Programação para Ciência de Dados

Pré-processamento de Dados - Limpeza e Tratamento

Arthur Casals

4 de Novembro de 2025

Avisos

- Última entrega - prática de 14/11 (sexta-feira)

Projeto Integrador Final

Informações Gerais

O que é:

- Trabalho final que integra **todo o conteúdo do curso**
- Análise completa de dados reais
- Simulação de cenário profissional
- Trabalho **INDIVIDUAL**

Peso e Pontuação:

- **30% da nota final**
- Total: 100 pontos base

Projeto Integrador Final

Informações Gerais

Divulgação do enunciado:

- Enunciado completo: **Última aula prática (14/11/25)**
- Materiais de apoio no Moodle
- Slides detalhados
- Rubrica de avaliação

Resultado esperado:

- Notebook Jupyter (Google Colab)
- Tempo estimado: **5-6 horas**
- Entrega: **17/11/25, 18:00 (São Paulo)**
- Entrega feita via Moodle (arquivo .ipynb)

O que Será Avaliado

Competências Técnicas (70%):

- Python, NumPy, Pandas
- Análise exploratória de dados
- Pré-processamento e limpeza
- Feature engineering
- Análise multidimensional
- Visualização de dados
- Qualidade e eficiência do código

Soft Skills (30%):

- **Documentação**
  - Clareza nas explicações
  - Estrutura organizada
  - Interpretação de resultados
- **Visão de Negócio**
  - Comunicação para não-técnicos
  - Priorização estratégica
  - Recomendações acionáveis

⚠ Atenção

O projeto avalia não apenas **o que você sabe fazer**, mas também **como você comunica** seus achados e insights.

Como Funciona a Avaliação

Avaliação em Dois Níveis:

1. Verificação Automática

- Via Otter-Grader
- Verifica requisitos mínimos:
  - Estrutura correta
  - Elementos obrigatórios presentes
  - Notebook executa sem erros
- Garante que você atendeu o básico

2. Avaliação Manual

- Feita pelo professor
- Avalia qualidade e profundidade
- Diferencial entre notas

Critérios de Excelência:

- Análises **profundas** (não superficiais)
- Insights **não óbvios**
- Interpretações **claras e conectadas**
- Código **limpo e eficiente**
- Comunicação **adaptada ao público**
- Recomendações **específicas e implementáveis**

Nota Importante

Não basta fazer, tem que fazer **bem feito** e **comunicar bem**.

Como Se Preparar Agora

Revisão de Conteúdo:

- Revise notebooks das aulas anteriores
- Refaça exercícios das práticas independentes
- Pratique especialmente:
  - Pandas: groupby, pivot, merge
  - EDA completa
  - Visualizações com Seaborn
  - Feature engineering
  - Aulas 9-12

Habilidades Não-Técnicas:

- Pratique escrever textos explicativos
- Interprete visualizações em voz alta

Organize Seu Tempo:

- O projeto precisa de **5-6 horas**
- **Não deixe para última hora**

Recursos Disponíveis:

- Horário de atendimento (use!)
- Fórum do Moodle
- Material completo das aulas
- Documentação oficial

Agenda

- Introdução: Por que Limpar Dados?
- Bloco 1: Identificando Dados Faltantes
- Bloco 2: Estratégias de Imputação
- Bloco 3: Limpeza e Validação de Dados
- Bloco 4: Pipeline Completo e Documentação

Bloco 0

Introdução e Contexto

A Realidade dos Dados

Dados do mundo real são SEMPRE sujos!

O que você vai encontrar:

- Valores faltantes (missing data)
- Erros de digitação
- Formatos inconsistentes
- Duplicatas
- Outliers problemáticos
- Dados impossíveis (idade = -5, área = 0)
- Unidades misturadas (metros e pés)

⚠ Atenção

Estudos mostram: 80% do tempo em Data Science é gasto limpando dados!

Por que Limpeza de Dados Importa?

Garbage In, Garbage Out (GIGO)

Impacto de dados sujos:

- **Análises erradas**
  - Conclusões incorretas
  - Decisões ruins de negócio
  - Perda de credibilidade
- **Modelos ruins**
  - Baixa acurácia
  - Previsões não confiáveis
  - Overfitting em ruído
- **Perda de tempo**
  - Retrabalho constante
  - Debugging difícil
  - Frustração da equipe

Impacto Financeiro: Exemplo Real

Caso: Empresa de E-commerce

Problema:

- Sistema de recomendação usando dados sujos
- 30% dos produtos com descrições incompletas
- Preços com erros de digitação (R$ 1.0 em vez de R$ 100)
- Categorias inconsistentes

Resultado:

- Recomendações irrelevantes
- Clientes frustrados
- Perda de vendas: R$ 2 milhões/ano

Solução:

- Investimento em limpeza: R$ 100 mil
- ROI: 20x em um ano!

Caso Real: Preços de Imóveis

Cenário: Startup de análise imobiliária

Problemas encontrados nos dados:

- **Missing:** 15% dos preços faltando
- **Formato:** Áreas como ”N/A”, ”unknown”, ”???”
- **Impossíveis:** Quartos = 0, -1, 99
- **Inconsistência:** Datas DD/MM vs MM/DD
- **Duplicatas:** 5% registros repetidos
- **Outliers:** Preços R$ 1 ou R$ 999.999.999
- **Texto:** Bairros com variações de escrita

Resultado:

- Modelo inicial: erro de 80% (inutilizável)
- Após limpeza: erro de 15% (excelente)

Estatísticas Sobre Qualidade de Dados

Pesquisas revelam:

- **80-90%** do tempo é gasto em preparação de dados
- **1 em 3** analistas cita dados ruins como maior problema
- **40%** das iniciativas de BI falham por dados ruins
- Custo anual de dados ruins: **trilhões de dólares** globalmente

Tipos de problemas mais comuns:

1. Valores faltantes (presente em 90% dos datasets)
2. Inconsistências de formato (70%)
3. Duplicatas (50%)
4. Valores impossíveis (40%)
5. Erros de digitação (60%)

Pipeline de Pré-processamento

Ordem típica das operações:

`1. Carregar Dados`
Ler arquivo, conferir formato
↓
`2. Exploração Inicial (EDA)`
head(), info(), describe()
↓
`3. Identificar Problemas`
Missing, outliers, duplicatas
↓

Pipeline de Pré-processamento

3. Identificar Problemas
Missing, outliers, duplicatas

↓

4. Tratar Valores Faltantes (HOJE)
Imputação, remoção

↓

5. Limpar Inconsistências (HOJE)
Padronização, validação

↓

6. Transformar (Próxima Aula)
Encoding, normalização

Tipos de Problemas de Qualidade

Framework de qualidade de dados:

1. Completude (Completeness)
- Dados faltantes, valores nulos
- **Exemplo:** 20% dos emails estão vazios

2. Validade (Validity)
- Valores fora do range esperado
- **Exemplo:** Idade = 200 anos

3. Consistência (Consistency)
- Formatos diferentes para mesma informação
- **Exemplo:** ”SP”, ”São Paulo”, ”Sao Paulo”

4. Unicidade (Uniqueness)
- Registros duplicados
- **Exemplo:** Mesmo cliente cadastrado 3 vezes

Tipos de Problemas de Qualidade

5. Acurácia (Accuracy)

Definição:

- Dados refletem a realidade?
- Medições estão corretas?

Exemplos de problemas:

- Endereço cadastrado está desatualizado
- Sensor descalibrado gerando medições erradas
- Dados transcritos incorretamente
- Conversão de unidades errada (km $\rightarrow$ m)

Como detectar:

- Validação com fonte externa
- Verificação amostral manual
- Comparação com benchmarks conhecidos

Dataset de Hoje: Housing Prices

Dados de preços de imóveis com problemas reais
Variáveis principais:

- **price:** Preço do imóvel (R$)
- **area:** Área total em m²
- **bedrooms:** Número de quartos
- **bathrooms:** Número de banheiros
- **age:** Idade do imóvel (anos desde construção)
- **neighborhood:** Bairro/região
- **condition:** Estado de conservação
- **garage_spaces:** Vagas de garagem
- **has_garden:** Possui jardim (sim/não)

Dataset de Hoje: Housing Prices

Tamanho:

- 3000 registros
- 9 variáveis

Origem:

- The Ames Housing Dataset
- Subconjunto de features ($82 \rightarrow 9$)
- Poluído propositalmente

Problemas Inseridos no Dataset

Problemas propositalmente incluídos:
Valores Faltantes:

- price: 12% missing
- area: 8% missing
- age: 15% missing
- neighborhood: 5% missing

Inconsistências:

- Bairros com nomes variados: ”Centro”, ”centro”, ”CENTRO”
- condition: ”Excelente”, ”Muito Bom”, ”Bom”, etc.
- Espaços extras em strings

Problemas Inseridos no Dataset

Problemas propositalmente incluídos:
Valores Impossíveis:

- bedrooms = -1, 0, 99
- area = 0, 5, 10000 m²
- age = -5, 250 anos
- price = R$ 1, R$ 999999999

Objetivos de Aprendizado

Conhecimento:

- Identificar tipos de problemas de qualidade
- Classificar dados faltantes (MCAR, MAR, MNAR)
- Conhecer múltiplas estratégias de imputação

Habilidades:

- Aplicar técnicas de imputação apropriadas
- Limpar inconsistências de formato
- Validar ranges e tipos de dados
- Construir pipeline de limpeza reproduzível

Atitudes:

- Sempre documentar decisões
- Ser conservador com remoção de dados
- Validar resultados sistematicamente

Ferramentas de Hoje

Pandas: Nossa ferramenta principal

Detecção de problemas:

- `isnull()`, `notnull()`, `isna()`
- `duplicated()`
- `describe()`, `value_counts()`

Tratamento de missing:

- `fillna()`, `dropna()`
- `interpolate()`
- `groupby().transform()`

Ferramentas de Hoje

Limpeza de strings:

- `str.strip()`, `str.lower()`, `str.upper()`
- `str.replace()`, `str.extract()`

Validação e conversão:

- `astype()`, `to_numeric()`, `to_datetime()`
- `between()`, `isin()`

Filosofia da Limpeza de Dados

1. Sempre documente suas decisões

- Por que removeu ou alterou algo?
- Qual critério utilizou?
- Quantos registros foram afetados?

2. Mantenha os dados originais intactos

- Sempre trabalhe em uma cópia
- Possibilidade de reverter decisões
- Comparar antes/depois

3. Seja conservador

- Dados são valiosos
- Não remova sem razão forte
- Prefira transformar a deletar

Filosofia da Limpeza de Dados (cont.)

4. Entenda o contexto do negócio

- Consulte especialistas do domínio
- O que faz sentido no negócio?
- Há regras específicas do setor?

5. Seja transparente

- Reporte tudo que foi feito
- Mostre impacto das decisões
- Permita reprodução do processo

6. Valide, valide, valide

- Sempre verifique resultados
- Compare distribuições antes/depois
- Testes de sanidade (sanity checks)

Fluxo Sistemático de Limpeza

Processo iterativo em 4 fases:

FASE 1: Auditoria Inicial (Assess)

- Carregar dados
- Dimensões: quantos registros? Features?
- Tipos de dados
- Primeira inspeção visual

FASE 2: Diagnóstico (Diagnose)

- Identificar todos os problemas
- Quantificar missing, duplicatas, outliers
- Classificar severidade
- Priorizar o que atacar

Fluxo Sistemático de Limpeza (cont.)

FASE 3: Planejamento (Plan)

- Definir estratégias para cada problema
- Estabelecer critérios de validação
- Criar regras de negócio
- Decidir ordem de execução

FASE 4: Execução e Validação (Execute & Validate)

- Aplicar transformações
- Verificar resultados após cada passo
- Comparar antes/depois
- Documentar mudanças
- Gerar relatório final

Fluxo Sistemático de Limpeza (cont.)

Nota Importante

Este é um processo iterativo - você voltará às fases anteriores!

Setup Inicial: Imports

Python

```python
# Imports necessarios
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
# Configuracoes de display
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', 100)
pd.set_option('display.float_format', '{:.2f}'.format)
# Configuracoes de plots
plt.style.use('seaborn-v0_8-darkgrid')
sns.set_palette("husl")
print("Bibliotecas carregadas com sucesso!")
```

Carregar Dados: Primeira Etapa

Python

```python
# Carregar dataset
df = pd.read_csv('housing_prices_dirty.csv')
print("="*50)
print("DADOS CARREGADOS")
print("="*50)
print(f"\nShape: {df.shape}")
print(f"Linhas: {df.shape[0]}")
print(f"Colunas: {df.shape[1]}")
print(f"\nMemoria utilizada: {df.memory_usage(deep=True).sum() /
    1024**2:.2f} MB")

# SEMPRE salve uma copia dos dados originais
df_original = df.copy()
print("\nCopia de seguranca criada!")
```

Carregar Dados: Primeira Etapa (cont.)

Python

```
# ================================================
# DADOS CARREGADOS
# ================================================
#
# Shape: (3076, 9)
# Linhas: 3076
# Colunas: 9
#
# Memoria utilizada: 0.74 MB
#
# Copia de seguranca criada!
```

Inspeção Visual: Primeiras Linhas

Python

```python
# Ver primeiras linhas
print("=== PRIMEIRAS 5 LINHAS ===")
print(df.head())

# Ver ultimas linhas (detectar problemas no fim)
print("\n=== ULTIMAS 5 LINHAS ===")
print(df.tail())

# Amostra aleatoria (visao geral melhor)
print("\n=== AMOSTRA ALEATORIA (10 linhas) ===")
print(df.sample(10, random_state=42))

# Sempre olhe head(), tail() E sample() - cada um revela coisas diferentes
```

Info: Visão Geral do Dataset

Python

```python
# Informacoes gerais
print("=== INFORMACOES DO DATASET ===")
df.info()

# Interpretacao:
# - RangeIndex: indices numericos
# - Data columns: numero de colunas
# - Non-Null Count: quantos NAO sao nulos
# - Dtype: tipo de cada coluna
# - memory usage: memoria RAM usada
```

Info: Visão Geral do Dataset (cont.)

O que observar:

- Colunas com Non-Null Count < total $\rightarrow$ têm dados faltantes!
- Tipos incorretos (idade como object?)
- Uso excessivo de memória

Describe: Estatísticas Descritivas

Python

```python
# Estatisticas das numericas
print("=== ESTATISTICAS NUMERICAS ===")
print(df.describe())

# Estatisticas das categoricas
print("\n=== ESTATISTICAS CATEGORICAS ===")
print(df.describe(include=['object', 'category']))

# Transpor para melhor visualizacao
print("\n=== TRANSPOSTO ===")
print(df.describe().T)
```

Describe: Estatísticas Descritivas

O que buscar:

- Min/max estranhos (negativos, zeros, muito altos)
- Média muito diferente da mediana (assimetria)
- Desvio padrão muito alto (outliers?)

Tipos de Dados: Verificação

Python

```python
# Ver tipos de cada coluna
print("=== TIPOS DE DADOS ===")
print(df.dtypes)

# Contar tipos
print("\n=== CONTAGEM POR TIPO ===")
print(df.dtypes.value_counts())

# Separar por tipo
numericas = df.select_dtypes(include=['number']).columns
categoricas = df.select_dtypes(include=['object']).columns

print(f"\nNumericas ({len(numericas)}): {list(numericas)}")
print(f"Categoricas ({len(categoricas)}): {list(categoricas)}")
```

Valores Únicos: Primeira Análise

Python

```python
# Contar valores unicos em cada coluna
print("=== VALORES UNICOS POR COLUNA ===")
for col in df.columns:
    n_unique = df[col].nunique()
    pct_unique = (n_unique / len(df)) * 100
    print(f"{col:20s}: {n_unique:4d} unicos ({pct_unique:5.1f}%)")

# Valores unicos das categoricas
print("\n=== CATEGORIAS UNICAS ===")
for col in categoricas:
    print(f"\n{col}:")
    print(df[col].value_counts())
```

Checklist de Auditoria Inicial

Antes de começar a limpar, responda:

1. ☐ Quantas linhas e colunas tenho?
2. ☐ Quais são numéricas? Quais são categóricas?
3. ☐ Os tipos de dados estão corretos?
4. ☐ Há valores faltantes? Em quais colunas?
5. ☐ Quais são os ranges das variáveis numéricas?
6. ☐ Há valores impossíveis (negativos, zeros, extremos)?
7. ☐ As categóricas têm quantas categorias?
8. ☐ Há inconsistências visíveis (espaços, maiúsculas)?
9. ☐ Há duplicatas aparentes?
10. ☐ Os dados fazem sentido no contexto do negócio?

⚠ Atenção

Só avance quando puder responder todas essas perguntas!

Exercício Guiado: Auditoria Completa

Exercício Prático

Execute a auditoria inicial do Housing dataset:

1. Carregue os dados e crie cópia de segurança
2. Execute todos os comandos de inspeção:

```
df.head()
df.tail()
df.sample(10)
df.info()
df.describe()
df.dtypes
```

3. Anote TODOS os problemas que identificar
4. Classifique por tipo (missing, invalid, inconsistent)
5. Estime severidade (crítico, moderado, leve)

Transição para Próximo Bloco

Recapitulando Introdução:

Aprendemos:

- Por que limpeza é crítica (80% do tempo!)
- Tipos de problemas de qualidade
- Filosofia e princípios de limpeza
- Workflow sistemático em 4 fases
- Como fazer auditoria inicial completa

Agora vamos para o Bloco 1:

- Identificação detalhada de dados faltantes
- Classificação: MCAR, MAR, MNAR
- Análise de padrões de missing
- Decisão: quando remover vs imputar

Bloco 1

Identificando e Entendendo Dados Faltantes

O que São Dados Faltantes?

Missing data = ausência de valor onde deveria existir um

Como aparecem em diferentes contextos:

- **Pandas/NumPy:** `NaN` (Not a Number)
- **Python nativo:** `None`
- **Bancos SQL:** `NULL`
- **Excel:** Células vazias
- **CSV:** (string vazia)
- **Códigos especiais:** -999, 9999, "N/A", "unknown", "?"

⚠ Atenção

Cuidado: nem tudo que parece missing é missing! Zero pode ser válido.

Por que Valores Ficam Faltando?

Razões técnicas:

- Sensor/equipamento quebrado
- Falha na transmissão de dados
- Erro no sistema de coleta
- Problema de integração entre sistemas
- Corrupção de arquivo

Razões humanas:

- Esquecimento ao preencher formulário
- Erro de digitação
- Informação não disponível no momento
- Pessoa não quis responder (privacidade)
- Campo não aplicável ao caso

Exemplos Reais de Missing Data

Saúde:

- Paciente não fez exame de sangue $\rightarrow$ colesterol faltando
- Sensor de frequência cardíaca desconectou

E-commerce:

- Cliente não preencheu telefone no cadastro
- Produto sem categoria definida

Financeiro:

- Renda não declarada em formulário
- Valor de transação perdido em falha de rede

Imóveis (nosso caso):

- Preço não definido (negociação em andamento)
- Área não medida ainda
- Idade desconhecida (imóvel muito antigo)

Identificar Missing: Comandos Básicos

Python

```python
# Verificar se ha valores faltantes
print("=== HA VALORES FALTANTES? ===")
print(df.isnull().any())
# True = tem missing, False = sem missing

# Contar missing por coluna
print("\n=== CONTAGEM DE MISSING ===")
print(df.isnull().sum())

# Total de missing no dataset inteiro
total_missing = df.isnull().sum().sum()
print(f"\nTotal de valores faltantes: {total_missing}")
```

Percentual de Missing

Python

```python
# Calcular percentual de missing
print("=== PERCENTUAL DE MISSING ===")
missing_count = df.isnull().sum()
missing_pct = (missing_count / len(df)) * 100
# Criar DataFrame resumo
missing_summary = pd.DataFrame({
    'Missing_Count': missing_count,
    'Percentage': missing_pct
})
# Ordenar por percentual (decrescente)
missing_summary = missing_summary.sort_values(
    'Percentage', ascending=False
)
# Mostrar apenas colunas com missing
print(missing_summary[missing_summary['Missing_Count'] > 0])
```

Interpretando Percentuais de Missing

Guia de decisão baseado em %:

0-5%: PROBLEMA PEQUENO

- ✓ Pode remover linhas com segurança
- ✓ Ou imputar com método simples
- Impacto mínimo nas análises
- Decisão rápida

5-20%: PROBLEMA MODERADO

- Remover pode perder muita informação
- Imputação deve ser cuidadosa
- Analise padrão de missing (é aleatório?)
- Considere imputação por grupos
- Documente estratégia escolhida

Interpretando Percentuais (continuação)

20-40%: PROBLEMA SÉRIO

- ✗ Muito missing para ignorar
- ✗ Imputação pode distorcer resultados
- Opções:
  - Coletar mais dados
  - Usar modelos que lidam com missing (alguns ML)
  - Criar flag "was_missing"+ imputação
  - Considerar remover a variável

40%: PROBLEMA CRÍTICO

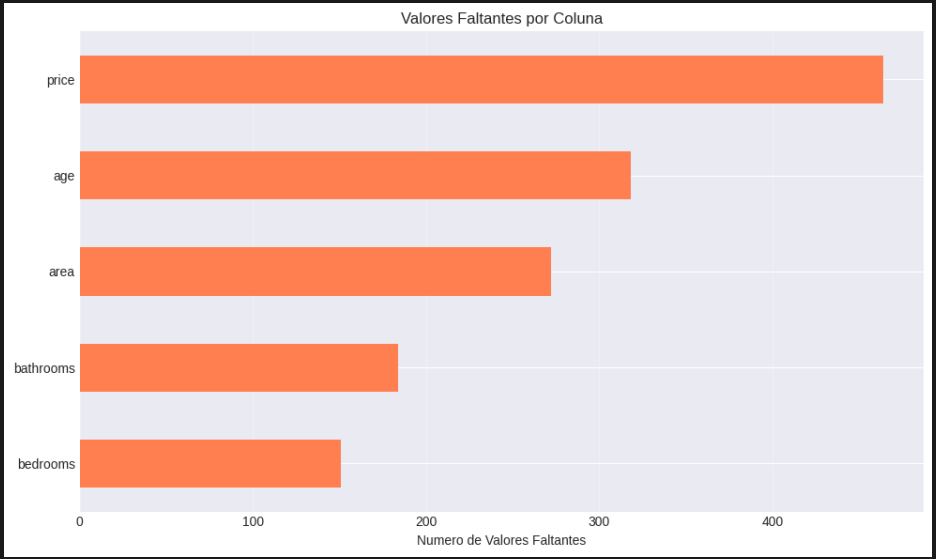
- ✗ Variável praticamente inútil
- ✗ Mais missing que dados!
- Recomendação: remover a coluna inteira
- Exceção: se missing é a informação (ex: "não respondeu")

Visualizar Missing: Gráfico de Barras

Python

```python
import matplotlib.pyplot as plt

# Criar grafico de missing
missing_data = df.isnull().sum()
missing_data = missing_data[missing_data > 0].sort_values()

fig, ax = plt.subplots(figsize=(10, 6))
missing_data.plot(kind='barh', ax=ax, color='coral')

ax.set_xlabel('Numero de Valores Faltantes')
ax.set_title('Valores Faltantes por Coluna')
ax.grid(True, alpha=0.3, axis='x')

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

Visualizar Missing: Gráfico de Barras

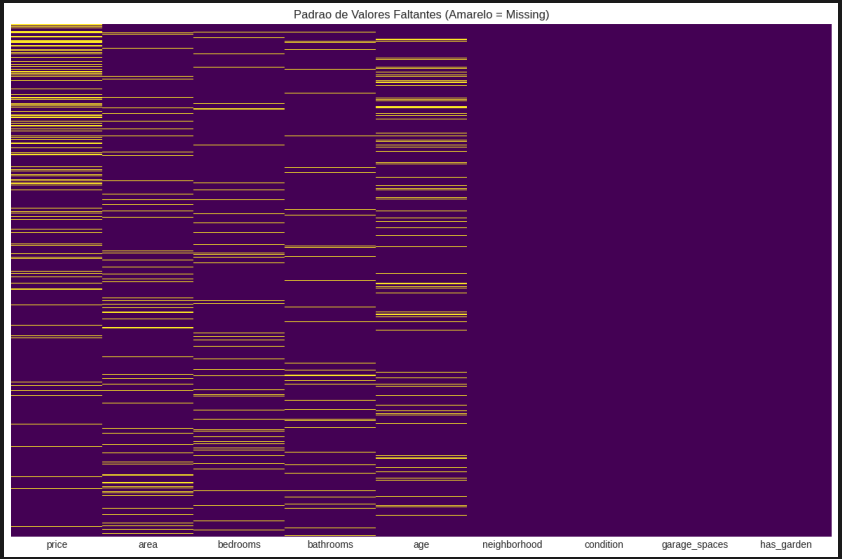


Visualizar Missing: Heatmap

</> Python

```python
import seaborn as sns
# Heatmap de valores faltantes
fig, ax = plt.subplots(figsize=(12, 8))
# Criar matriz: True = missing, False = presente
missing_matrix = df.isnull()
# Plotar (amarelo = missing, roxo = presente)
sns.heatmap(missing_matrix,
            yticklabels=False,
            cbar=False,
            cmap='viridis',
            ax=ax)
ax.set_title('Padrao de Valores Faltantes (Amarelo = Missing)')
plt.tight_layout()
plt.show()
```

Visualizar Missing: Heatmap



Interpretando o Heatmap de Missing

O que procurar no heatmap:

Padrões Verticais (colunas):

- Linhas amarelas contínuas $\rightarrow$ muitas linhas com missing
- Linhas esparsas $\rightarrow$ missing aleatório

Padrões Horizontais (linhas):

- Colunas amarelas $\rightarrow$ variável com muito missing
- Colunas roxas $\rightarrow$ variável completa

Padrões Correlacionados:

- Duas colunas amarelas nos mesmos lugares
- Significa: quando uma falta, outra também falta
- Pode indicar problema sistemático

Missing por Linha (Casos Completos)

Python

```python
# Quantas linhas estao completas (sem missing)?
complete_rows = df.notnull().all(axis=1).sum()
incomplete_rows = len(df) - complete_rows

print(f"Linhas completas: {complete_rows} ({complete_rows/len(df)*100:.1f}%)")
print(f"Linhas com missing: {incomplete_rows} ({incomplete_rows/len(df)*100:.1f}%)")

# Contar quantos missing por linha
missing_per_row = df.isnull().sum(axis=1)

print("\n=== DISTRIBUICAO DE MISSING POR LINHA ===")
print(missing_per_row.value_counts().sort_index())
```

Decisão: Remover vs Imputar

Árvore de decisão:

Quanto missing há?

↓

< 5% → Pode remover linhas

5-20% → Imputação recomendada

20% → Análise profunda necessária

↓

É variável crítica?

↓

SIM (ex: target) → Remover linhas

NÃO (ex: feature) → Imputar

Tipos de Missing: Introdução

Nem todo missing é igual!

Três categorias fundamentais:

1. **MCAR - Missing Completely At Random**
   - Ausência é 100% aleatória
   - Não relacionada a NADA (nem observado, nem não observado)
2. **MAR - Missing At Random**
   - Ausência relacionada a variáveis observadas
   - Podemos explicar o missing com outras colunas
3. **MNAR - Missing Not At Random**
   - Ausência relacionada ao próprio valor faltante
   - Missing porque o valor é alto/baixo/específico

MCAR: Missing Completely At Random

Características:

- Probabilidade de missing é igual para todos
- Ausência não depende de nenhuma variável
- É o tipo mais ”inofensivo”de missing
- Raro na prática

Exemplo clássico:

- Técnico derrubou café no equipamento
- Equipamento quebrou aleatoriamente
- Sorteio aleatório de quem responde pergunta
- Bug de software afetando amostras aleatórias

Implicação:

- ✓ Remover linhas não introduz viés
- ✓ Qualquer método de imputação funciona bem

MCAR: Exemplo Prático

Cenário: Medição de Pressão Arterial

Situação:

- Estudo com 1000 pacientes
- Medidor de pressão quebrou em dia específico
- 50 medições perdidas (5%)
- Não há padrão: jovens, idosos, saudáveis, doentes

ID	Idade	Peso	Pressão	Dia
1	25	70	120	01/10
2	30	75	NaN	15/10 ← quebrou
3	35	80	130	01/10
4	40	85	NaN	15/10 ← quebrou
5	45	90	125	02/10

MCAR: Exemplo Prático

Análise:

- Missing não relacionado a idade, peso ou pressão real
- É MCAR!

MAR: Missing At Random

Características:

- Ausência relacionada a outras variáveis observadas
- Podemos ”explicar” o missing com dados que temos
- Mais comum que MCAR
- Requer tratamento mais sofisticado

Exemplos:

- Homens não respondem pergunta sobre gravidez
- Jovens pulam pergunta sobre aposentadoria
- Pessoas sem carro deixam vazio ”marca do carro”
- Imóveis sem garagem $\rightarrow$ vagas = NaN

Implicação:

- Remover linhas pode introduzir viés
- ✓ Imputação por grupos funciona bem
- ✓ Podemos usar outras variáveis para prever

MAR: Exemplo Prático

Cenário: Pesquisa de Saúde

Situação:

- Pergunta sobre ”quantos filhos tem?”
- Muitos jovens (< 25 anos) não responderam
- Idosos responderam quase todos

ID	Idade	Sexo	Filhos
1	22	F	NaN ← jovem
2	24	M	NaN ← jovem
3	45	F	2
4	50	M	3
5	23	F	NaN ← jovem
6	48	F	1

MAR: Exemplo Prático

Análise:

- Missing relacionado à IDADE (observada)
- Jovens ainda não têm filhos ou não quiseram responder
- É MAR!

MNAR: Missing Not At Random

Características:

- Ausência relacionada ao próprio valor que falta
- Valor está faltando PORQUE é alto, baixo ou específico
- Tipo mais problemático
- Introduz viés sistemático

Exemplos:

- Pessoas com salário muito alto não revelam
- Deprimidos não respondem sobre humor
- Muito endividados omitem dívidas
- Alunos com notas baixas faltam à prova
- Imóveis muito caros sem preço (em negociação)

Implicação:

- ✗ Difícil de detectar
- ✗ Qualquer imputação pode ser enviesada
- ✗ Pode exigir coletar mais dados

MNAR: Exemplo Prático

Cenário: Pesquisa Salarial

Situação:

- Pergunta: ”Qual seu salário mensal?”
- Executivos e CEOs não responderam
- Porque salário é MUITO alto (privacidade)

ID	Cargo	Salário
1	Analista	R$ 5.000
2	Gerente	R$ 15.000
3	Diretor	NaN ← muito alto!
4	CEO	NaN ← muito alto!
5	Coordenador	R$ 10.000
6	VP	NaN ← muito alto!

MNAR: Exemplo Prático

Análise:

- Missing porque o salário é alto
- Não podemos ver o salário, mas sabemos que falta porque é alto
- É MNAR!

Comparação: MCAR vs MAR vs MNAR

Aspecto	**MCAR**	**MAR**	**MNAR**
Aleatoriedade	Completamente aleatório	Relacionado a observados	Relacionado ao próprio valor
Frequência	Raro	Comum	Moderado
Detectabilidade	Fácil	Moderada	Difícil
Viés ao remover	Nenhum	Possível	Sério
Imputação simples	Funciona bem	Funciona razoavelmente	Problemática
Tratamento	Qualquer método	Por grupos	Muito cuidadoso

Investigar Tipo de Missing: Análise Exploratória

Python

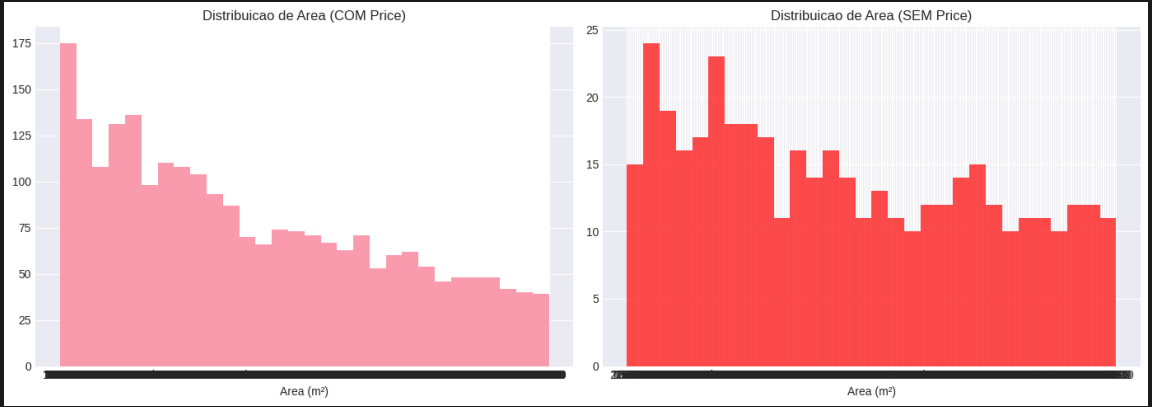
```python
# Comparar caracteristicas de quem tem vs nao tem missing
# Exemplo: price faltante
# Grupo COM price
with_price = df[df['price'].notnull()]
# Grupo SEM price
without_price = df[df['price'].isnull()]
print("=== COM PRECO ===")
print(with_price[['area', 'bedrooms', 'age']].describe())
print("\n=== SEM PRECO ===")
print(without_price[['area', 'bedrooms', 'age']].describe())

# Sao muito diferentes?  Pode ser MAR ou MNAR
# Sao similares?  Provavelmente MCAR
```

Teste Visual: Comparar Distribuições

Python

```python
import matplotlib.pyplot as plt
fig, axes = plt.subplots(1, 2, figsize=(14, 5))
# Distribuicao de area: COM price
axes[0].hist(with_price['area'].dropna(),
             bins=30, alpha=0.7, label='Com Price')
axes[0].set_title('Distribuicao de Area (COM Price)')
axes[0].set_xlabel('Area (m²)')
# Distribuicao de area: SEM price
axes[1].hist(without_price['area'].dropna(),
             bins=30, alpha=0.7, color='red',
             label='Sem Price')
axes[1].set_title('Distribuicao de Area (SEM Price)')
axes[1].set_xlabel('Area (m²)')
plt.tight_layout()
plt.show()
```

Teste Visual: Comparar Distribuições

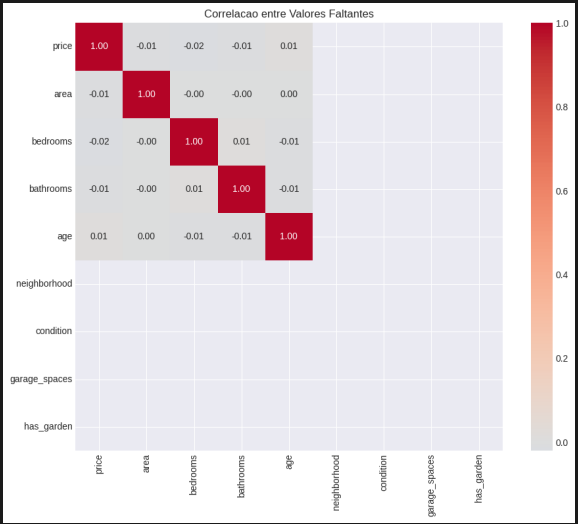


Correlação entre Missings

</> Python

```python
# Verificar se missings ocorrem juntos
# Criar matriz de correlacao de missing
missing_matrix = df.isnull().astype(int)
missing_corr = missing_matrix.corr()
# Plotar heatmap
import seaborn as sns
fig, ax = plt.subplots(figsize=(10, 8))
sns.heatmap(missing_corr, annot=True, fmt='.2f',
            cmap='coolwarm', center=0,
            square=True, ax=ax)
ax.set_title('Correlacao entre Valores Faltantes')
plt.tight_layout()
plt.show()
# Correlacao alta?  Missings ocorrem juntos
```

Correlação entre Missings



Como Identificar o Tipo na Prática

Checklist de investigação:

1. Consulte especialistas do domínio
- Como os dados foram coletados?
- Há razões conhecidas para missing?
- Processo de coleta introduz viés?

2. Análise exploratória
- Compare grupos com/sem missing
- Distribuições são similares? (MCAR)
- Diferem em variáveis observadas? (MAR)

3. Use lógica e contexto
- Missing faz sentido dado o valor?
- Há incentivo para não responder?
- É questão sensível?

Decisão de Tratamento por Tipo

Se identificou como MCAR:

- ✓ Pode remover linhas se <5%
- ✓ Imputação simples (média/mediana) funciona
- ✓ Sem grandes preocupações

Se identificou como MAR:

- Cuidado ao remover (pode introduzir viés)
- ✓ Use imputação por grupos
- ✓ Considere variáveis relacionadas
- ✓ Modelos preditivos podem ajudar

Se identificou como MNAR:

- ✗ Muito difícil de tratar bem
- Qualquer imputação pode ser enviesada
- Considere: criar flag, coletar mais dados, ou remover variável

Caso de Uso: Housing Prices Missing

Análise do nosso dataset:

price missing (12%):

- Hipótese: Imóveis em negociação ainda
- Pode ser MNAR se preços muito altos estão faltando
- Investigar: comparar area/bedrooms de missing vs não-missing

area missing (8%):

- Hipótese: Não foi medido ainda
- Provavelmente MCAR (aleatório)
- Pode imputar com mediana ou por bairro

age missing (15%):

- Hipótese: Imóveis muito antigos (sem registro)
- Pode ser MNAR!
- Cuidado: age faltante pode significar ”muito velho”

Criar Relatório de Missing

</> Python

```python
def generate_missing_report(df):
    """Gera relatorio completo de missing"""
    missing_count = df.isnull().sum()
    missing_pct = (missing_count / len(df)) * 100
    report = pd.DataFrame({
        'Column': df.columns,
        'Missing_Count': missing_count.values,
        'Missing_Pct': missing_pct.values,
        'Dtype': df.dtypes.values
    })
    report = report[report['Missing_Count'] > 0]
    report = report.sort_values('Missing_Pct', ascending=False)
    return report
# Usar
print(generate_missing_report(df))
```

Documentar Classificação de Missing

Python

```python
# Criar dicionario documentando tipo de missing
missing_classification = {
    'price': {
        'type': 'MNAR',
        'reason': 'Imoveis caros em negociacao',
        'evidence': 'Area maior em missing',
        'strategy': 'Imputar por grupo + flag'
    },
    'area': {
        'type': 'MCAR',
        'reason': 'Nao medido ainda',
        'evidence': 'Distribuicao similar',
        'strategy': 'Imputar com mediana'
    },
```

Documentar Classificação de Missing (cont.)

Python

```python
    'age': {
        'type': 'MNAR',
        'reason': 'Imoveis muito antigos sem registro',
        'evidence': 'Logica do negocio',
        'strategy': 'Valor alto fixo + flag'
    }
}
```

Estratégia Geral de Tratamento

Remoção (Deletion):

- Listwise: remover linha inteira se tem qualquer missing
- Pairwise: usar linhas completas para cada análise
- Quando: MCAR + pequeno % + muitos dados disponíveis

Imputação (Imputation):

- Simples: média, mediana, moda
- Por grupos: considerar outras variáveis
- Avançada: modelos preditivos
- Quando: MAR + % moderado + poucos dados

Manter como está:

- Alguns modelos aceitam missing (XGBoost, LightGBM)
- Missing é a informação (flag binária)
- Quando: MNAR + missing é relevante

Exercício Completo: Análise de Missing

Exercício Prático

Investigue completamente o missing no Housing:

1. Identifique todas as colunas com missing
2. Para cada uma:

```
# a) Calcule percentual
# b) Compare grupo com/sem missing
# c) Visualize distribuicoes
# d) Verifique correlacao com outras vars
```

3. Classifique como MCAR, MAR ou MNAR
4. Justifique sua classificação
5. Documente estratégia recomendada
6. Crie relatório consolidado

Resumo do Bloco 1

O que aprendemos:

Identificação:

- Como detectar missing: isnull(), sum()
- Calcular percentuais e criar relatórios
- Visualizar com gráficos e heatmaps

Classificação:

- MCAR: completamente aleatório
- MAR: relacionado a observados
- MNAR: relacionado ao próprio valor

Investigação:

- Comparar grupos com/sem missing
- Analisar correlações entre missings
- Consultar especialistas do domínio

Transição para Próximo Bloco

Agora que identificamos e classificamos missing:

Bloco 2: Estratégias de Imputação

- Remoção de linhas (quando e como)
- Imputação simples (média, mediana, moda)
- Imputação por grupos (groupby + transform)
- Forward fill e backward fill
- Interpolação
- Valor constante
- Criar flags de imputação
- Casos de uso práticos

Nota Importante

Aprenderemos COMO preencher os valores faltantes de forma inteligente!

Bloco 2

Estratégias de Imputação

O que é Imputação?

Imputação = preencher valores faltantes com estimativas

Por que imputar?

- Muitos algoritmos de ML não aceitam NaN
- Remover todas as linhas pode perder muita informação
- Informação parcial é melhor que nenhuma informação
- Manter tamanho do dataset

Quando NÃO imputar?

- > 50% da coluna é missing (mais missing que dados!)
- Missing é a informação (ex: não respondeu = desinteressado)
- Dados são MNAR com forte viés
- Você tem dados suficientes após remoção

O que é Imputação?

⚠ Atenção

Imputação é sempre uma estimativa - há incerteza!

Métodos de Imputação: Panorama

Nível 1: Remoção

- Remover linhas (dropna)
- Remover colunas

Nível 2: Imputação Simples

- Média, mediana, moda
- Valor constante

Nível 3: Imputação Avançada

- Forward fill / Backward fill
- Imputação por grupos
- Interpolação

Nível 4: Modelagem (não veremos hoje)

- Regressão, KNN, Random Forest

Estratégia 1: Remoção de Linhas (Listwise)

Quando usar:

- ✓ Você tem MUITOS dados (perder 5% é OK)
- ✓ Missing é pequeno (< 5%)
- ✓ Dados são MCAR
- ✓ Variável crítica está missing (ex: target)

Vantagens:

- Muito simples
- Rápido
- Sem estimativas (dados reais apenas)

Desvantagens:

- Perde informação
- Reduz tamanho do dataset
- Pode introduzir viés se não for MCAR
- Desperdiça dados parcialmente completos

Remoção: Implementação Básica

</> Python

```python
# Contar linhas antes
print(f"Linhas originais: {len(df)}")

# Remover todas as linhas com QUALQUER NaN
df_clean = df.dropna()
print(f"Apos dropna(): {len(df_clean)}")
print(f"Perdidos: {len(df) - len(df_clean)} linhas")

# Percentual perdido
pct_lost = ((len(df) - len(df_clean)) / len(df)) * 100
print(f"Percentual perdido: {pct_lost:.1f}%")

# CUIDADO: Se perder > 20%, reconsidere!
if pct_lost > 20:
    print("ALERTA: Perdendo muitos dados!")
```

Remoção: Opções Avançadas

Python

```python
# Remover apenas se TODAS as colunas forem NaN
df_clean = df.dropna(how='all')
# Use: linhas completamente vazias

# Remover se NaN em colunas especificas
df_clean = df.dropna(subset=['price', 'area'])
# Use: quando certas colunas sao criticas

# Remover se missing em N ou mais colunas
# (manter apenas se tem pelo menos 5 valores)
df_clean = df.dropna(thresh=5)

# Remover linhas, mas manter todas as colunas
df_clean = df.dropna(axis=0)  # axis=0 = linhas
```

Remoção: Caso de Uso - Análise de Satisfação de Clientes

Dados:

- 100.000 respostas de pesquisa
- Pergunta crítica: ”Nota de 0-10 para o produto”
- 3% não responderam esta pergunta
- Outras perguntas têm <1% missing

Decisão:

- Remover 3.000 linhas sem nota (variável crítica)
- Ainda restam 97.000 respostas (excelente!)
- Missing é pequeno e não introduz viés significativo
- Melhor que tentar ”adivinhar”a nota

Resultado:

- ✓ Dataset limpo com target válido
- ✓ Análise confiável

Estratégia 2: Imputação Simples - Conceito

Preencher com estatística central

1. **Média (mean):**
   - Soma de todos / quantidade
   - Para: dados numéricos simétricos
   - Sensível a outliers
2. **Mediana (median):**
   - Valor do meio
   - Para: dados numéricos com outliers
   - Mais robusta (RECOMENDADA!)
3. **Moda (mode):**
   - Valor mais frequente
   - Para: dados categóricos
   - Ou numéricos discretos

Imputação com Média

</> Python

```python
# Calcular media ANTES de preencher
df['area'] = pd.to_numeric(df['area'], errors='coerce')
mean_area = df['area'].mean()
print(f"Media de area: {mean_area:.2f} m²")
# Verificar quantos missing
n_missing = df['area'].isnull().sum()
print(f"Valores faltantes: {n_missing}")
# Preencher NaN com a media
df['area'] = df['area'].fillna(mean_area)
# Verificar se funcionou
print(f"Missing apos imputacao: {df['area'].isnull().sum()}")
# Deve ser 0!
print(f"\n{n_missing} valores imputados com media")
```

Imputação com Mediana (Recomendado)

Python

```python
# Mediana é mais robusta que media!
median_price = df['price'].median()
print(f"Mediana de price: R$ {median_price:,.2f}")

# Comparar com media
mean_price = df['price'].mean()
print(f"Media de price: R$ {mean_price:,.2f}")

# Se muito diferente, ha outliers!
diff_pct = abs(median_price - mean_price) / median_price * 100
if diff_pct > 10:
    print(f"{diff_pct:.1f}% diferenca! Use MEDIANA.")

# Imputar com mediana
df['price'] = df['price'].fillna(median_price)
```

Média vs Mediana: Quando Usar?

Use MÉDIA quando:

- Distribuição é simétrica (sino)
- Sem outliers significativos
- Quer refletir o ”total”
- Exemplo: altura, temperatura, notas

Use MEDIANA quando:

- Distribuição é assimétrica
- Há outliers
- Quer o valor ”típico”
- Exemplo: preços, salários, renda
- **EM CASO DE DÚVIDA: USE MEDIANA!**

Teste prático:

- Se média e mediana são similares $\rightarrow$ qualquer uma OK
- Se diferem muito $\rightarrow$ use mediana

Imputação com Moda

Python

```python
# Moda para variaveis CATEGORICAS
mode_neighborhood = df['neighborhood'].mode()[0]
print(f"Bairro mais comum: {mode_neighborhood}")

# Contar frequencias
print("\nDistribuicao de bairros:")
print(df['neighborhood'].value_counts())

# Imputar com moda
n_missing = df['neighborhood'].isnull().sum()
df['neighborhood'] = df['neighborhood'].fillna(mode_neighborhood)

print(f"\n\cmark  {n_missing} valores imputados com moda")

# NOTA: mode() retorna Series, por isso [0]
```

Imputar Múltiplas Colunas de Uma Vez

</> Python

```python
# Imputar todas as numericas com mediana
numeric_cols = df.select_dtypes(include=['number']).columns

for col in numeric_cols:
    if df[col].isnull().sum() > 0:
        median_val = df[col].median()
        n_missing = df[col].isnull().sum()
        df[col] = df[col].fillna(median_val)
        print(f"{col}: {n_missing} valores  $\rightarrow$  mediana {
    median_val:.2f}")
```

Imputar Múltiplas Colunas de Uma Vez (cont.)

</> Python

```python
# Imputar todas as categoricas com moda
cat_cols = df.select_dtypes(include=['object']).columns

for col in cat_cols:
    if df[col].isnull().sum() > 0:
        mode_val = df[col].mode()[0]
        n_missing = df[col].isnull().sum()
        df[col] = df[col].fillna(mode_val)
        print(f"{col}: {n_missing} valores  $\rightarrow$  moda '{mode_val
    }'")
```

Imputação Simples: Prós e Contras

Vantagens:

- ✓ Muito fácil de implementar
- ✓ Rápido (mesmo para milhões de linhas)
- ✓ Funciona razoavelmente para MCAR
- ✓ Mantém tamanho do dataset
- ✓ Não requer outras variáveis

Desvantagens:

- ✗ Reduz variabilidade (todos ficam iguais)
- ✗ Não considera relações entre variáveis
- ✗ Pode distorcer distribuição
- ✗ Subestima desvio padrão
- ✗ Não funciona bem para MAR ou MNAR

Problema da Imputação Simples: Exemplo

Dados originais (sem missing):

`10, 15, 20, 25, 30, 35, 40`
Média = 25 | Desvio = 10.6

Dados com missing:

`10, NaN, 20, NaN, 30, NaN, 40`

Após imputar com média (25):

`10, 25, 20, 25, 30, 25, 40`
Média = 25 (OK) | Desvio = 8.9 (✗ menor!)

⚠ Atenção

Imputação reduz artificialmente a variabilidade!

Estratégia 3: Imputação por Grupos

Ideia:

- Não usar média global
- Usar média do GRUPO a que pertence
- Mais sofisticado e realista

Quando usar:

- Dados são MAR (relacionados a outras variáveis)
- Há agrupamento natural nos dados
- Valores variam significativamente entre grupos

Exemplos:

- Área de imóvel: média POR BAIRRO
- Salário: média POR CARGO
- Gasto: média POR FAIXA ETÁRIA
- Preço: mediana POR CATEGORIA

Por que Imputação por Grupos é Melhor?

Cenário:

- **Centro:** Apartamentos pequenos (média 60 m²)
- **Subúrbio:** Casas médias (média 120 m²)
- **Condomínio:** Casas grandes (média 200 m²)

Missing: área de imóvel no Centro

Imputação global (ruim):

- Média geral: (60+120+200)/3 = 127 m²
- Imputar 127 m² no Centro
- ✗ Ruim!

Imputação por grupo (melhor):

- Média do Centro: 60 m²
- Imputar 60 m²
- ✓ Muito mais realista!

Imputação por Grupos: Implementação

</> Python

```python
# Imputar area com media POR BAIRRO
# (mais sofisticado que media geral!)
# ANTES: ver medias por bairro
print("Medias por bairro:")
print(df.groupby('neighborhood')['area'].mean())
# Imputar usando groupby + transform
df['area'] = df.groupby('neighborhood')['area'] \
    .transform(lambda x: x.fillna(x.mean()))

# O que faz:
# 1. Agrupa por neighborhood
# 2. Para cada grupo, calcula mean de area
# 3. Preenche NaN com mean do GRUPO
# 4. Transform retorna shape original
```

Imputação por Grupos: Mediana

</> Python

```python
# Usar MEDIANA por grupo (mais robusto)
# Imputar price com mediana por bairro
df['price'] = df.groupby('neighborhood')['price'] \
    .transform(lambda x: x.fillna(x.median()))
# Verificar resultado
print("Medianas de price por bairro:")
print(df.groupby('neighborhood')['price'].median())
# Se algum grupo tem TODOS missing?
# Pandas usa mediana global nesses casos!
# Ou especifique fallback:
global_median = df['price'].median()
df['price'] = df.groupby('neighborhood')['price'] \
    .transform(lambda x: x.fillna(x.median() if x.notna().any() else
    global_median))
```

Múltiplos Grupos: Hierárquico

</> Python

```python
# Imputar considerando MULTIPLOS grupos
# Exemplo: por bairro E condicao

# Tentar nivel 1: bairro + condicao
df['price'] = df.groupby(['neighborhood', 'condition'])['price'] \
    .transform(lambda x: x.fillna(x.median()))
# Se ainda tem missing, tentar nivel 2: so bairro
df['price'] = df.groupby('neighborhood')['price'] \
    .transform(lambda x: x.fillna(x.median()))
# Se AINDA tem missing, usar global
df['price'] = df['price'].fillna(df['price'].median())
print(f"Missing restante: {df['price'].isnull().sum()}")
# Deve ser 0!
```

Estratégia 4: Valor Constante

Preencher com valor específico do domínio

Quando usar:

- Valor padrão faz sentido logicamente
- Missing significa ”zero”ou ”nenhum”
- Você conhece bem o domínio

Exemplos:

- garage_spaces = 0 (sem garagem)
- discount = 0 (sem desconto)
- children = 0 (sem filhos)
- response_time = 999 (não respondeu)
- category = ”Unknown”(desconhecido)

Estratégia 4: Valor Constante

⚠ Atenção

CUIDADO: Valor deve fazer sentido! Não preencha idade com 0.

Imputação com Valor Constante

</> Python

```python
# Exemplo 1: Vagas de garagem
# Se nao mencionou, provavelmente nao tem
df['garage_spaces'] = df['garage_spaces'].fillna(0)
# Exemplo 2: Categoria desconhecida
df['neighborhood'] = df['neighborhood'].fillna('Unknown')
# Exemplo 3: Estado medio (mais conservador)
df['condition'] = df['condition'].fillna('Average')
# Exemplo 4: Jardim = nao (se nao mencionou)
df['has_garden'] = df['has_garden'].fillna('No')
# Exemplo 5: Valor especifico do negocio
# Imóveis muito antigos (age missing) = 100 anos
df['age'] = df['age'].fillna(100)
```

Estratégia 5: Forward e Backward Fill

Propagação de valores (útil para séries temporais)

Forward Fill (ffill):

- Propaga último valor válido para frente
- `10, NaN, NaN, 20` $\rightarrow$ `10, 10, 10, 20`
- Assume: valor permanece até mudar

Backward Fill (bfill):

- Propaga próximo valor válido para trás
- `10, NaN, NaN, 20` $\rightarrow$ `10, 20, 20, 20`
- Assume: próximo valor já era válido

Quando usar:

- Dados ordenados (tempo, sequência)
- Valores mudam lentamente
- NÃO usar para dados não ordenados!

Forward e Backward Fill: Código

</> Python

```python
# Forward fill: propaga ultimo valor valido
df_sorted = df.sort_values('age')  # Ordenar primeiro!
df_sorted['price'] = df_sorted['price'].ffill()

# Backward fill: propaga proximo valor valido
df_sorted['price'] = df_sorted['price'].bfill()

# Exemplo visual:
# Original:  10, NaN, NaN, 20, NaN, 30, NaN
# ffill:     10,  10,  10, 20,  20, 30,  30
# bfill:     10,  20,  20, 20,  30, 30, NaN (ultimo fica)

# Combinar: ffill + bfill (preenche tudo)
df['price'] = df['price'].ffill().bfill()
```

Forward e Backward Fill: Código

Nota Importante

Só funciona bem com dados ORDENADOS (séries temporais)

Estratégia 6: Interpolação

Estimar valor entre dois pontos conhecidos

Interpolação Linear:

- Assume mudança linear entre pontos
- `age: 10, NaN, 30` → `age: 10, 20, 30`
- `price: 100, NaN, NaN, 200` → `100, 133, 166, 200`

Quando usar:

- Dados ordenados e contínuos
- Séries temporais
- Gradiente suave esperado

Quando NÃO usar:

- Dados categóricos
- Dados não ordenados
- Mudanças abruptas

Interpolação: Implementação

</> Python

```python
# Ordenar por variavel relevante
df_sorted = df.sort_values('age')

# Interpolacao linear
df_sorted['price'] = df_sorted['price'].interpolate(method='linear')

# Outros metodos:
# - 'polynomial': curva polinomial
# - 'spline': curva suave
# - 'nearest': valor mais proximo
```

Interpolação: Implementação (cont.)

Python

```python
# Exemplo de resultado:
# age:    10,  15,  NaN,  25,  30
# price: 100, 110,  NaN, 130, 140
#
# Apos interpolacao:
# age:    10,  15,  20,  25,  30
# price: 100, 110, 120, 130, 140
```

Estratégia 7: Criar Flag de Imputação

SEMPRE uma boa prática!

Por que criar flag?

- Marca onde valores foram imputados
- Preserva informação sobre missing
- ML pode aprender que missing importa
- Permite análise de impacto

Como funciona:

- Criar coluna binária (0/1)
- 1 = foi imputado
- 0 = valor original
- Nome: `was_missing_[coluna]`

Exemplo:

- price = 300000, price_was_missing = 0 (original)
- price = 250000, price_was_missing = 1 (imputado)

Criar Flag: Implementação

</> Python

```python
# ANTES de imputar, crie a flag!
# Flag indicando onde era missing
df['price_was_missing'] = df['price'].isnull().astype(int)
# Agora impute
df['price'] = df['price'].fillna(df['price'].median())
# Resultado:
# - price: valores completos (alguns imputados)
# - price_was_missing: 1 onde foi imputado, 0 original
# Verificar
print(f"Valores imputados: {df['price_was_missing'].sum()}")
# Analise: preco imputado vs original difere?
print(df.groupby('price_was_missing')['area'].mean())
```

Criar Flags para Múltiplas Colunas

Python

```python
# Criar flags para todas as colunas com missing
cols_with_missing = df.columns[df.isnull().any()].tolist()
for col in cols_with_missing:
    flag_name = f'{col}_was_missing'
    df[flag_name] = df[col].isnull().astype(int)
    print(f"\cmark  Flag criada: {flag_name}")
# Verificar flags criadas
flag_cols = [c for c in df.columns if '_was_missing' in c]
print(f"\nFlags de imputacao: {flag_cols}")
# Ver quantos foram imputados
print("\nTotal de imputacoes por coluna:")
print(df[flag_cols].sum())
```

Escolher Estratégia: Árvore de Decisão

PASSO 1: Tipo de variável?

- Numérica → média ou mediana
- Categórica → moda ou constante

PASSO 2: Há agrupamento natural?

- SIM → imputação por grupo
- NÃO → imputação global

PASSO 3: Dados têm ordem?

- Temporal → interpolação ou ffill
- Não ordenado → média/mediana

PASSO 4: Tipo de missing?

- MCAR → qualquer método
- MAR → por grupos
- MNAR → cuidado! Talvez constante + flag

Caso de Uso Completo: Housing Prices - Estratégia por variável

price (12% missing, MNAR):

- Imputar com mediana por bairro
- Criar flag price_was_missing
- Razão: preços variam muito por bairro

area (8% missing, MCAR):

- Imputar com mediana por bairro
- Razão: áreas também variam por bairro

age (15% missing, MNAR):

- Imputar com valor alto (100 anos) + flag
- Razão: missing pode significar ”muito antigo”

neighborhood (5% missing):

- Imputar com moda (”Centro”)
- Ou categoria ”Unknown”

Pipeline de Imputação Completo

Python

```python
def impute_housing_data(df):
    """Pipeline de imputacao completo"""
    df = df.copy()

    # 1. Criar flags ANTES de imputar
    df['price_was_missing'] = df['price'].isnull().astype(int)
    df['age_was_missing'] = df['age'].isnull().astype(int)

    # 2. Imputar price por bairro
    df['price'] = df.groupby('neighborhood')['price'] \
        .transform(lambda x: x.fillna(x.median()))

    # 3. Imputar area por bairro
    df['area'] = df.groupby('neighborhood')['area'] \
        .transform(lambda x: x.fillna(x.median()))
```

Pipeline de Imputação (cont.)

Python

```python
    # 4. Imputar age com valor alto (antigos)
    df['age'] = df['age'].fillna(100)
    # 5. Imputar bedrooms/bathrooms com mediana
    df['bedrooms'] = df['bedrooms'].fillna(df['bedrooms'].median())
    df['bathrooms'] = df['bathrooms'].fillna(df['bathrooms'].median())
    # 6. Imputar categoricas com moda
    df['neighborhood'] = df['neighborhood'].fillna(df['neighborhood'].mode
    ()[0])
    df['condition'] = df['condition'].fillna('Average')
    # 7. Validacao final
    assert df.isnull().sum().sum() == 0, "Ainda há missing!"
    return df
# Usar pipeline
df_imputed = impute_housing_data(df)
```

Validar Imputação: Comparar Distribuições

Python

```python
import matplotlib.pyplot as plt
# Comparar distribuicao antes/depois
fig, axes = plt.subplots(1, 2, figsize=(14, 5))
# Antes: apenas valores originais
original_values = df_original['price'].dropna()
axes[0].hist(original_values, bins=30, alpha=0.7)
axes[0].set_title('Distribuicao Original (sem missing)')
axes[0].set_xlabel('Price')
# Depois: com valores imputados
axes[1].hist(df_imputed['price'], bins=30, alpha=0.7, color='green')
axes[1].set_title('Distribuicao Apos Imputacao')
axes[1].set_xlabel('Price')
plt.tight_layout()
plt.show()
```

Exercício Completo: Imputação Housing

Exercício Prático

Implemente imputação completa do Housing:

1. Para cada variável com missing:
   - Decida qual estratégia usar
   - Justifique a escolha
2. Implemente o pipeline:

```
# - Criar flags ANTES de imputar
# - Aplicar estrategias escolhidas
# - Validar: nenhum missing deve restar
assert df.isnull().sum().sum() == 0
```

3. Compare distribuições antes/depois
4. Documente todas as decisões
5. Gere relatório de imputação

Resumo do Bloco 2

Remoção:

- dropna() - quando < 5% e MCAR

Imputação Simples:

- Média, mediana, moda
- Rápido mas reduz variabilidade

Imputação Avançada:

- Por grupos (groupby + transform)
- Forward/backward fill (séries)
- Interpolação (dados ordenados)
- Valor constante (conhecimento do domínio)

Boa Prática:

- Sempre criar flags de imputação!

Transição para Próximo Bloco

Já sabemos lidar com missing!

Agora vamos para:

- Limpeza de strings (espaços, maiúsculas)
- Padronização de formatos
- Validação de ranges
- Detecção de valores impossíveis
- Tratamento de duplicatas
- Conversão de tipos
- Consistência lógica
- Pipeline completo de limpeza

Nota Importante

Bloco 3: Deixar os dados impecáveis!

Bloco 3

Limpeza de Inconsistências e Validação

Além de Missing: Outros Problemas

Dados podem estar ”presentes”mas ERRADOS:

1. Problemas de Formato

- Espaços extras: ”Casa ”vs ”Casa”
- Maiúsculas/minúsculas: ”SIM”vs ”sim”vs ”Sim”
- Inconsistências: ”São Paulo”vs ”Sao Paulo”vs ”SP”
- Caracteres especiais: ”R$ 1.000”vs ”1000”

2. Valores Impossíveis

- Idade = -5 ou 200 anos
- Área = 0 m^2 ou 10000 m^2
- Data de nascimento no futuro
- Preço = R$ 1 ou R$ 999.999.999

3. Duplicatas

- Mesmo registro múltiplas vezes
- Inconsistências entre duplicatas

Limpeza de Strings: Por que Importa?

Strings sujas causam problemas:

Exemplo problemático:

”Centro”
”Centro”
”Centro ”
”Centro ”
”centro”
”CENTRO”

Limpeza de Strings: Por que Importa?

Para o computador:

- São 6 categorias diferentes!
- value_counts() mostra 6 valores únicos
- Análises ficam fragmentadas
- Gráficos com categorias duplicadas

Para o humano:

- É tudo a mesma coisa: Centro

Limpeza: Remover Espaços

Python

```python
# Problema: espacos extras
print("ANTES:")
print(df['neighborhood'].unique())
# [' Centro', 'Centro ', ' Centro ', 'Suburbio', ...]

# Solucao: strip() remove espacos das pontas
df['neighborhood'] = df['neighborhood'].str.strip()

print("\nDEPOIS:")
print(df['neighborhood'].unique())
# ['Centro', 'Suburbio', ...]

# Verificar impacto
print(f"\nCategorias antes: {df_original['neighborhood'].nunique()}")
print(f"Categorias depois: {df['neighborhood'].nunique()}")
```

Limpeza: Padronizar Capitalização

Python

```python
# Problema: capitalizacao inconsistente
print("ANTES:")
print(df['condition'].value_counts())
# Excelente    50
# EXCELENTE    30
# excelente    20
# Bom          40
# BOM          15

# Solucao 1: Tudo minusculo (RECOMENDADO)
df['condition'] = df['condition'].str.lower()

# Solucao 2: Tudo maiusculo
# df['condition'] = df['condition'].str.upper()
```

Limpeza: Padronizar Capitalização (cont.)

Python

```python
# Solucao 3: Primeira letra maiuscula
# df['condition'] = df['condition'].str.title()

print("\nDEPOIS:")
print(df['condition'].value_counts())
# excelente   100
# bom          55
```

Qual Capitalização Usar?

Guia de decisão:

Use .lower() (minúsculo):

- ✓ RECOMENDADO para análise
- ✓ Mais fácil de digitar e buscar
- ✓ Menos propenso a erros
- Exemplo: ”centro”, ”excelente”, ”bom”

Use .upper() (maiúsculo):

- Códigos e siglas (ISO, UF)
- Exemplo: ”SP”, ”RJ”, ”USD”, ”BRL”

Use .title() (capitalizado):

- Para apresentação final
- Nomes próprios
- Exemplo: ”São Paulo”, ”Excelente”

Limpeza: Substituir Variações

Python

```python
# Problema: multiplas formas de escrever o mesmo
print("ANTES:")
print(df['neighborhood'].unique())
# ['Sao Paulo', 'São Paulo', 'Sao Paulo (SP)',
#  'S. Paulo', 'SP']

# Solucao: Substituir variacoes
df['neighborhood'] = df['neighborhood'] \
    .str.replace('Sao Paulo', 'São Paulo') \
    .str.replace('S. Paulo', 'São Paulo') \
    .str.replace(' (SP)', '') \
    .str.replace('SP', 'São Paulo') \
    .str.strip()

print("\nDEPOIS:")
print(df['neighborhood'].unique())
```

Limpeza: Remover Caracteres Especiais

Python

```python
# Problema: caracteres indesejados
print("ANTES:")
print(df['price'].head())
# 'R$ 300.000', 'R$ 250,000', '150000', '$200k'

# Se price esta como string, converter!
# Remover caracteres nao numericos
df['price'] = df['price'].str.replace('R$', '') \
                         .str.replace('$', '') \
                         .str.replace('.', '') \
                         .str.replace(',', '.') \
                         .str.replace('k', '000') \
                         .str.strip()
```

Limpeza: Remover Caracteres Especiais (cont.)

Python

```python
# Converter para numero
df['price'] = pd.to_numeric(df['price'], errors='coerce')

print("\nDEPOIS:")
print(df['price'].head())
# 300000, 250000, 150000, 200000
```

Padronização com Mapeamento

Python

```python
# Problema: categorias nao padronizadas
print("ANTES:")
print(df['condition'].unique())
# ['Excelente', 'Muito Bom', 'Bom', 'Regular',
#  'Ruim', 'Pessimo', 'Ok']
# Solucao: Mapear para valores padrao
condition_map = {
    'excelente': 'excellent',
    'muito bom': 'good',
    'bom': 'good',
    'ok': 'average',
    'regular': 'average',
    'ruim': 'poor',
    'pessimo': 'poor'
}
```

Padronização com Mapeamento (cont.)

Python

```python
df['condition'] = df['condition'].str.lower().map(condition_map)

print("\nDEPOIS:")
print(df['condition'].unique())
# ['excellent', 'good', 'average', 'poor']
```

Aplicar Limpeza em Todas as Categóricas

</> Python

```python
# Pipeline de limpeza para todas as categoricas
categorical_cols = df.select_dtypes(include=['object']).columns
for col in categorical_cols:
    print(f"Limpando {col}...")
    # 1. Remover espacos
    df[col] = df[col].str.strip()
    # 2. Minusculo
    df[col] = df[col].str.lower()
    # 3. Remover espacos duplos internos
    df[col] = df[col].str.replace('  ', ' ')
    print(f"{col}: {df[col].nunique()} categorias unicas")
print("\nLimpeza concluida!")
```

Validação de Dados

1. Range (Intervalo):

- Valor está dentro do esperado?
- Ex: idade entre 0-150, área entre 20-1000

2. Tipo:

- Variável tem o tipo correto?
- Ex: price é numérico, neighborhood é texto

3. Consistência Lógica:

- Relações entre variáveis fazem sentido?
- Ex: bathrooms ≤ bedrooms + 2

4. Formato:

- Dados seguem padrão esperado?
- Ex: data em formato DD/MM/YYYY, CEP com 8 dígitos

Definir Regras de Validação

Para Housing Prices - regras de negócio:

Ranges aceitáveis:

- **price:** R$ 50.000 a R$ 10.000.000
- **area:** 20 m² a 1.000 m²
- **bedrooms:** 0 a 10
- **bathrooms:** 0 a 10
- **age:** 0 a 150 anos
- **garage_spaces:** 0 a 10

Como definir regras?

- Consulte especialistas do domínio
- Analise distribuição (percentis 1% e 99%)
- Use senso comum e conhecimento do mundo real
- Seja conservador (não muito restrito)
- Documente as regras e suas fontes

Validação: Identificar Valores Impossíveis

Python

```python
# Converter todas as colunas de interesse para numérico
cols = ['age', 'price', 'area', 'bedrooms', 'bathrooms']
df[cols] = df[cols].apply(pd.to_numeric, errors='coerce')

# Verificar valores impossíveis
print("=== VALORES NEGATIVOS ===")
for col in cols:
    negative = df[df[col] < 0]
    if len(negative) > 0:
        print(f"{col}: {len(negative)} valores negativos")
        print(negative[[col]].head())
```

Validação: Identificar Valores Impossíveis (cont.)

Python

```python
print("\n=== VALORES EXTREMOS ===")
print(f"Bedrooms > 20: {len(df[df['bedrooms'] > 20])}")
print(f"Age > 200: {len(df[df['age'] > 200])}")
print(f"Area > 10000: {len(df[df['area'] > 10000])}")

print("\n=== ZEROS SUSPEITOS ===")
print(f"Area = 0: {len(df[df['area'] == 0])}")
print(f"Price = 0: {len(df[df['price'] == 0])}")
```

Validação: Aplicar Regras

Python

```python
# Definir e aplicar regras de validacao

rules = {
    'price': (50_000, 10_000_000),
    'area': (20, 1000),
    'bedrooms': (0, 10),
    'bathrooms': (0, 10),
    'age': (0, 150),
    'garage_spaces': (0, 10)
}
```

Validação: Aplicar Regras (cont.)

Python

```python
print("=== VALIDACAO DE RANGES ===")
for col, (min_val, max_val) in rules.items():
    invalid = df[(df[col] < min_val) | (df[col] > max_val)]

    if len(invalid) > 0:
        print(f"\n{col}: {len(invalid)} valores invalidos")
        print(f"  Range esperado: [{min_val}, {max_val}]")
        print(f"  Valores encontrados:")
        print(f"    Min: {invalid[col].min()}")
        print(f"    Max: {invalid[col].max()}")
```

O que Fazer com Valores Inválidos?

1. Corrigir manualmente (MELHOR):

- Se erro óbvio: 3000 $\rightarrow$ 300 m²
- Se tem acesso à fonte original
- Se poucos casos (< 10)

2. Tratar como missing (RECOMENDADO):

- Substituir por NaN
- Aplicar estratégias de imputação
- Mais conservador e seguro

O que Fazer com Valores Inválidos?

3. Clipar nos limites:

- age = -5 $\rightarrow$ 0
- area = 10000 $\rightarrow$ 1000
- Menos recomendado (distorce)

4. Remover registro:

- Apenas se muitos valores inválidos na linha
- Última opção

Tratar Inválidos como Missing

Python

```python
# Substituir valores fora do range por NaN
# Depois aplicar imputacao

# Exemplo: bedrooms
print(f"ANTES: {df['bedrooms'].isnull().sum()} missing")

# Valores invalidos virao NaN
df.loc[df['bedrooms'] < 0, 'bedrooms'] = np.nan
df.loc[df['bedrooms'] > 10, 'bedrooms'] = np.nan
```

Tratar Inválidos como Missing

Python

```python
print(f"DEPOIS: {df['bedrooms'].isnull().sum()} missing")

# Ou usando between() (mais elegante)
df.loc[~df['bedrooms'].between(0, 10), 'bedrooms'] = np.nan

# Agora imputar
df['bedrooms'] = df['bedrooms'].fillna(df['bedrooms'].median())

print(f"FINAL: {df['bedrooms'].isnull().sum()} missing")
```

Identificar Duplicatas

Python

```python
# Verificar duplicatas completas (todas as colunas iguais)
print("=== DUPLICATAS COMPLETAS ===")
n_duplicates = df.duplicated().sum()
print(f"Total de duplicatas: {n_duplicates}")

# Ver as duplicatas
if n_duplicates > 0:
    duplicates = df[df.duplicated(keep=False)]
    print("\nExemplos de duplicatas:")
    print(duplicates.sort_values(by=df.columns[0]).head(10))
```

Identificar Duplicatas (cont.)

Python

```python
# Duplicatas em colunas especificas
# (ex: mesmo endereco)
print("\n=== DUPLICATAS PARCIAIS ===")
dup_address = df.duplicated(
    subset=['neighborhood', 'area', 'bedrooms'],
    keep=False
)
print(f"Duplicatas de endereco: {dup_address.sum()}")
```

Remover Duplicatas

Python

```python
print(f"Linhas ANTES: {len(df)}")

# Remover duplicatas completas
df_clean = df.drop_duplicates()
print(f"Linhas DEPOIS (completas): {len(df_clean)}")
print(f"Removidos: {len(df) - len(df_clean)}")

# Manter primeira ocorrencia (padrao)
df_clean = df.drop_duplicates(keep='first')

# Manter ultima ocorrencia
# df_clean = df.drop_duplicates(keep='last')
```

Remover Duplicatas (cont.)

Python

```python
# Remover TODAS (incluindo originais)
# df_clean = df.drop_duplicates(keep=False)

# Duplicatas em colunas especificas
df_clean = df.drop_duplicates(
    subset=['neighborhood', 'area', 'bedrooms']
)
print(f"Linhas finais: {len(df_clean)}")
```

Conversão de Tipos

Python

```python
# Verificar tipos atuais
print("=== TIPOS ATUAIS ===")
print(df.dtypes)

# Converter para tipos corretos
print("\n=== CONVERTENDO ===")

# Numericas inteiras
df['bedrooms'] = df['bedrooms'].astype('int')
df['bathrooms'] = df['bathrooms'].astype('int')
df['age'] = df['age'].astype('int')

# Numericas decimais
df['price'] = df['price'].astype('float')
df['area'] = df['area'].astype('float')
```

Conversão de Tipos (cont.)

Python

```python
# Categoricas (economiza memoria!)
df['neighborhood'] = df['neighborhood'].astype('category')
df['condition'] = df['condition'].astype('category')

print("\n=== TIPOS FINAIS ===")
print(df.dtypes)

# CUIDADO: NaN vai gerar erro
```

Conversão Segura com to_numeric

Python

```python
# Se coluna tem valores nao numericos misturados

# Opcao 1: Converter, invalidos viram NaN
df['area'] = pd.to_numeric(df['area'], errors='coerce')
# "100"  $\rightarrow$  100
# "abc"  $\rightarrow$  NaN
# "50.5"  $\rightarrow$  50.5

# Opcao 2: Ignorar erros (mantem original)
df['area'] = pd.to_numeric(df['area'], errors='ignore')
```

Conversão Segura com to_numeric (cont.)

Python

```python
# Opcao 3: Levantar erro (aborta se tiver invalido)
try:
    df['area'] = pd.to_numeric(df['area'], errors='raise')
except ValueError as e:
    print(f"Erro: {e}")
    print("Ha valores nao numericos!")

# RECOMENDADO: usar 'coerce' e depois imputar NaNs
```

Tratamento de Datas

</> Python

```python
# Converter strings para datetime
# Formato padrao (YYYY-MM-DD)
df['sale_date'] = pd.to_datetime(df['sale_date'])
# Formato brasileiro (DD/MM/YYYY)
df['sale_date'] = pd.to_datetime(df['sale_date'],
                                 format='%d/%m/%Y')
# Erros viram NaT (Not a Time)
df['sale_date'] = pd.to_datetime(df['sale_date'],
                                 errors='coerce')
# Extrair componentes
df['sale_year'] = df['sale_date'].dt.year
df['sale_month'] = df['sale_date'].dt.month
df['sale_day'] = df['sale_date'].dt.day
df['sale_weekday'] = df['sale_date'].dt.dayofweek
print(df[['sale_date', 'sale_year', 'sale_month']].head())
```

Validação de Consistência Lógica

Python

```python
# Verificar relacoes logicas entre variaveis

print("=== VALIDACAO DE LOGICA ===")

# Regra 1: Banheiros nao pode ser >> quartos
invalid_bath = df[df['bathrooms'] > df['bedrooms'] + 2]
print(f"Banheiros > Quartos+2: {len(invalid_bath)}")
if len(invalid_bath) > 0:
    print(invalid_bath[['bedrooms', 'bathrooms']].head())
```

Validação de Consistência Lógica (cont.)

Python

```python
# Regra 2: Preco/m² deve estar em range razoavel
df['price_per_sqm'] = df['price'] / df['area']
median_price_sqm = df['price_per_sqm'].median()

outliers = df[
    (df['price_per_sqm'] < median_price_sqm * 0.1) |
    (df['price_per_sqm'] > median_price_sqm * 10)
]
print(f"\nPreco/m² anormal: {len(outliers)}")
```

Validação: Criar Função Reutilizável

Python

```python
def validate_housing_data(df):
    """Valida regras de negocio"""
    issues = []
    # Verificar negativos
    for col in ['price', 'area', 'age']:
        n_negative = (df[col] < 0).sum()
        if n_negative > 0:
            issues.append(f"{col}: {n_negative} negativos")
    # Verificar ranges
    if (df['bedrooms'] > 20).any():
        issues.append("Bedrooms > 20 encontrados")
    if (df['area'] > 10000).any():
        issues.append("Area > 10000 encontrada")
```

Validação: Criar Função Reutilizável (cont.)

Python

```python
    # ... outras validacoes

    return issues

# Usar
problems = validate_housing_data(df)
if problems:
    print("PROBLEMAS ENCONTRADOS:")
    for p in problems:
        print(f"  - {p}")
```

Exercício: Limpeza e Validação Completa

Exercício Prático

Limpe e valide completamente o Housing dataset:

1. Limpeza de strings: strip(), lower(), replace()
2. Validação de ranges:
   - Defina regras para cada variável
   - Identifique valores inválidos
   - Trate como missing e impute
3. Duplicatas:
   - Identifique e remova
4. Conversão de tipos corretos
5. Validação de consistência lógica
6. Documente TUDO!

Resumo do Bloco 3

Técnicas de limpeza aprendidas:

Strings:

- strip(), lower(), upper(), replace()
- Padronização de categóricas
- Mapeamento de valores

Validação:

- Definir regras de negócio
- Identificar valores impossíveis
- Tratar inválidos (missing ou corrigir)

Outros:

- Remover duplicatas
- Converter tipos corretamente
- Tratar datas
- Validar consistência lógica

Bloco 4

Pipeline Completo e Documentação

Pipeline Completo: Estrutura

Python

```python
def clean_housing_pipeline(df):
    """
    Pipeline completo de limpeza:
    1. Copiar dados originais
    2. Limpar strings
    3. Validar e tratar invalidos
    4. Criar flags de missing
    5. Imputar valores faltantes
    6. Remover duplicatas
    7. Converter tipos
    8. Validar resultado
    """
    df_clean = df.copy()
    # ... implementacao
    return df_clean
```

Pipeline: Implementação (cont.)

Python

```python
def clean_housing_pipeline(df):
    df_clean = df.copy()
    print("Pipeline de limpeza iniciado...")

    # PASSO 1: Limpar strings
    print("\n[1/7] Limpando strings...")
    cat_cols = df_clean.select_dtypes(include=['object']).columns
    for col in cat_cols:
        df_clean[col] = df_clean[col].str.strip().str.lower()
```

Pipeline: Implementação (cont.)

Python

```python
    # PASSO 2: Validar ranges e marcar invalidos
    print("[2/7] Validando ranges...")
    df_clean.loc[~df_clean['bedrooms'].between(0, 10), 'bedrooms'] = np.
    nan
    df_clean.loc[~df_clean['area'].between(20, 1000), 'area'] = np.nan
    df_clean.loc[~df_clean['age'].between(0, 150), 'age'] = np.nan
```

Pipeline: Implementação (cont.)

Python

```python
    # PASSO 3: Criar flags de imputacao
    print("[3/7] Criando flags...")
    df_clean['price_was_missing'] = df_clean['price'].isnull().astype(int)
    df_clean['age_was_missing'] = df_clean['age'].isnull().astype(int)

    # PASSO 4: Imputar valores faltantes
    print("[4/7] Imputando valores...")
    # Por grupo
    df_clean['price'] = df_clean.groupby('neighborhood')['price'] \
        .transform(lambda x: x.fillna(x.median()))
    df_clean['area'] = df_clean.groupby('neighborhood')['area'] \
        .transform(lambda x: x.fillna(x.median()))
```

Pipeline: Implementação (cont.)

Python

```python
    # Global
    df_clean['age'] = df_clean['age'].fillna(100)
    df_clean['bedrooms'] = df_clean['bedrooms'].fillna(df_clean['bedrooms']
    ].median())

    # PASSO 5: Remover duplicatas
    print("[5/7] Removendo duplicatas...")
    n_before = len(df_clean)
    df_clean = df_clean.drop_duplicates()
    n_removed = n_before - len(df_clean)
    print(f"  Removidos: {n_removed} duplicatas")
```

Pipeline: Implementação (cont.)

Python

```python
    # PASSO 6: Converter tipos - Cuidado com conversão numérica
    print("[6/7] Convertendo tipos...")
    df_clean['bedrooms'] = df_clean['bedrooms'].astype('int')
    df_clean['bathrooms'] = df_clean['bathrooms'].astype('int')
    df_clean['neighborhood'] = df_clean['neighborhood'].astype('category')
    df_clean['condition'] = df_clean['condition'].astype('category')
```

Pipeline: Implementação (cont.)

Python

```python
    # PASSO 7: Validacao final
    print("[7/7] Validando resultado...")
    assert df_clean.isnull().sum().sum() == 0, "Ainda ha missing!"

    print("\n\cmark  Pipeline concluido com sucesso!")
    return df_clean

```

Gerar Relatório de Limpeza

Python

```python
def generate_cleaning_report(df_original, df_clean):
    """Relatorio completo de limpeza"""
    report = {
        'Linhas originais': len(df_original),
        'Linhas finais': len(df_clean),
        'Linhas removidas': len(df_original) - len(df_clean),
        'Missing antes': df_original.isnull().sum().sum(),
        'Missing depois': df_clean.isnull().sum().sum(),
        'Duplicatas removidas': len(df_original) - len(df_original.drop_duplicates()),
        'Memoria antes (MB)': df_original.memory_usage(deep=True).sum() / 1024**2,
        'Memoria depois (MB)': df_clean.memory_usage(deep=True).sum() / 1024**2
    }
```

Gerar Relatório de Limpeza (cont.)

Python

```python
    print("="*50)
    print("RELATÓRIO DE LIMPEZA")
    print("="*50)
    for key, value in report.items():
        print(f"{key:25s}: {value:,.2f}")

    return report

```

Checklist Final de Limpeza

Antes de considerar dados "limpos":

1. ☐ Valores faltantes identificados e tratados
2. ☐ Flags de imputação criadas
3. ☐ Strings padronizadas (strip, lower)
4. ☐ Tipos de dados corretos
5. ☐ Valores impossíveis removidos/corrigidos
6. ☐ Ranges validados
7. ☐ Duplicatas removidas
8. ☐ Consistência lógica verificada
9. ☐ Todas as decisões documentadas
10. ☐ Relatório de limpeza gerado
11. ☐ Pipeline reproduzível criado
12. ☐ Testes de validação passando

Checklist Final de Limpeza

⚠ Atenção

Só avance se TODOS os itens estiverem ✓

Demonstração Prática

Live Coding

Vamos praticar juntos:

1. (atualizado em aula)

Recap: Aula 09 Completa

Jornada de hoje:

Bloco 1: Identificação de Missing

- Tipos: MCAR, MAR, MNAR
- Visualização e análise de padrões
- Decisão: remover vs imputar

Bloco 2: Estratégias de Imputação

- Simples: média, mediana, moda
- Avançada: por grupos, interpolação
- Flags de imputação

Recap: Aula 09 Completa

Jornada de hoje:

Bloco 3: Limpeza e Validação

- Padronização de strings
- Validação de ranges
- Remoção de duplicatas
- Conversão de tipos

Bloco 4: Pipeline

- Workflow completo e reproduzível

Erros Comuns - Evite!

1. **Não fazer cópia dos dados**
   - ✗ df_clean = df (referência!)
   - ✓ df_clean = df.copy()
2. **Imputar sem análise**
   - ✗ fillna(0) em tudo
   - ✓ Entender tipo de missing primeiro
3. **Não documentar decisões**
   - ✗ Fazer mudanças silenciosamente
   - ✓ Comentar e documentar cada passo
4. **Ser muito agressivo**
   - ✗ Remover 50% dos dados
   - ✓ Ser conservador
5. **Não validar resultado**
   - ✗ Assumir que funcionou
   - ✓ Sempre verificar com asserts

Preview: Aula 10

Quinta-feira (06/11) - Transformação e Normalização:

Feature Engineering:

- Criar novas variáveis úteis
- Extrair informação de datas
- Interações entre variáveis

Encoding de Categóricas:

- Label Encoding
- One-Hot Encoding
- Quando usar cada um

Normalização e Escalonamento:

- Min-Max Scaling
- Z-score Standardization
- Robust Scaling

A Seguir: Aula Prática

Exercício Prático

Tempo: 60 minutos
Entrega: via Moodle (notebook)

Tarefas:

1. (atualizado durante a aula)

Notebook: Disponível no Moodle

Obrigado!