Preprocessing: Biblioteca de Pré-Processamento de Dados em Python

Fátima P. Santos Pinho[1], Ismael R. de Oliveira Neto[2], Rebeca Helen B. Amorim[3]

[1]Curso de Sistemas de Informação – Centro Universitário de Excelência (UNEX)
Av. Artêmia Pires Freitas – s/n – Sim – 44085-370 – Feira de Santana, BA – Brasil

fatmapinh0@gmail.com,
rebecabatista165@gmail.com,ismaelneto2003@gmail.com

Resumo. *Este artigo apresenta um relatório técnico que descreve o desenvolvimento de uma biblioteca de pré-processamento de dados em Python, implementada utilizando apenas recursos nativos da linguagem e uma biblioteca estatística base. São abordadas a importância crítica desta etapa no ciclo de vida de projetos de machine learning, a arquitetura de software modular adotada, a aplicação prática das técnicas de tratamento de dados e os resultados obtidos através de uma suíte de testes unitários que validam a sua funcionalidade.*

1. Introdução

O pré-processamento de dados constitui uma etapa fundamental no pipeline de ciência de dados e aprendizado de máquina. Esta fase é responsável pela transformação de dados brutos em um formato adequado para análise e modelagem, garantindo qualidade, consistência e compatibilidade com algoritmos de *machine learning*.

No contexto do desenvolvimento de modelos de classificação, o pré-processamento assume papel crítico. Dados mal preparados podem levar a modelos com baixa precisão ou que não se generalizem adequadamente para novos conjuntos de dados. Através de técnicas como tratamento de valores ausentes, normalização e codificação de variáveis categóricas, é possível melhorar significativamente o desempenho dos algoritmos.

Este relatório apresenta o desenvolvimento de uma biblioteca robusta de pré-processamento de dados em Python, implementada utilizando apenas recursos nativos da linguagem e a biblioteca *Statistics* desenvolvida na etapa anterior deste projeto. A solução aborda os principais desafios encontrados na preparação de dados para modelos de aprendizado de máquina.

2. Fundamentação Teórica

O pré-processamento de dados engloba um conjunto sistemático de técnicas aplicadas aos dados brutos para transformá-los em formato adequado para análise e modelagem. Segundo Guimarães et al. (2019), esta etapa pode representar até 80% do tempo total de um projeto de ciência de dados, evidenciando sua importância crítica no processo analítico.

O fluxo típico de pré-processamento segue a seguinte estrutura:

Dados brutos → Limpeza → Transformação → Integração → Dados preparados.

A seguir, detalhamos os conceitos que fundamentam as funcionalidades da nossa biblioteca.

2.1 Tratamento de dados ausentes

Quando há um valor faltando em um conjunto de dados, usamos esse tratamento para evitar perda de dados importantes causadas por uma exclusão de linha ou coluna, preenchendo os valores ausentes através de algumas técnicas como Imputação, que utiliza uma estimativa calculada, como média, moda ou a mediana dos dados disponíveis. Na figura 1 é possível observar uma imputação por média.

```
ALGORITMO ImputarPorMedia(coluna)
INÍCIO
    valores_validos ← []
    PARA CADA valor em coluna FAÇA
        SE valor não é nulo ENTÃO
            Adicionar valor a valores_validos
        FIM SE
    FIM PARA

    SE valores_validos não está vazio ENTÃO
        media ← Somar(valores_validos) / Tamanho(valores_validos)
        PARA CADA posição em coluna FAÇA
            SE coluna[posição] é nulo ENTÃO
                coluna[posição] ← media
            FIM SE
        FIM PARA
    FIM SE
FIM
```

Figura 1: Pseudocódigo de Imputação por Média

Imagine que, em uma lista de notas de alunos, uma das notas está faltando. A imputação por média pode ser útil nesse tipo de situação, fazendo o cálculo das notas existentes para preencher o valor ausente e garantir que o aluno não seja prejudicado.

A Figura 2 demonstra exatamente essa situação na prática. No exemplo, a coluna de notas de matemática possui um valor ausente (*None*). O algoritmo resolve isso calculando a média apenas das notas válidas ([8.5, 7.2, 9.1, 6.8]), que resulta em 7.9. Por fim, o valor *None* é substituído por essa média, resultando em uma coluna de dados completa e consistente.

```python
# Dataset de notas com valores ausentes
notas_estudantes = {
    'matematica': [8.5, 7.2, None, 9.1, 6.8],
    'portugues': [7.8, None, 8.3, 8.7, 7.1]
}

# Aplicando imputação por média em matemática:
# Valores válidos: [8.5, 7.2, 9.1, 6.8]
# Média = (8.5 + 7.2 + 9.1 + 6.8) / 4 = 7.9
# Resultado: [8.5, 7.2, 7.9, 9.1, 6.8]
```

Figura 2: Exemplo de Imputação por Média

Em outros casos podem ser utilizados também a Modelagem de valores ausentes que trata a ausência como uma informação, criando uma coluna para indicar se o dado estava

faltando ou não (0 ou 1) ou a exclusão, a qual como o próprio nome indica, irá remover linhas ou colunas que possuem valores nulos. Ela deve ser usada com cautela porque, apesar de ser um método prático, a maior desvantagem de excluir uma linha ou coluna é a perda dos outros dados que poderiam ser muito relevantes. Na figura 3 visualizamos um pseudocódigo que trás um exemplo de exclusão.

```
ALGORITMO ExcluirValoresAusentes(dataset)
INÍCIO
    dataset_limpo ← {}
    PARA CADA linha em dataset FAÇA
        SE linha não contém valores nulos ENTÃO
            Adicionar linha a dataset_limpo
        FIM SE
    FIM PARA
    RETORNAR dataset_limpo
FIM
```

Figura 3: Pseudocódigo de Exclusão de Valores Ausentes

2.2 Detecção e remoção de outliers

Outliers, ou valores discrepantes, são pontos de dados que se diferenciam significativamente da maioria dos outros em um conjunto. Para evitar distorção em análises estatísticas causadas por *outliers* e correr o risco de prejudicar o desempenho de modelos, nós aplicamos técnicas que irão identificar essas discrepâncias, utilizando alguns métodos.

São eles: *Z-score*, que mede a distância de um ponto em relação à média em termos de desvios-padrão, IQR (*Amplitude Interquartil*), que define um intervalo considerado normal com base nos quartis dos dados, ou até mesmo técnicas visuais, como gráficos de caixa, gráficos de dispersão e histogramas, que podem auxiliar na identificação de exceções em um conjunto de dados. Após isso, os valores discrepantes podem ser removidos ou transformados a depender da influência na análise.

A figura 4 traz um pseudocódigo exemplo de O *Z-Score* que mede quantos desvios-padrão um valor está distante da média.

```
ALGORITMO DetectarOutliersZScore(dados, limiar=3.0)
INÍCIO
    media ← CalcularMedia(dados)
    desvio_padrao ← CalcularDesvioPadrao(dados)
    outliers ← []

    PARA CADA valor em dados FAÇA
        z_score ← ABS(valor - media) / desvio_padrao
        SE z_score > limiar ENTÃO
            Adicionar valor a outliers
        FIM SE
    FIM PARA

    RETORNAR outliers
FIM
```

Figura 4: Pseudocódigo Detecção de outliers

Podemos usar essas técnicas em situações como quando analisamos a altura de um grupo de pessoas e encontramos um valor muito distante da maioria, por exemplo, um registro de 2,80m em uma lista de alturas de estudantes do ensino médio. Nesse caso,

utilizando o método do IQR, esse ponto seria facilmente identificado como discrepante, pois está fora do intervalo esperado definido pelos *quartis*.

A detecção e remoção de *outliers* é aplicada em diversos contextos, como em estudos de mercado, onde valores extremos de renda podem distorcer a análise de consumo, ou em medições de sensores, onde uma leitura anormal pode indicar falha do equipamento em vez de refletir a realidade. Assim, identificar e tratar outliers é fundamental para que os resultados obtidos sejam confiáveis e representem melhor os dados coletados.

2.3 Codificação de dados

Muitos algoritmos de *machine learning* funcionam apenas com dados numéricos. A codificação de dados, ou *encoding*, é o processo de converter variáveis categóricas. Variáveis categóricas como "cor", "gênero" ou "cidade" precisam ser convertidas em representações numéricas para serem processadas pelos modelos (James et al., 2013). As duas técnicas mais comuns para isso são:

Label Encoding: Recomendado para categorias que possuem um ordenamento natural ou poucas classes, substitui a categoria por um valor inteiro, ou seja, simplesmente atribui um número inteiro diferente para cada categoria (ex: Considere uma coluna com os valores ["Bom", "Ruim", "Ótimo", "Bom", "Regular"]. Para preservar a hierarquia, definimos um mapa de regras onde "Ruim" se torna 1, "Regular" se torna 2, "Bom" se torna 3, e "Ótimo" se torna 4. Ao aplicar essa transformação, a lista original é convertida para sua representação numérica ordinal, resultando em [3, 1, 4, 3, 2]. É possível visualizar o exemplo na figura 5).

```
// 1. DADOS ORIGINAIS: A coluna 'avaliacao_servico' com valores em texto.
   COLUNA 'avaliacao_servico' (ANTES):
   ["Bom", "Ruim", "Ótimo", "Bom", "Regular"]

// 2. REGRAS DE TRANSFORMAÇÃO: Definimos a regra para cada nível de satisfação.
   SE valor é "Ruim"     ENTÃO substituir por 1
   SE valor é "Regular"  ENTÃO substituir por 2
   SE valor é "Bom"      ENTÃO substituir por 3
   SE valor é "Ótimo"    ENTÃO substituir por 4

// 3. PROCESSO: O algoritmo percorre a coluna aplicando as regras.
   "Bom"     -> se torna -> 3
   "Ruim"    -> se torna -> 1
   "Ótimo"   -> se torna -> 4
   "Bom"     -> se torna -> 3
   "Regular" -> se torna -> 2

// 4. RESULTADO FINAL: A coluna 'avaliacao_servico' agora é numérica e ordinal.
   COLUNA 'avaliacao_servico' (DEPOIS):
   [3, 1, 4, 3, 2]
```

Figura 5: Pseudocódigo Label Encoding

One-Hot Encoding: Cria novas colunas binárias (0 ou 1) para cada categoria, indicando a presença ou ausência dela. Sem essa etapa, informações valiosas contidas em colunas de texto seriam ignoradas. (ex: Na figura 6 vemos um caso de uso simples para *One-Hot Encoding* é uma coluna com apenas duas categorias, como ativo, contendo [“Sim”, “Não”, “Sim”]. A transformação converte a coluna de texto em uma única coluna numérica, onde "Sim" se torna 1 e "Não" se torna 0, resultando na lista [1, 0, 1]).

```
// ANTES: Coluna 'ativo' com duas categorias.
   ["Sim", "Não", "Sim"]

// PROCESSO: 'Sim' vira 1, 'Não' vira 0.

// DEPOIS: A coluna é transformada em uma única coluna numérica.
   [1, 0, 1]
```

Figura 6: Pseudocódigo One-Hot Encoding

2.4 Normalização dos dados

Algumas vezes certos atributos apresentam escalas diferentes. Dessa forma, pode ser útil realizar uma transformação a fim de evitar que um atributo predomine sobre outro (isso pode acontecer por exemplo na otimização do gradiente descendente). Contudo, existem situações em que essa variação deve ser preservada por ser importante para o modelo.

A normalização é uma das técnicas utilizadas para essa transformação, recomendável quando os limites de valores de atributos distintos são muito diferentes. É aplicada a cada atributo individualmente e pode ocorrer por:

Normalização por reescala (escala): É também chamada de normalização min-max. São definidos valores min e max para os novos valores de cada atributo. Podendo apresentar 0 como valor mínimo e 1 como valor máximo, ou seja, os dados estão em um intervalo de 0 a 1.

```
FUNÇÃO normalizar_min_max(valor, valor_min, valor_max):
    // Retorna o valor normalizado aplicando a fórmula min-max.
    RETORNO (valor - valor_min) / (valor_max - valor_min)
FIM FUNÇÃO
```

Figura 7: Pseudocódigo de Normalização (Min-Max)

Normalização por padronização (ou padronização): Utiliza a média e o desvio padrão. O resultado disso é uma média igual a 0 e um desvio padrão igual a 1.

3. Metodologia

3.1 Desenvolvimento da Biblioteca

O desenvolvimento seguiu uma abordagem modular, a biblioteca foi estruturada em quatro classes principais:

1. *MissingValueProcessor*: Responsável pelo tratamento de valores ausentes.
   - ➔ Esta classe é dedicada exclusivamente ao tratamento de valores ausentes (representados como *None* no *dataset*).
   - ➔ Métodos *isna*() e *notna*(): Estes métodos servem para filtrar o *dataset*. O *isna*() retorna um novo *dataset* contendo apenas as linhas que possuem pelo menos um valor nulo, enquanto o *notna*() retorna as linhas que não possuem nenhum valor nulo.
   - ➔ Método *fillna*(): Preenche os valores ausentes (*None*) em uma ou mais colunas usando uma estratégia definida. O método suporta diferentes técnicas (*mean, median, mode*). Para os métodos estatísticos, ele primeiro cria uma lista "limpa", sem os valores nulos, para calcular a métrica

desejada (média, mediana ou moda) utilizando a classe *Statistics*. Em seguida, ele percorre a coluna original e substitui cada *None* pelo valor calculado.

- ➔ Método *dropna*(): Remove as linhas que contêm valores ausentes. O método primeiro identifica os índices de todas as linhas que são "válidas" (ou seja, não contêm *None* nas colunas especificadas). Depois, ele constrói um novo *dataset* a partir do zero, incluindo apenas os dados das linhas cujos índices foram marcados como válidos.

2. *Scaler*: Implementa técnicas de normalização e padronização.
   - ➔ Esta classe é responsável por aplicar transformações de escala em colunas numéricas, garantindo que as variáveis tenham magnitudes comparáveis.
   - ➔ Método *minMax_scaler*() (Normalização): Ajusta os valores de uma coluna para que fiquem em um intervalo entre 0 e 1. Para cada coluna, o código primeiro encontra os valores mínimo e máximo. Em seguida, ele aplica a fórmula matemática de Min-Max: (valor - min) / (max - min) a cada elemento da coluna. Há também um tratamento para o caso extremo em que todos os valores da coluna são iguais (o que resultaria em uma divisão por zero); nesse cenário, todos os valores são convertidos para 0.0.
   - ➔ Método *standard_scaler*() (Padronização): Transforma os valores de uma coluna para que a nova distribuição tenha uma média de 0 e um desvio padrão de 1. Utiliza a classe *Statistics* para calcular a média e o desvio padrão da coluna. Depois, aplica a fórmula *Z-score*: (valor - media) / desvio_padrao a cada elemento. Assim como no Min-Max, ele também trata o caso extremo de um desvio padrão zero, convertendo todos os valores para 0.0.
3. *Encoder*: Gerencia a codificação de variáveis categóricas.
   - ➔ O objetivo desta classe é converter colunas com dados categóricos (texto) em um formato numérico que possa ser utilizado por algoritmos de *machine learning*.
   - ➔ Método *label_encode*(): Atribui um número inteiro único para cada categoria de uma coluna. O método primeiro identifica todas as categorias únicas e as ordena alfabeticamente para garantir que a codificação seja consistente. Em seguida, ele cria um dicionário de "mapeamento". Por fim, ele substitui cada valor de texto na coluna original pelo número correspondente no mapa.
   - ➔ Método *oneHot_encode*(): Cria novas colunas binárias (0 ou 1) para cada categoria presente na coluna original. Para cada categoria única em uma coluna, o método cria uma nova coluna. Ele preenche essa nova coluna com 1 se o valor na linha original corresponder àquela categoria, e com 0 caso contrário. Após criar todas as novas colunas binárias, a coluna de texto original é removida do *dataset*.
4. *Preprocessing*: Classe principal que orquestra todas as funcionalidades.
   - ➔ Esta é a classe principal que o usuário interage.
   - ➔ Construtor (*__init__*): Ao ser criada, a classe primeiro valida se todas as colunas do dataset têm o mesmo tamanho. Em seguida, ela inicializa instâncias de todas as outras classes (*MissingValueProcessor, Scaler, Encoder*), centralizando todas as funcionalidades em um só lugar.
   - ➔ Métodos de Atalho (*fillna, scale, encode*, etc.): Os métodos públicos desta classe (*.scale*(), *.encode*(), etc.) não contêm a lógica de pré-processamento em si. Eles simplesmente delegam a chamada para o método correspondente na classe especializada apropriada (ex: *self.scale*() chama

self.scaler.minMax_scaler()).

3.2 Reutilização da Biblioteca Statistics

A biblioteca *Statistics*, desenvolvida na primeira etapa do projeto, foi extensivamente reutilizada para cálculos de medidas de tendência central e dispersão. Esta integração demonstra a importância da modularidade no desenvolvimento de software e permite maior confiabilidade nos cálculos estatísticos.

3.3 Realização dos testes

Para garantir que cada funcionalidade da biblioteca se comportasse exatamente como o esperado, foi adotada uma metodologia de testes unitários. Utilizando a ferramenta *unittest* do Python, foram implementados testes específicos para cada método do projeto. Em cada teste, o código é executado com um conjunto de dados de exemplo e seu resultado é comparado com uma saída esperada através de verificações automáticas, assegurando a correção da lógica implementada.

Além disso, essa metodologia foi expandida com a criação da classe *TestEdgeCases*, uma suíte de testes dedicada a validar o comportamento do sistema em casos de borda e extremos. Estes novos testes verificam cenários não convencionais, como a aplicação de Scaler em colunas com valores idênticos para prevenir erros de divisão por zero. Também foi verificado o comportamento de *MissingValueProcessor* em colunas contendo apenas valores nulos e a inicialização da classe Preprocessing com um *dataset* vazio. Por fim, foram criados testes para o *Encoder* com dados de tipos mistos e *None* para identificar os limites de funcionalidade da biblioteca e garantir seu comportamento previsível diante de entradas inválidas.

4. Resultados

A aprovação da biblioteca foi realizada através da execução de uma suíte de testes unitários (*test_preprocessing.py*). Durante a fase de testes, foram identificados e corrigidos dois detalhes no arquivo de verificação para garantir sua correta execução. O primeiro foi um erro de lógica no teste *test_isna*, onde a asserção esperava que o dataset de exemplo retornasse 4 linhas com valores nulos, quando o número correto era 3. O segundo foi um ajuste técnico nos decoradores *@patch* da classe *TestPreprocessingFacade*, que apontavam para um módulo inexistente (*preprocessing_lib*) e foram corrigidos para o caminho correto (*preprocessing*).

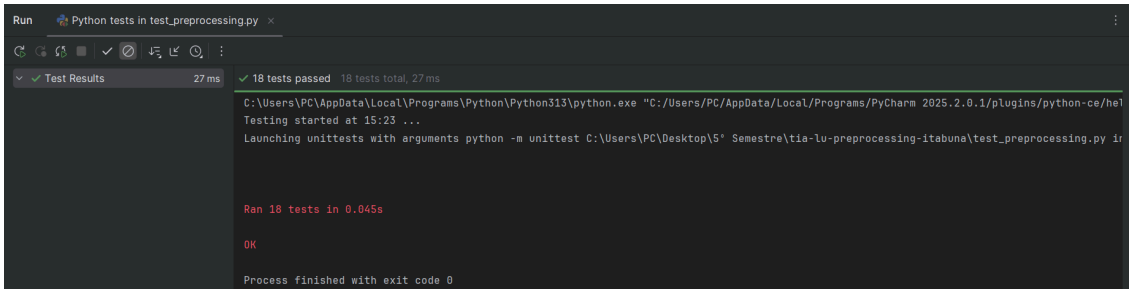

Figura 7: Execução do arquivo de testes

Após esses dois ajustes, como mostrado na figura 7, a suíte de testes foi executada completamente e todos os testes passaram com sucesso, validando a funcionalidade da biblioteca conforme detalhado abaixo:

Tratamento de Valores Ausentes: Os testes para a classe *MissingValueProcessor* validaram que os métodos *isna* e *notna* identificam corretamente as linhas com e sem valores nulos, enquanto *fillna* e *dropna* manipularam os dados ausentes com sucesso. O comportamento dessas funções demonstrou ser consistente com os resultados esperados de funções análogas na biblioteca pandas.

Normalização e Padronização: No teste da classe *Scaler*, os métodos *minMax_scaler* e *standard_scaler* normalizaram e padronizaram os dados de exemplo corretamente. Os resultados foram numericamente equivalentes aos que seriam obtidos com as funções da biblioteca scikit-learn.

Codificação de Dados: Os testes da classe *Encoder* confirmaram que *label_encode* e *oneHot_encode* converteram corretamente as categorias de texto em formatos numéricos, com saídas consistentes com as das funções correspondentes no *scikit-learn*.

Validação da Arquitetura (*Facade*): Os testes para a classe *Preprocessing* utilizaram *mocks* para simular as classes internas, confirmando que a fachada delega as chamadas aos métodos corretos das classes especializadas e, assim, validando a eficácia da arquitetura do projeto.

Para ilustrar a aplicação integrada da biblioteca, a Figura 8 demonstra um fluxo completo de pré-processamento sobre um *dataset* de exemplo. O objetivo é transformar dados brutos com múltiplas inconsistências em um formato limpo e pronto para ser utilizado em um modelo de *machine learning*.

```
# 1. Dataset inicial com valores ausentes e escalas diferentes
dados_brutos = {
    'produto': ['A', 'B', 'A', 'C', None],
    'vendas': [2000, 1500, 3000, None, 2200],
    'satisfacao': [4.5, 3.0, None, 4.8, 4.1]
}

# 2. Instanciando e aplicando o pré-processamento em cadeia
processador = Preprocessing(dados_brutos)
processador.fillna(columns={'vendas'}, method='mean')      \
           .fillna(columns={'satisfacao', 'produto'}, method='mode') \
           .scale(columns={'vendas'}, method='minMax')      \
           .encode(columns={'produto'}, method='label')

# 3. Resultado: dados limpos e prontos para modelagem
# print(processador.dataset)
# Saída esperada (aproximada):
# {
#   'produto': [0, 1, 0, 2, 0],
#   'vendas': [0.33, 0.0, 1.0, 0.48, 0.46],
#   'satisfacao': [4.5, 3.0, 4.5, 4.8, 4.1]
# }
```

Figura 8: Exemplo de aplicação da técnica sobre um dataset de exemplo

Comparativo com Bibliotecas Existentes: A validação através dos testes unitários permitiu não apenas confirmar a correção da nossa biblioteca, mas também comparar seu comportamento com as funções equivalentes das bibliotecas padrão de mercado, como *Pandas* e *Scikit-learn*. A tabela 1 abaixo resume a equivalência funcional que foi verificada, demonstrando que nossa implementação se comporta de maneira consistente com as ferramentas profissionais.

Tabela 1: Comparativo com bibliotecas existentes

Funcionalidade	Método na Nossa Biblioteca	Equivalente em Bibliotecas Padrão
Tratamento de Ausentes		
Preenchimento de Nulos	MissingValueProcessor.fillna()	pandas.DataFrame.fillna()
Remoção de Nulos	MissingValueProcessor.dropna()	pandas.DataFrame.dropna()
Ajuste de Escala		
Normalização (Min-Max)	Scaler.minMax_scaler()	sklearn.preprocessing.MinMaxScaler
Padronização (Z-score)	Scaler.standard_scaler()	sklearn.preprocessing.StandardScaler

5. Considerações Finais

Através do desenvolvimento deste projeto foi possível a criação bem-sucedida de uma biblioteca de pré-processamento de dados funcional, implementada inteiramente com recursos nativos do Python. A adoção do padrão de arquitetura *Facade* se mostrou eficaz, proporcionando uma interface de uso limpa e um código bem organizado. A validação completa através de uma suíte de testes unitários, incluindo casos de borda, garantiu a confiabilidade e a correção da lógica implementada em cada um dos componentes.

Durante o desenvolvimento, a principal complexidade encontrada foi garantir a robustez da biblioteca diante de cenários extremos. A implementação da classe *Scaler*, por exemplo, exigiu um tratamento cuidadoso para evitar erros de divisão por zero em colunas com valores idênticos. Além disso, a integração com a biblioteca *Statistics*, desenvolvida na etapa anterior, apresentou desafios, como a necessidade de refatorar seu construtor para lidar corretamente com *datasets* vazios, uma condição descoberta durante a criação dos testes de casos de borda.

Como trabalhos futuros e melhorias, a classe *Encoder* poderia ser aprimorada para lidar de forma mais resiliente com valores ausentes (*None*) ou tipos de dados mistos, tratando-os como uma categoria especial em vez de gerar um erro. Essas melhorias aumentariam ainda mais a aplicabilidade da biblioteca em cenários de dados do mundo real.

6. Referências

ASTERA. O que é pré-processamento de dados? Definição, conceitos, importância, ferramentas. [Internet]. 13 mar. 2025. Disponível em: https://www.astera.com/pt/type/blog/data-preprocessing/. Acesso em: 16 set. 2025.

IBM. *O que é um pipeline de aprendizado de máquina?* [Internet]. [s.d.]. Disponível em: https://www.ibm.com/br-pt/think/topics/machine-learning-pipeline/. Acesso em: 16 set. 2025.

GUIMARÃES, L. M. S. et al. Avaliação das etapas de pré-processamento e de análise de atributos para classificação de dados. *Perspectivas em Ciência da Informação*, v. 24, n. 2, 2019. Disponível em: https://www.scielo.br/j/pci/a/p55xWvrLwj3zTjbscWdtp4M/. Acesso em: 16 set. 2025.

SANTOS, L. A. dos et al. Um pipeline de pré-processamento de dados textuais em português para análise de redes sociais. *Anais do STIL 2024*. SBC, 2024. Disponível em: https://sol.sbc.org.br/index.php/stil/article/view/31163. Acesso em: 22 set. 2025.

HOSOUME, J. M. Recomendação de técnicas de pré-processamento por meta-aprendizado. 2020. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) – Universidade de Brasília. Disponível em: https://bdm.unb.br/bitstream/10483/27587/1/2020_JulianaMayumiHosoume_tcc.pdf. Acesso em: 22 set. 2025.

LOUREIRO, A. A. B.; PEREIRA, G. G. Manipulação de dados para modelagem em aprendizado de máquina. *Revista Acadêmica Souza EAD*, v. 80, dez. 2024. Disponível em: https://souzaeadrevistaacademica.com.br/revista/80-dezembro-2024/02-arthur-afonso-bitencourt-loureiro-artigo.pdf. Acesso em: 22 set. 2025.

DATACAMP. Pré-processamento de dados: Um guia completo com exemplos em Python. [*S. l.*]: DataCamp, 23 maio 2023. Disponível em: https://www.datacamp.com/pt/blog/data-preprocessing. Acesso em: 22 set. 2025.