Pré-Processamento de Dados com Python

Anna Beatriz S. Lima[1], Ian Salomão S. Carneiro[1], Valéria S. Santos[1]

[1]Sistemas de Informação – Centro Universitário de Excelência (UNEX)
44.085-370 – Feira de Santana– BA – Brazil

ab.annabeatrizlima@gmail.com, iansalomao.ca@gmail.com,
valeriasoressantos2@gmail.com

Abstract. *This article presents a technical-scientific report describing the implementation of preprocessing techniques applied to data mining. It highlights the importance of this stage in the analytical process, its implementation in Python, and the main results and challenges observed during the development.*

Resumo. *Este artigo apresenta um relatório técnico-científico que descreve a implementação de técnicas de pré-processamento aplicadas à mineração de dados. São abordadas a importância dessa etapa no processo analítico, sua aplicação em Python e os principais resultados e desafios observados durante o desenvolvimento.*

1. Introdução

A mineração de dados é uma das atividades mais importantes na tomada de decisões de empresas no cenário contemporâneo. Segundo Crabtree e Nehme(2024), é através dela que insights essenciais são extraídos, transformando grandes volumes de dados em informação útil para apoiar decisões organizacionais. Entretanto, para que a qualidade final das informações extraídas da mineração seja satisfatória é necessário mais do que o estudo de dados brutos. Isso ocorre porque estes geralmente possuem inconsistências, erros e informações irrelevantes. Tais problemas podem levar a distorções na análise ou insights falsos.

Assim, o pré-processamento de dados surge como um aspecto fundamental no processo de extração de informação útil, pois prepara, organiza e estrutura os dados brutos para análise, resolvendo problemas estruturais em um conjunto de dados, também conhecido como *dataset*, o que inclui o preenchimento de valores faltantes, remoção ou ajuste de *outliers*, entre outras, garantindo que os algoritmos recebam dados limpos e consistentes nas etapas seguintes da mineração.

Este relatório, portanto, apresenta a implementação de técnicas de pré-processamento em Python, sem recorrer a bibliotecas externas. Tais técnicas foram aplicadas a um projeto simulado de uma empresa de delivery fictícia, a *FoodDelivery*, evidenciando o impacto da qualidade de dados e a confiabilidade das análises.

2. Fundamentação Teórica

O pré-processamento de dados é uma etapa essencial na ciência de dados, pois assegura que as informações estejam completas, consistentes e bem estruturadas antes de serem aplicadas em análises e modelos preditivos, evitando distorções e decisões equivocadas. Um exemplo disso é sua importância em sistemas de recomendação, que poderiam gerar resultados incorretos sem tratamento adequado. Nesta seção, serão abordadas as principais técnicas utilizadas, como tratamento de valores ausentes, normalização e padronização de variáveis, codificação de dados categóricos e organização da arquitetura, ressaltando sua relevância para a qualidade e eficiência do processo analítico.

2.1 Tratamento de Valores Ausentes

Dados faltantes (NaN, *Not a Number*) é um dos problemas mais comuns enfrentados no pré-processamento e podem ocorrer por diversos motivos, classificados em três tipos:

- MCAR (*Missing Completely at Random*): A probabilidade de ausência é aleatória, sem relação com outras variáveis ou com o valor faltante. Exemplo: respostas perdidas em um formulário digital por uma queda do servidor.
- MAR *(Missing at Random):* A ausência tem relação com outras variáveis observáveis, mas independe do valor faltante em si. Exemplo: clientes mais jovens tendem a não responder perguntas sobre atendimento em pesquisas de satisfação.
- MNAR *(Missing Not at Random)*: A ausência se relaciona diretamente com o valor faltante. Exemplo: clientes com histórico de inadimplência não informam dívidas em formulários de empréstimos.

Assim, tratar valores ausentes é essencial para garantir dados consistentes e minimizar vieses. Nesse contexto, destacam-se duas técnicas cruciais no pré-processamento: remoção de dados e imputação estatística.

Na remoção de dados, as linhas que possuem valores ausentes são excluídas, deletando também todos os dados não faltantes que estão nelas. É uma técnica de aplicação simples mas que pode levar a perda de dados valiosos ou diminuição expressiva do *dataset*. Dessa maneira, é de extrema importância que essa técnica seja usada com cautela, sendo aconselhada apenas para *datasets* com menos de 5% de dados faltantes do tipo MCAR. Por exemplo, em uma pesquisa hipotética, apenas 2% dos participantes deixaram respostas aleatórias em branco. Nesse caso, a exclusão dessas linhas não distorce o resultado, tornando a remoção de dados uma estratégia eficiente. A Figura 1 apresenta o pseudocódigo da técnica.

```
CLASSE TratamentoDeValoresAusentes:
    MÉTODO remover_linhas_ausentes(dataset):
        PARA CADA linha EM dataset:
            SE algum valor na linha ESTIVER AUSENTE ENTÃO:
                REMOVER linha
```

Figura 1: Pseudocódigo de Remoção de Valores Ausentes

Já na imputação de dados, os valores ausentes são substituídos por estimativas calculadas a partir de operações estatísticas, geralmente de tendência central, realizadas

com os dados não faltantes. Para que este método seja eficiente, é crucial escolher a métrica de substituição adequada, a qual depende do tipo de distribuição e da presença de valores extremos:

- Se a distribuição de valores for simétrica, ou seja, dados se distribuem igualmente em torno da média, a média será a melhor estimativa para preencher os valores ausentes.

  Exemplo: em um dataset de idades de funcionários com valores ausentes, a média pode preencher os dados faltantes em uma distribuição simétrica, mantendo a consistência da análise.

- Se a distribuição for assimétrica, ou seja, os dados se concentram mais em um lado, gerando valores extremos (*outliers*), a mediana ou moda seria mais adequada, pois evitaria maiores distorções por conta dos *outliers*.

  Exemplo: Em um dataset de preços de imóveis com valores ausentes e alguns outliers (como valores de imóveis de luxo), a mediana preserva melhor a representação realista do mercado. A Figura 2 mostra o pseudocódigo que faz esse tratamento.

```
CLASSE TratamentoDeValoresAusentes
  MÉTODO imputar_valores_ausentes(dataset, metodo)
    PARA CADA coluna EM dataset:
      valor_substituto = NULO
      SE metodo É IGUAL A 'media' ENTÃO
        valor_substituto = CALCULAR_MEDIA(coluna)
      SENÃO SE metodo É IGUAL A 'mediana' ENTÃO
        valor_substituto = CALCULAR_MEDIANA(coluna)
      SENÃO SE metodo É IGUAL A 'moda' ENTÃO
        valor_substituto = CALCULAR_MODA(coluna)

      PARA CADA celula EM coluna:
        SE celula ESTIVER AUSENTE ENTÃO
          SUBSTITUIR celula POR valor_substituto
```

Figura 2: Pseudocódigo de Imputação de Valores Ausentes.

Portanto, o tratamento de dados faltantes é uma etapa fundamental no pré-processamento, sendo especialmente importante em modelos de classificação, pois valores ausentes podem distorcer previsões do algoritmo. Assim, a aplicação cuidadosa das estratégias de imputação e remoção permitem a preservação da integridade do conjunto de dados, assegurando que os resultados obtidos reflitam mais fielmente a realidade dos dados observados.

2.2 Normalização e Padronização de Dados Numéricos

No campo da ciência de dados e do aprendizado de máquina, é comum lidar com variáveis numéricas em diferentes escalas. Essa discrepância pode comprometer o desempenho dos modelos, tornando necessária a aplicação de técnicas de pré-processamento que ajustem os dados para uma escala mais adequada.

- Normalização (*Min-Max Scaling*): **:** ajusta os valores de uma variável para um intervalo definido [0 e 1], mantendo a proporcionalidade entre eles. Essa técnica evita que atributos com valores maiores dominem os menores, permitindo que o modelo aprenda de forma proporcional. É especialmente útil quando se conhece o intervalo esperado dos dados ou quando os algoritmos exigem limites nos valores, garantindo consistência entre treinamento e predição. A Figura 3 mostra o pseudocódigo que implementa essa técnica.

```
função normalizar(valor, valor_min, valor_max):
   retorno (valor - valor_min) / (valor_max - valor_min)
```

Figura 3: Pseudocódigo de Normalização (Min-Max Scaling)

- Padronização (*Standard Scaling ou Z-score*): transforma os valores de forma que cada variável tenha média 0 e desvio padrão 1, uniformizando a distribuição. É recomendada quando os atributos possuem escalas e dispersões diferentes, garantindo que todos os recursos contribuam de forma equivalente para o modelo. O pseudocódigo correspondente pode ser visto na Figura 4.

```
função padronizar(valor, media, desvio_padrao):
   retorno (valor - media) / desvio_padrao
```

Figura 4: Pseudocódigo de Padronização (Standard Scaling)

Em um *dataset* de delivery de comida ,como a *FoodDelivery*, a normalização evita que o número de pedidos (milhões) sobreponha a avaliação média (1 a 5). A padronização, por sua vez, ajusta valores para média 0 e desvio padrão 1, equilibrando escalas e melhorando a precisão.

A normalização é mais utilizada em algoritmos baseados em distância, como o *k-Nearest Neighbors* (k-NN), pois os recursos precisam estar na mesma escala para que os cálculos de distância sejam precisos. Já a padronização é fundamental para algoritmos baseados em gradiente, como o *Support Vector Machines* (SVM), e é frequentemente aplicada em técnicas de redução de dimensionalidade, como o PCA, onde é importante preservar corretamente a variação dos atributos.

2.3 Codificação de Variáveis Categóricas

A maioria dos algoritmos de aprendizado de máquina, como redes neurais e modelos lineares, opera fundamentalmente com dados numéricos. Assim, *datasets* com variáveis categóricas textuais, como o tipo de culinária (Italiana, Japonesa, Árabe) ou o dia da semana (Segunda, Terça), precisam convertê-las em um formato numérico. Esse processo é conhecido como Codificação de Variáveis Categóricas, passo indispensável no pré-processamento de dados. Por exemplo, em um sistema de recomendação de restaurantes, é essencial que o modelo entenda a preferência do usuário por um tipo de cozinha, e o encoding é o que torna essa informação processável. Para isso, usam-se duas

técnicas:

- *Label Encoding:* técnica simples na qual cada categoria única recebe um valor numérico inteiro. Essa técnica é mais adequada para variáveis ordinais, que possuem uma ordem ou hierarquia inerente (ex: Pequeno -> 0, Médio -> 1, Grande -> 2), pois a sequência numérica reflete a ordem real. Por exemplo, se a coluna 'dia_semana' tiver categorias 'Segunda', 'Terça' e 'Quarta', o *encoder* pode mapeá-las para 0, 1 e 2, respectivamente. No entanto, está técnica é contraindicada para variáveis nominais, pois o modelo poderia interpretar o valor 2 (categoria 1) como "duas vezes maior" que 1 (categoria 2), introduzindo um viés e prejudicando o aprendizado.

- *One-Hot Encoding:* técnica que transforma cada categoria original em uma nova coluna binária separada, mitigando o risco de criar uma relação ordinal artificial. Por exemplo, se uma coluna 'tipo_culinaria' tiver três categorias, ela será removida e substituída por três novas colunas: 'culinaria_italiana', 'culinaria_japonesa' e 'culinaria_arabe'. Para cada linha, apenas a coluna correspondente à categoria original receberá o valor 1, e as demais receberão 0. Isso garante que todas as categorias sejam tratadas como iguais em peso, sem nenhuma implicação ordinal. Contudo, a técnica pode ser ineficiente em *datasets* com variáveis de alta cardinalidade (muitas categorias únicas, como o CEP), pois gera muitas novas colunas, resultando em um conjunto esparso (muitos zeros) e exigindo maior poder computacional.

A Figura 5 demonstra com um pseudocódigo como seriam os resultados das funções de *Label Encoding* e *One-Hot Encoding* em um dataset de exemplo que contém dados nominais não ordinais.

```
INPUT_DADOS = {
    'culinaria': ['Italiana', 'Japonesa', 'Italiana', 'Árabe'],
    'avaliacao': [4.5, 4.8, 4.2, 4.0]
}

RESULTADO_LABEL = {
    'culinaria': [1, 2, 1, 0],
    'avaliacao': [4.5, 4.8, 4.2, 4.0]
}

RESULTADO_ONEHOT = {
    'culinaria_Italiana': [1, 0, 1, 0],
    'culinaria_Japonesa': [0, 1, 0, 0],
    'culinaria_Arabe':    [0, 0, 0, 1],
    'avaliacao': [4.5, 4.8, 4.2, 4.0]
}
```

Figura 5: Pseudocódigo de utilização das classes de Codificação de Variáveis Categóricas

2.4 Arquitetura de Pré-Processamento de Dados

Uma arquitetura de pré-processamento de dados robusta e centralizada simplifica a preparação de dados e garante consistência. Para isso, a classe *Preprocessing* atua como

fachada, permitindo encadear operações como tratamento de nulos, codificação e ajuste de escalas em um único fluxo, facilitando manutenção e reutilização em projetos de maior porte. Imagine, por exemplo, que você está construindo um modelo de recomendação que usa o *dia da semana (Segunda, Terça...)* e o valor da compra. O método *encode* traduz o dia da semana para números, enquanto o *scale* padroniza os valores de compra para evitar que eles dominem o modelo, e *fillna* ou *dropna* lidam com qualquer dado faltante nessas colunas. Ao invés de escrever 10 linhas de código separadas, a classe permite encadear essas operações de forma mais limpa, como demonstrado no pseudocódigo da figura 6.

```
DATASET = {
    'dia_semana': ['Segunda', 'Quarta', 'Terça', 'Segunda', 'Quinta', ...],
    'valor_compra': [50.00, 120.50, NULO, 35.00, 200.00, ...],
    'forma_pagamento': ['Credito', 'Debito', 'Credito', 'Dinheiro', 'Pix', ...]
}

PROCESSADOR = nova Preprocessing(DATASET)

RESULTADO = PROCESSADOR
    .fillna()
    .scale(colunas=['valor_compra'], metodo='minMax')
    .encode(colunas=['dia_semana'], metodo='label')
    .encode(colunas=['forma_pagamento'], metodo='oneHot')
```

Figura 6: Pseudocódigo de utilização da classe Preprocessing

Essa abordagem centralizada é essencial em projetos de grande escala, pois garante que todos os modelos sejam treinados sob as mesmas transformações.

3. Metodologia

O projeto adota uma arquitetura modular com o padrão *Facade*, por meio da classe Preprocessing, que centraliza e coordena componentes para diferentes etapas do pré-processamento de dados, como tratamento de valores ausentes, remoção de duplicados, normalização de escalas e codificação de variáveis categóricas. Os *datasets* são representados como dicionários Python, permitindo manipulação direta dos dados sem depender de bibliotecas externas.

3.1 Métodos Aplicados e Estrutura da Classe

A construção da biblioteca foi organizada em etapas, com classes especializadas para cada função:

1. Classe *Statistics*: calcula métricas básicas (média, mediana, moda, variância, desvio padrão, etc.), fornecendo suporte analítico essencial para a etapa de pré-processamento.
2. Classe *Preprocessing:* atua como fachada, orquestrando o uso de classes especializadas para diferentes etapas do pré-processamento de dados, que são compostas por:
    a. "*MissingValueProcessor":* Tratamento valores ausentes com métodos como "*isna()"* e "*notna()"* para identificação de valores nulos ou não nulos, "*fillna()"* para o preenchimento de lacunas por média, mediana,

moda ou valores definidos, e *"dropna()"* para a remoção de registros incompletos.

b. *"Scaler"*: aplica transformações de escala em colunas numéricas.
   - O método *"minMax_scaler()"* realiza a normalização Min-Max, mas foi expandido para tratar o caso em que todos os valores de uma coluna são iguais, retornando uma lista de zeros ao invés de provocar divisão por zero.
   - O método *"standard_scaler()"* aplica a padronização Z-score, agora com lógica adicional para evitar divisão por zero quando o desvio padrão da coluna é igual a zero.

c. "*Encoder":* Codificação variáveis categóricas com os métodos "*label_encode()" para* atribuir valores numéricos a categorias distintas, e "*oneHot_encode()"*, para gerar colunas binárias para cada categoria. Ambas conseguem lidar com valores *None*, que são convertidos para a categoria especial *"__MISSING__"*. No *label encoding*, esses valores recebem um código numérico único; no *one-hot encoding*, uma nova coluna "*coluna___MISSING__"* é criada para representar esses registros.

3.2 Estratégia de Testes

A validação da biblioteca foi realizada por meio de testes unitários, utilizando o módulo *"unittest"* do Python. Cada classe foi testada isoladamente para garantir o correto funcionamento de suas funções, tratando valores ausentes, escalonando dados e codificando variáveis categóricas. Foram considerados diferentes cenários de uso, como *datasets* com valores nulos, colunas numéricas e categóricas, e casos em que nenhuma coluna era especificada para uma operação. Dessa forma, ficou garantido que os métodos lidariam corretamente com entradas inesperadas ou incompletas. O encadeamento de métodos também foi testado, assegurando que múltiplas operações pudessem ser aplicadas sequencialmente sem erros.

Além desses cenários gerais, foi criada a classe *"TestEdgeCases"*, voltada para situações extremas, que validou a normalização em colunas com valores iguais ou contendo *None* (*"test_minmax_with_none_and_equal_values()")*, a padronização em colunas compostas apenas por valores nulos(*"test_standard_with_only_none()")*, a codificação categórica com *None* (*"test_label_encode_with_none()"* e "*test_onehot_encode_with_none()")*, assegurando a criação da categoria *"__MISSING__",* e a remoção de linhas totalmente nulas(*"test_dropna_all_none_rows()")* , ampliando assim a confiabilidade e a cobertura dos testes.

4. Resultados

Os resultados da implementação das classes de pré-processamento foram obtidos a partir da execução de um arquivo de testes unitários (*test_processing.py*) com diferentes *datasets* simulando variáveis numérica e categóricas da *FoodDelivery*, o que demonstrou a aplicabilidade das técnicas implementadas.

4.1. Tratamento de Valores Ausentes

Primeiramente, foi utilizada a classe *TestMissingValueProcessor* cujo objetivo era verificar se a classe *MissingValueProcessor* identifica corretamente os valores ausentes, fazendo o seu preenchimento ou remoção conforme solicitado.

Tabela 1. *Dataset* utilizado para testar a *class MissingValueProcessor*

idade	salário	cidade
20	500	A
30	None	B
None	800	C
50	1200	None

Utilizando o *dataset* na tabela 1, os resultados de cada função chamado foram os seguintes:

- A função "*isna()"* identificou corretamente o número de linhas com valores nulos(1 na coluna 'idade' e 3 em todas as colunas). É importante citar que, durante a realização deste teste, o retorno esperado estava constando como '4' no arquivo *test_processing.py*, o que não reflete verdadeiramente a quantidade de linhas nulas no dataset. Dessa forma, esse trecho foi corrigido para esperar '3'.
- A função *"notna()"* retornou 2 linhas com valores não nulos quando a busca foi especificada para as colunas "idade" e "salario"
- A função *"fillna()"* preencheu o valor nulo da posição 2 na coluna "idade" com 33.33 (média das idades da coluna). Para fazer o preenchimento com mediana, um segundo dataset foi utilizado(*'cat': ['A', 'B', 'A', None]),* preenchendo o valor da linha 3 com a moda do conjunto *('A').*
- A função "dropna()" removeu a linha com *"None"* da coluna cidade, deixando apenas 3 linhas no dataset final.

Portanto, todos os métodos se comportaram conforme esperado, agindo em correspondência as funções "*dropna()"* e "*fillna()"*da biblioteca externa *"pandas"*, garantindo a integridade do dataset para possíveis análises posteriores

4.2. Normalização e Padronização de Dados Numéricos

Para ajustar corretamente os valores para as escalas desejadas, a classe *Scaler* foi utilizada através da classe *TestScaler,* utilizando o *dataset* "*feature = [10, 20, 30, 40, 50]"*. Os resultados obtidos foram os seguintes:

- A função *"minMax_scaler()"* retornou os valores 0.0, 0.25, 0.5, 0.75 e 1.0.
- A função *"standard_scaler()",* que utiliza *Z-score* ((valor - média)/desvio padrão), retornou os valores -1.4142, -0.7071, 0.0, 0.7071 e 1.4142.

Com os novos testes de casos extremos, confirmou-se que:

- Quando todos os valores são iguais, o "*minMax_scaler()"* retorna [0.0, 0.0, …];
- Quando o desvio padrão é zero, o *"standard_scaler()"* retorna [0.0, …] ao invés de falhar;
- Em colunas compostas apenas por *None*, ambos os métodos funcionam sem levantar erros.

Assim, os valores normalizados e padronizados condizem com cálculos manuais de Min-Max e *Z-score* e coincidem com os resultados das funções *MinMaxScaler()* e *StandardScaler()* da biblioteca *scikit-learn*.

4.3. Codificação de Variáveis Categóricas

Para a codificação de variáveis categóricas, a classe *TestEncoder* foi utilizada testando a classe *Encoder,* utilizando o *dataset "cor = ['azul', 'verde', 'vermelho', 'azul']"*. As funções aplicadas retornaram os seguintes resultados:

- A função *"label_encode()"* retornou os valores 0, 1, 2 e 0, ordenando as cores por ordem alfabética.
- A função *"oneHot_encode()"* criou as colunas "cor_azul", "cor_verde" e "cor_vermelho".

Nos testes de casos extremos, verificou-se que valores *None* são convertidos em *"__MISSING__"*. Assim, o *label encoding* gera um código exclusivo para essa categoria, enquanto o *one-hot encoding* cria a coluna "*cor___MISSING__"*, permitindo que os dados ausentes sejam representados de forma explícita e consistente.

Dessa maneira, a codificação se mostrou funcional e gerou saídas equivalentes Às que seriam obtidas com o uso das funções *LabelEncoder()* e *OneHotEncoder()* do *scikit-learn*. prontas para outras análises e modelagens.

4.4. Testes da Classe Preprocessing

O objetivo principal da classe *Processing* é invocar corretamente os métodos das classes internas. Para testá-la foi utilizada a classe *TestPreprocessingFacade* com o uso de *mocks(MockStats, MockEncoder, MockScaler e MockMVP)* para simular o comportamento das classes internas sem alterar os dados originais. Como resultado, todos os atalhos chamaram as classes internas correspondentes e um erro foi levantado quando um método inválido foi passado para a função *"scale()"*. Assim, a classe funcionou como interface centralizada, simplificando a utilização das classes de pré-processamento.

5. Considerações finais

O desenvolvimento deste módulo resultou em uma arquitetura de pré-processamento de dados robusta e modular, na qual a classe Preprocessing atua como um facade para orquestrar as transformações necessárias. O tratamento explícito de casos extremos nas classes "*Scaler"* e no "*Encoder"* ampliaram a aplicabilidade da biblioteca em datasets reais, que frequentemente apresentam dados degenerados ou ausentes. A criação da classe *"TestEdgeCases"* garantiu que cenários de borda fossem contemplados, fortalecendo a confiabilidade e a cobertura de testes.

Ainda que a implementação atual seja satisfatória, reconhece-se como desafio futuro a otimização do *one-hot encoding* em variáveis de alta cardinalidade, evitando

explosão dimensional. No entanto, os avanços alcançados já demonstram a maturidade da arquitetura e sua capacidade de lidar com contextos reais de pré-processamento de dados.

Um dos desafios que surgiram na implementação foi durante o tratamento de valores vazios, pois, na implementação da classe *Statistics,* a possibilidade de listas com valores vazios nos cálculos de métricas centrais não havia sido considerada. Assim, a classe *Statistics* precisou ser alterada para contemplar esse cenário. No que tange a complexidade de desenvolvimento, acreditamos que a classe *Scaler* tem conceitos um pouco mais complexos de serem entendidos se comparados às outras classes.

6. Referências

Carbtee, M. e Nehme, A. (2024) "O que é Análise de Dados? Guia para Especialistas". DataCamp Blog. Disponível em: https://www.datacamp.com/pt/blog/what-is-data-analysis-expert-guide. Acesso em: 17 set. 2025.

Equipe DSA (2025) "Guia Definitivo para o Tratamento de Valores Ausentes em Data Science: os Três Mecanismos de Ausência de Dados". Blog DSA. Disponível em: https://blog.dsacademy.com.br/guia-definitivo-para-o-tratamento-de-valores-ausentes-em-data-science-os-tres-mecanismos-de-ausencia-de-dados/. Acesso em: 17 set. 2025.

Gomes, P. C. T. (2019) "Pré-processamento de Dados". DataGeeks. Disponível em: https://www.datageeks.com.br/pre-processamento-de-dados/. Acesso em: 17 set. 2025.

Kelta, Z. (2024) "Técnicas para Lidar com Valores de Dados Faltantes". DataCamp Tutorial. Disponível em: https://www.datacamp.com/pt/tutorial/techniques-to-handle-missing-data-values. Acesso em: 17 set. 2025.

Oliveira Júnior, C. (2023) "Feature Engineering: Técnicas para Lidar com Dados Faltantes em um Projeto de Ciência de Dados". Medium (Data Hackers). Disponível em: https://medium.com/data-hackers/feature-engineering-t%C3%A9cnicas-para-lidar-com-dados-faltantes-em-um-projeto-de-ci%C3%AAncia-de-dados-debdd57eb662. Acesso em: 17 set. 2025.

Pykes, K. (2025) "Pré-processamento de Dados". DataCamp Blog. Disponível em: https://www.datacamp.com/pt/blog/data-preprocessing. Acesso em: 17 set. 2025.

Reichert Junior, I. (2023) "Pré-processamento de Dados: o que é, por que fazer?". Medium. Disponível em: https://medium.com/@ingoreichertjr/pr%C3%A9-processamento-de-dados-o-que-%C3%A9-por-que-fazer-dfc9fa3df8a3. Acesso em: 17 set. 2025.

Shalbu, S. (2024) "Normalização vs Padronização". DataCamp Tutorial. Disponível em: https://www.datacamp.com/pt/tutorial/normalization-vs-standardization. Acesso em: 17 set. 2025.