').columns
cat_cols = [c for c in cat_cols if c not in ['customerID', 'Churn']]
print(f"   {len(cat_cols)} colunas")
if len(cat_cols) > 0:
    print(f"    {cat_cols}")
```

# Análise Exploratória Avançada

**Agora podemos fazer análises que não eram possíveis antes:**

**</> Python**

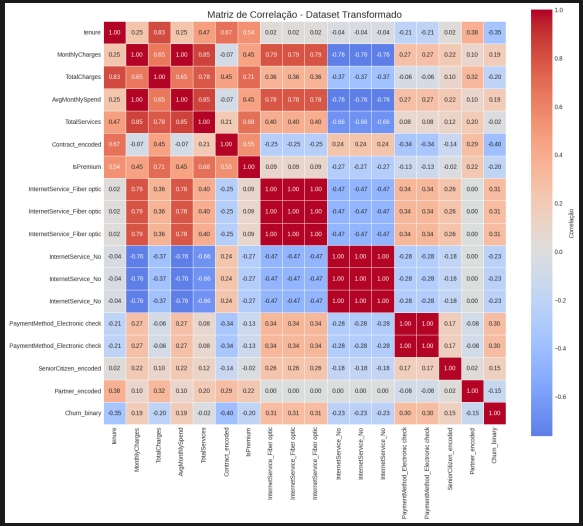
```python
# Matriz de correlação completa
import seaborn as sns
import matplotlib.pyplot as plt

# Selecionar features relevantes
analysis_features = [
    'tenure', 'MonthlyCharges', 'TotalCharges',
    'AvgMonthlySpend', 'TotalServices',
    'Contract_encoded', 'IsPremium',
    'InternetService_Fiber optic', 'InternetService_No',
    'PaymentMethod_Electronic check',
    'SeniorCitizen_encoded', 'Partner_encoded',
    'Churn_binary'
]
```

# Análise Exploratória Avançada (cont.)

**Python**

```python
# Calcular correlações
corr_matrix = df_final[analysis_features].corr()

# Heatmap
fig, ax = plt.subplots(figsize=(14, 12))
sns.heatmap(corr_matrix, annot=True, fmt='.2f', cmap='coolwarm',
            center=0, square=True, linewidths=0.5,
            cbar_kws={'label': 'Correlação'}, ax=ax)
ax.set_title('Matriz de Correlação - Dataset Transformado', fontsize=16)
plt.tight_layout()
plt.show()
```

# Análise Exploratória Avançada (cont.)



# Insights dos Dados Transformados

**Python**

```python
# Top correlações com Churn
print("=== TOP CORRELACOES COM CHURN ===\n")

correlations = corr_matrix['Churn_binary'].drop('Churn_binary')
correlations_abs = correlations.abs().sort_values(ascending=False)

print("Positivas (aumentam churn):")
positive = correlations[correlations > 0].sort_values(ascending=False)
for feat, corr in positive.head(5).items():
    print(f"  {feat}: {corr:.3f}")
```

# Insights dos Dados Transformados (cont.)

**Python**

```python
print("\nNegativas (reduzem churn):")
negative = correlations[correlations < 0].sort_values()
for feat, corr in negative.head(5).items():
    print(f"  {feat}: {corr:.3f}")

# Insights de negócio
print("\n=== INSIGHTS DE NEGOCIO ===")
print("  Contratos longos (Contract_encoded) reduzem churn")
print("  Mais serviços (TotalServices) reduz churn")
print("  Fibra óptica aumenta churn (provavelmente por preço)")
print("  Clientes premium têm menos churn")
```

# Comparação: Antes vs Depois das Transformações

**Python**

```python
# Visualizar impacto das transformações
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# 1. Antes: variáveis em escalas diferentes
axes[0,0].scatter(df['tenure'], df['TotalCharges'], alpha=0.3)
axes[0,0].set_xlabel('tenure (0-72)')
axes[0,0].set_ylabel('TotalCharges (0-8684)')
axes[0,0].set_title('ANTES: Escalas Diferentes')

# 2. Depois: variáveis normalizadas
axes[0,1].scatter(df_final['tenure'], df_final['TotalCharges'], alpha=0.3)
axes[0,1].set_xlabel('tenure (normalizado)')
axes[0,1].set_ylabel('TotalCharges (normalizado)')
axes[0,1].set_title('DEPOIS: Escalas Comparáveis')
```

# Comparação: Antes vs Depois das Transformações (cont.)

**Python**

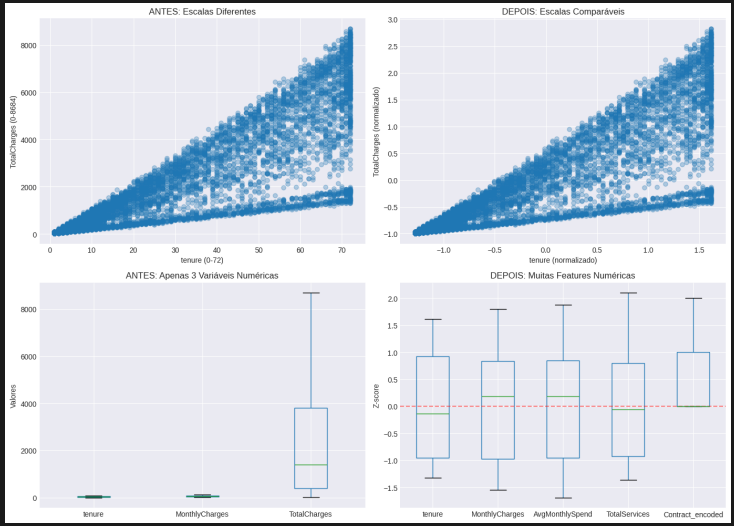
```python
# 3. Antes: limitado a numéricas originais
original_numeric = ['tenure', 'MonthlyCharges', 'TotalCharges']
df[original_numeric].boxplot(ax=axes[1,0])
axes[1,0].set_title('ANTES: Apenas 3 Variáveis Numéricas')
axes[1,0].set_ylabel('Valores')
```

# Comparação: Antes vs Depois das Transformações (cont.)

**Python**

```python
# 4. Depois: muitas features numéricas
analysis_subset = ['tenure', 'MonthlyCharges', 'AvgMonthlySpend',
                   'TotalServices', 'Contract_encoded']
df_final[analysis_subset].boxplot(ax=axes[1,1])
axes[1,1].set_title('DEPOIS: Muitas Features Numéricas')
axes[1,1].set_ylabel('Z-score')
axes[1,1].axhline(y=0, color='r', linestyle='--', alpha=0.5)
plt.tight_layout()
plt.show()
print("\n=== GANHOS ===")
print("  Variáveis comparáveis visualmente")
print("  Features criadas revelam padrões")
print("  Todas as variáveis são analisáveis numericamente")
print("  Pronto para análises avançadas e ML (futuro)")
```

# Comparação: Antes vs Depois das Transformações (cont.)



# Conexão com Machine Learning

**Como tudo isso se conecta ao ML:**

**O que fizemos hoje:**

- Feature Engineering: criar variáveis úteis
- Encoding: converter categorias em números
- Normalização: colocar em escalas comparáveis
- **Objetivo:** análise exploratória avançada

**Em Machine Learning (cursos futuros):**

- Usaremos *exatamente o mesmo pipeline*
- Mas ao invés de análise manual...
- Treinaremos algoritmos para aprender padrões automaticamente
- Ex: algoritmo aprende que ”Contract_encoded alto + TotalServices alto = baixo churn”
- Depois aplica em clientes novos para prever risco de churn

# Conexão com Machine Learning

**Nota Importante**

Transformação é pré-requisito tanto para análise quanto para ML!
Você já domina a base essencial.

# Checklist Final de Transformação

**Antes de considerar dados prontos:**

1. ☐ **Feature Engineering aplicado**
   - Features úteis criadas e validadas
   - Correlação com target verificada
   - Faz sentido para o negócio
2. ☐ **Encoding completo**
   - Nenhuma coluna categórica restante (exceto IDs)
   - Estratégia apropriada para cada tipo
   - Ordinais preservam ordem, nominais viram One-Hot
3. ☐ **Normalização aplicada (se necessário)**
   - Variáveis em escalas comparáveis
   - Técnica apropriada escolhida
   - Parâmetros salvos para dados futuros

# Checklist Final de Transformação

**Antes de considerar dados prontos:**

4. ☐ **Validações passam**
   - Missing values tratados
   - Ranges fazem sentido
   - Apenas dados numéricos
5. ☐ **Pipeline documentado**
   - Código organizado e comentado
   - Reproduzível em novos dados
   - Decisões justificadas

# Recap: Aula 10 Completa

**O que cobrimos:**

**Bloco 0: Preview de ML**

- Contexto: o que é ML e por que transformações importam

**Bloco 1: Feature Engineering**

- 12 técnicas para criar variáveis derivadas
- Validação através de correlações e visualizações

**Bloco 2: Encoding**

- Label, One-Hot, Binary encoding
- Estratégias para diferentes tipos de variáveis

# Recap: Aula 10 Completa

**O que cobrimos:**

**Bloco 3: Normalização**

- Min-Max, Standardization, Robust Scaling
- Comparabilidade e visualização

**Bloco 4: Pipeline**

- Integração completa
- Análise exploratória avançada

# Habilidades Adquiridas

**Você agora é capaz de:**

**Técnicas:**

- ✓ Criar features úteis sistematicamente
- ✓ Encodar qualquer tipo de variável categórica
- ✓ Normalizar dados apropriadamente
- ✓ Construir pipeline reproduzível

**Análise:**

- ✓ Validar transformações através de EDA
- ✓ Identificar padrões em dados transformados
- ✓ Interpretar correlações e visualizações
- ✓ Extrair insights acionáveis

# Habilidades Adquiridas

**Boas Práticas:**

- ✓ Documentar decisões
- ✓ Separar dados para validação
- ✓ Pensar em conhecimento de domínio
- ✓ Criar código reproduzível

# Aplicação Prática

**Como usar essas habilidades:**

**Em projetos de análise:**

- Preparar dados para análises quantitativas
- Criar visualizações mais informativas
- Identificar fatores associados a outcomes
- Gerar recomendações baseadas em dados

**No mercado de trabalho:**

- Analista de Dados: preparação de relatórios
- Cientista de Dados: base para ML
- Business Intelligence: dashboards avançados
- Pesquisa: análises estatísticas robustas

# Aplicação Prática

**Próximos passos:**

- Praticar com diferentes datasets
- Explorar domínios novos
- Combinar com técnicas de visualização avançada
- Preparação para cursos de ML

# Preview: Aula 11 - Séries Temporais

**Tema: Séries Temporais**

- Dados que variam ao longo do tempo
- Exemplos: vendas mensais, temperatura diária, preços de ações

**O que aprenderemos:**

- Manipulação de timestamps com Pandas
- Resampling e agregações temporais
- Decomposição de séries (tendência, sazonalidade)
- Visualizações temporais
- Rolling windows e médias móveis

**Conexão com hoje:**

- Usaremos as mesmas técnicas de transformação
- Mas adaptadas para dados temporais
- Feature engineering temporal (lag features, diferenças)

# Prática em casa

**Exercícios sugeridos:**

**1. Aplicar pipeline a novo dataset**

- Escolha dataset do Kaggle ou UCI ML Repository
- Aplique todas as transformações aprendidas
- Compare antes/depois

**2. Criar features criativas**

- Para dataset Telco: invente 5 features novas
- Valide com correlação e visualização
- Justifique cada uma

# Prática em casa

**Exercícios sugeridos:**

**3. Experimentar diferentes scalings**

- Compare visualmente Min-Max vs Standard vs Robust
- Em dataset com outliers
- Documente diferenças

**4. Refinar pipeline**

- Adicionar tratamento de erros
- Logging de transformações
- Testes automatizados

# A Seguir: Aula Prática

**Exercício Prático**

**Tempo:** 60 minutos
**Entrega:** via Moodle (notebook)

**Tarefas:**

1. (atualizado durante a aula)

**Notebook:** Disponível no Moodle

Obrigado!