Sumário

[1. Numpy 2](#_Toc190772453)

[Introdução ao NumPy 2](#_Toc190772454)

[Arrays e Aprendizado de Máquina 3](#_Toc190772455)

[Array (NumPy) 3](#_Toc190772456)

[**Resumo** 3](#_Toc190772457)

[Broadcasting (NumPy) 3](#_Toc190772458)

[Funções Lambda 4](#_Toc190772459)

[**np.zeros()** 5](#_Toc190772460)

[**np.arange()** 5](#_Toc190772461)

[**np.random.random()** 5](#_Toc190772462)

[**list(range(3))** 6](#_Toc190772463)

[**Resumo das Diferenças** 6](#_Toc190772464)

[Arrays NumPy vs. Listas em Python 8](#_Toc190772465)

[**Exemplos Práticos** 8](#_Toc190772466)

[**Resumo** 9](#_Toc190772467)

[**O que é um Array 3D?** 9](#_Toc190772468)

[**Aplicações de Arrays 3D** 10](#_Toc190772469)

[**Como Criar um Array 3D em NumPy** 10](#_Toc190772470)

[**Resumo** 11](#_Toc190772471)

[Vetores 11](#_Toc190772472)

[Matrizes 11](#_Toc190772473)

[Tensores 12](#_Toc190772474)

[**Exemplos Práticos com NumPy** 12](#_Toc190772475)

[**Resumo** 13](#_Toc190772476)

[### Introdução ao Pandas 14](#_Toc190772477)

[### O Que é um DataFrame? 14](#_Toc190772478)

[# Criando um DataFrame a partir de um dicionário 14](#_Toc190772479)

[### Explorando um DataFrame 14](#_Toc190772480)

[### Componentes de um DataFrame 14](#_Toc190772481)

[### Ordenação e Conjuntos no DataFrame 14](#_Toc190772482)

[### Subconjuntos e Condições Múltiplas 14](#_Toc190772483)

[### Múltiplas Manipulações 15](#_Toc190772484)

[### Métodos para Estatística Cumulativa 15](#_Toc190772485)

[### Lidando com Valores Duplicados 15](#_Toc190772486)

[Agrupamento de Dados 15](#_Toc190772487)

[Tabelas Pivot 15](#_Toc190772488)

[Índices e Fatiamento de um DataFrame 15](#_Toc190772489)

[Trabalhando com Datas 15](#_Toc190772490)

[Gráficos e Valores Faltosos 15](#_Toc190772491)

[Tratamento de Valores Ausentes 17](#_Toc190772492)

[Codificação de Variáveis Categóricas 17](#_Toc190772493)

[Normalização ou Padronização 17](#_Toc190772494)

[Conversão de Tipos de Dados 17](#_Toc190772495)

[drop\_duplicates() 17](#_Toc190772496)

[groupby() e agg() 17](#_Toc190772497)

[Criação de Novas Variáveis (Feature Engineering) 17](#_Toc190772498)

[Divisão de Dados 18](#_Toc190772499)

[Mudança do Formato do DataFrame 18](#_Toc190772500)

[str.replace() 21](#_Toc190772501)

[get\_dummies e sep 21](#_Toc190772502)

# Numpy

## Introdução ao NumPy

NumPy é uma biblioteca fundamental para a computação científica em Python. Ela fornece suporte para arrays multidimensionais e uma vasta coleção de funções matemáticas para operar sobre esses arrays de forma eficiente.

**Bibliotecas Construídas a Partir do NumPy**

Várias bibliotecas populares foram construídas com base no NumPy, aproveitando suas capacidades de manipulação de arrays e operações matemáticas. Entre elas estão:

* **Pandas**: Utilizada para análise de dados, fornece estruturas de dados como DataFrames, que facilitam a manipulação e análise de dados tabulares.
* **SciPy**: Extensão do NumPy, oferece funções adicionais para integração, otimização, álgebra linear, estatísticas e muito mais.
* **Matplotlib**: Biblioteca de plotagem que permite a criação de gráficos e visualizações de dados.
* **Scikit-learn**: Biblioteca de aprendizado de máquina que fornece ferramentas para modelagem preditiva, incluindo classificação, regressão, clustering e redução de dimensionalidade.
* **TensorFlow**: Framework de aprendizado profundo desenvolvido pelo Google, que utiliza arrays do NumPy para operações de tensor.

## Arrays e Aprendizado de Máquina

Os arrays do NumPy são a base para muitas operações em aprendizado de máquina. Eles permitem a manipulação eficiente de grandes volumes de dados, o que é essencial para o treinamento e a implementação de modelos de aprendizado de máquina. Com o NumPy, é possível realizar operações como:

* **Criação de Arrays**: Arrays podem ser criados a partir de listas, tuplas ou gerados usando funções como numpy.arange e numpy.linspace.
* **Operações Matemáticas**: Operações elementares como soma, subtração, multiplicação e divisão podem ser realizadas diretamente nos arrays.
* **Manipulação de Dados**: Funções para reshaping, slicing e indexing permitem a reorganização e acesso eficiente aos dados.
* **Álgebra Linear**: Funções para multiplicação de matrizes, decomposição de valores singulares (SVD) e resolução de sistemas lineares.

NumPy é, portanto, uma ferramenta essencial para qualquer pessoa que trabalhe com ciência de dados, análise de dados ou aprendizado de máquina em Python.

## Array (NumPy)

* **Definição**: Um array é uma coleção ordenada de elementos do mesmo tipo, fornecida pela biblioteca NumPy.
* **Sintaxe**: Arrays são criados usando a função numpy.array().
* **Exemplo**:
* import numpy as np
* meu\_array = np.array([1, 2, 3, 4, 5])
* **Características**:
  + **Homogêneo**: Todos os elementos devem ser do mesmo tipo (por exemplo, todos inteiros ou todos floats).
  + **Mutável**: Você pode alterar os elementos após a criação.
  + **Eficiente**: Operações matemáticas em arrays são mais rápidas e eficientes em comparação com listas.

**Resumo**

* **Lista**: Mutável, heterogênea, definida com [].
* **Tupla**: Imutável, heterogênea, definida com ().
* **Array**: Mutável, homogêneo, definido com numpy.array().

## Broadcasting (NumPy)

* **Definição**: Técnica que permite que operações em arrays de diferentes formas sejam realizadas sem a necessidade de copiar dados.
* **Exemplo**:
* import numpy as np
* a = np.array([1, 2, 3])
* b = np.array([4, 5, 6])
* c = a + b # Broadcasting permite a soma elemento a elemento
* **Características**:
  + **Eficiência**: Reduz a necessidade de criar arrays temporários, economizando memória e tempo de processamento.
  + **Flexibilidade**: Permite operações entre arrays de diferentes formas, desde que sejam compatíveis.

## Funções Lambda

* **Definição**: Funções anônimas e pequenas definidas usando a palavra-chave lambda.
* **Sintaxe**: lambda argumentos: expressão
* **Exemplo**: soma = lambda x, y: x + y
* **Características**:
  + **Concisas**: Úteis para funções simples e rápidas.
  + **Anônimas**: Não requerem um nome explícito.

Array é o objeto principal do numpy e ele pode ter qualquer dimensão mas ele so consegue manter 1 tipo de dados.

Para começar importamos numpy

Import numpy as np

Como criamos a primeira array?

Por meio de uma lista:

l= [4,8,2,12,5,8,0]

Arr = ny.array (l)

Criando um numpy array do zero

Np.zeros()

Np.arange()

Np.random.random()

**np.zeros()**

* **Definição**: Cria um array de NumPy preenchido com zeros.
* **Sintaxe**: np.zeros(shape, dtype=float)
* **Exemplo**: np.zeros((3, 3)) cria uma matriz 3x3 de zeros.
* **Aplicação em Negócios**: Pode ser usado para inicializar matrizes ou tabelas onde os valores serão preenchidos posteriormente, como em modelos financeiros ou de inventário.

import numpy as np

# Exemplo de uso

matriz\_zeros = np.zeros((3, 3))

print(matriz\_zeros)

**np.arange()**

* **Definição**: Cria um array de NumPy com uma sequência de números.
* **Sintaxe**: np.arange(start, stop, step, dtype=None)
* **Exemplo**: np.arange(0, 10, 2) cria um array com números de 0 a 8, com passo 2.
* **Aplicação em Negócios**: Útil para gerar sequências de datas, índices de tempo ou intervalos de valores para simulações e análises.

import numpy as np

# Exemplo de uso

sequencia = np.arange(0, 10, 2)

print(sequencia)

**np.random.random()**

* **Definição**: Gera um array de NumPy com valores aleatórios entre 0 e 1.
* **Sintaxe**: np.random.random(size)
* **Exemplo**: np.random.random((2, 3)) cria uma matriz 2x3 com valores aleatórios.
* **Aplicação em Negócios**: Pode ser usado para simulações de Monte Carlo, geração de dados de teste ou modelagem de incertezas em previsões financeiras.

import numpy as np

# Exemplo de uso

valores\_aleatorios = np.random.random((2, 3))

print(valores\_aleatorios)

**list(range(3))**

* **Definição**: Cria uma lista em Python com uma sequência de números.
* **Sintaxe**: list(range(start, stop, step))
* **Exemplo**: list(range(3)) cria uma lista [0, 1, 2].
* **Aplicação em Negócios**: Útil para criar listas de índices, iterações em loops ou para inicializar estruturas de dados simples.

# Exemplo de uso

lista = list(range(3))

print(lista)

**Resumo das Diferenças**

* **np.zeros()**: Cria arrays preenchidos com zeros, útil para inicialização.
* **np.arange()**: Cria sequências de números, útil para gerar intervalos.
* **np.random.random()**: Gera valores aleatórios, útil para simulações.
* **list(range(3))**: Cria listas de sequências, útil para iterações simples.

A função np.random.randint() do NumPy é usada para gerar números inteiros aleatórios dentro de um intervalo especificado. Vamos detalhar como ela funciona e como definir os limites inferior e superior.

np.random.randint()

Definição: Gera números inteiros aleatórios dentro de um intervalo especificado.

Sintaxe: np.random.randint(low, high=None, size=None, dtype=int)

low: Limite inferior (inclusive).

high: Limite superior (exclusive). Se não for especificado, o intervalo será de 0 até low.

size: Forma ou tamanho do array de saída. Pode ser um inteiro ou uma tupla de inteiros.

dtype: Tipo de dados do array de saída (por padrão, int).

Exemplo de Uso

Vamos ver alguns exemplos práticos de como usar np.random.randint():

Gerar um Único Número Inteiro Aleatório

import numpy as np

# Gera um número inteiro aleatório entre 10 (inclusive) e 20 (exclusive)

numero\_aleatorio = np.random.randint(10, 20)

print(numero\_aleatorio)

Gerar um Array de Números Inteiros Aleatórios

import numpy as np

# Gera um array 1D de 5 números inteiros aleatórios entre 10 (inclusive) e 20 (exclusive)

array\_aleatorio = np.random.randint(10, 20, size=5)

print(array\_aleatorio)

# Gera uma matriz 2x3 de números inteiros aleatórios entre 10 (inclusive) e 20 (exclusive)

matriz\_aleatoria = np.random.randint(10, 20, size=(2, 3))

print(matriz\_aleatoria)

Aplicações em Negócios

Simulações de Monte Carlo: Usado para gerar cenários aleatórios em modelos financeiros.

Teste de Algoritmos: Geração de dados de teste para validar algoritmos.

Modelagem de Incertezas: Criação de modelos que incorporam variabilidade e incerteza nos dados.

Resumo

np.random.randint(low, high, size): Gera números inteiros aleatórios entre low (inclusive) e high (exclusive), com a forma especificada por size.

Porque usar numpay e não listas?

## Arrays NumPy vs. Listas em Python

1. **Tipo de Dados Uniforme**
   * **NumPy**: Arrays NumPy são homogêneos, ou seja, todos os elementos devem ser do mesmo tipo de dados. Isso permite otimizações de memória e desempenho.
   * **Listas**: Listas podem conter elementos de diferentes tipos de dados, o que oferece flexibilidade, mas pode ser menos eficiente.
2. **Eficiência de Memória**
   * **NumPy**: Devido ao tipo de dados uniforme, arrays NumPy ocupam menos espaço na memória em comparação com listas.
   * **Listas**: Listas ocupam mais espaço na memória porque cada elemento é um objeto Python completo com metadados adicionais.
3. **Estruturas Multidimensionais**
   * **NumPy**: Suporta arrays multidimensionais (matrizes), o que é essencial para muitas operações em ciência de dados e computação científica.
   * **Listas**: Embora listas possam ser aninhadas para criar estruturas multidimensionais, elas não são otimizadas para operações matemáticas e podem ser menos eficientes.
4. **Comprimento Fixo e Alocação Estática**
   * **NumPy**: Arrays NumPy têm comprimento fixo e alocação estática, o que significa que o tamanho do array é definido na criação e não pode ser alterado. Isso permite otimizações de desempenho.
   * **Listas**: Listas são dinâmicas e podem crescer ou encolher conforme necessário, o que oferece flexibilidade, mas pode ser menos eficiente em termos de desempenho.
5. **Processamento de Dados Rápido**
   * **NumPy**: Arrays NumPy são otimizados para operações matemáticas e processamento de dados, resultando em desempenho significativamente melhor em comparação com listas.
   * **Listas**: Para operações matemáticas intensivas, listas são menos eficientes e geralmente requerem loops explícitos.
6. **Armazenamento de Tipos de Dados Primitivos**
   * **NumPy**: Tipos de dados primitivos (como inteiros e floats) são armazenados diretamente em arrays NumPy, o que melhora a eficiência.
   * **Listas**: Em listas, cada elemento é um objeto Python completo, o que adiciona overhead de memória.

**Exemplos Práticos**

**Criando um Array NumPy com Zeros**

import numpy as np

# Array 3x3 de zeros

array\_zeros = np.zeros((3, 3))

print(array\_zeros)

**Criando uma Sequência com np.arange()**

import numpy as np

# Sequência de 0 a 9

sequencia = np.arange(10)

print(sequencia)

**Gerando Valores Aleatórios com np.random.random()**

import numpy as np

# Matriz 2x3 de valores aleatórios entre 0 e 1

valores\_aleatorios = np.random.random((2, 3))

print(valores\_aleatorios)

**Resumo**

* **NumPy**: Ideal para computação científica e análise de dados devido à sua eficiência, suporte a estruturas multidimensionais e operações rápidas.
* **Listas**: Úteis para manipulação de dados simples e operações gerais, mas menos eficientes para tarefas matemáticas intensivas.

 arrays podem ter mais de duas dimensões! Arrays 3D são muito úteis em várias aplicações, como processamento de imagens, modelagem de dados tridimensionais e simulações científicas. Vamos explorar como criar e usar arrays 3D com NumPy.

**O que é um Array 3D?**

Um array 3D é uma estrutura de dados que possui três dimensões: altura, largura e profundidade. Você pode pensar nele como uma coleção de matrizes 2D empilhadas umas sobre as outras.

**Aplicações de Arrays 3D**

* **Processamento de Imagens**: Imagens coloridas podem ser representadas como arrays 3D, onde as dimensões representam altura, largura e canais de cor (RGB).
* **Simulações Científicas**: Modelagem de fenômenos físicos em três dimensões, como simulações de fluidos ou campos eletromagnéticos.
* **Dados Tridimensionais**: Representação de dados espaciais, como coordenadas em um espaço 3D.

**Como Criar um Array 3D em NumPy**

Vamos ver alguns exemplos de como criar e manipular arrays 3D em NumPy.

**Criando um Array 3D com np.zeros()**

import numpy as np

# Array 3D de zeros com dimensões 2x3x4 (2 matrizes 3x4)

array\_3d\_zeros = np.zeros((2, 3, 4))

print(array\_3d\_zeros)

**Criando um Array 3D com np.random.random()**

import numpy as np

# Array 3D de valores aleatórios entre 0 e 1 com dimensões 2x3x4

array\_3d\_random = np.random.random((2, 3, 4))

print(array\_3d\_random)

**Acessando Elementos em um Array 3D**

Você pode acessar elementos em um array 3D usando índices para cada dimensão.

import numpy as np

# Criando um array 3D de exemplo

array\_3d = np.array([[[1, 2, 3], [4, 5, 6], [7, 8, 9]],

[[10, 11, 12], [13, 14, 15], [16, 17, 18]]])

# Acessando o elemento na posição (1, 2, 1)

elemento = array\_3d[1, 2, 1]

print(elemento) # Saída: 17

**Resumo**

* **Arrays 3D**: Estruturas de dados com três dimensões, úteis para representar dados tridimensionais.
* **Criação**: Pode ser feito usando funções como np.zeros() e np.random.random().
* **Acesso**: Elementos podem ser acessados usando índices para cada dimensão.

Arrays as vezes são considerados vetores, matrizes ou tensores.

## Vetores

* **Definição**: Um vetor é um array unidimensional. Ele pode ser visto como uma lista de números.
* **Dimensão**: 1D (uma dimensão).
* **Exemplo**: [1, 2, 3, 4, 5]
* **Aplicações**: Representação de pontos em uma linha, séries temporais, características de dados em aprendizado de máquina.

## Matrizes

* **Definição**: Uma matriz é um array bidimensional. É uma tabela de números organizada em linhas e colunas.
* **Dimensão**: 2D (duas dimensões).
* **Exemplo**:
* [[1, 2, 3],
* [4, 5, 6],
* [7, 8, 9]]
* **Aplicações**: Álgebra linear, transformações geométricas, representação de imagens em tons de cinza.

## Tensores

* **Definição**: Um tensor é um array com mais de duas dimensões. Pode ser visto como uma generalização de matrizes para dimensões superiores.
* **Dimensão**: 3D ou mais (três ou mais dimensões).
* **Exemplo**: Um tensor 3D pode ser representado como uma coleção de matrizes 2D empilhadas.
* [[[1, 2, 3],
* [4, 5, 6],
* [7, 8, 9]],
* [[10, 11, 12],
* [13, 14, 15],
* [16, 17, 18]]]
* **Aplicações**: Processamento de imagens coloridas (RGB), redes neurais convolucionais, modelagem de dados multidimensionais.

**Exemplos Práticos com NumPy**

**Vetores**

import numpy as np

# Criando um vetor

vetor = np.array([1, 2, 3, 4, 5])

print(vetor)

**Matrizes**

import numpy as np

# Criando uma matriz 3x3

matriz = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print(matriz)

**Tensores**

import numpy as np

# Criando um tensor 3D com dimensões 2x3x3

tensor = np.array([[[1, 2, 3], [4, 5, 6], [7, 8, 9]],

[[10, 11, 12], [13, 14, 15], [16, 17, 18]]])

print(tensor)

**Resumo**

* **Vetores (1D)**: Usados para representar séries temporais, características de dados.
* **Matrizes (2D)**: Usadas em álgebra linear, transformações geométricas, imagens em tons de cinza.
* **Tensores (3D ou mais)**: Usados em processamento de imagens coloridas, redes neurais, modelagem de dados multidimensionais.

Vamos explorar as diferenças entre atributos, argumentos, funções e métodos:

# ### Introdução ao Pandas

Pandas é uma biblioteca poderosa para a manipulação e análise de dados em Python. Ela fornece estruturas de dados flexíveis e expressivas, como DataFrames, que facilitam o trabalho com dados tabulares.

## ### O Que é um DataFrame?

Um DataFrame é uma estrutura de dados bidimensional, semelhante a uma tabela em uma base de dados ou uma planilha do Excel. Ele pode ser criado a partir de diversas fontes, como dicionários, listas ou arquivos CSV.

## # Criando um DataFrame a partir de um dicionário

data = {'Nome': ['Ana', 'Bruno', 'Carlos'], 'Idade': [23, 35, 45]}

df = pd.DataFrame(data)

```

## ### Explorando um DataFrame

- \*\*`df.head()`\*\*: Mostra as primeiras 5 linhas do DataFrame.

- \*\*`df.info()`\*\*: Exibe informações sobre as colunas, incluindo tipos de dados e quantidade de valores não nulos.

- \*\*`df.shape`\*\*: Retorna uma tupla com o número de linhas e colunas do DataFrame.

- \*\*`df.describe()`\*\*: Fornece estatísticas básicas das colunas numéricas, como média, desvio padrão e quartis.

```python

print(df.head())

print(df.info())

print(df.shape)

print(df.describe())

```

## ### Componentes de um DataFrame

- \*\*Valores (`df.values`)\*\*: Os dados contidos no DataFrame.

- \*\*Colunas (`df.columns`)\*\*: Os nomes das colunas.

- \*\*Índice (`df.index`)\*\*: Os rótulos das linhas.

## ### Ordenação e Conjuntos no DataFrame

- \*\*`df.sort\_values(by=['coluna1', 'coluna2'], ascending=[True, False])`\*\*: Ordena o DataFrame com base em uma ou mais colunas.

## ### Subconjuntos e Condições Múltiplas

- \*\*`df[df['coluna'].isin([valores])]`\*\*: Filtra o DataFrame com base em uma lista de valores.

- \*\*Adicionar uma Coluna\*\*: `df['nova\_coluna'] = valores`.

## ### Múltiplas Manipulações

- \*\*`df.columns`\*\*: Acessa ou modifica os nomes das colunas.

- \*\*`df.index`\*\*: Acessa ou modifica o índice do DataFrame.

- \*\*Agregação de Dados (`df.agg`)\*\*: Aplica funções de agregação, como média, soma, mínimo e máximo.

## ### Métodos para Estatística Cumulativa

- \*\*`df.cumsum()`\*\*: Calcula a soma cumulativa.

- \*\*`df.cummax()`\*\*: Calcula o valor máximo cumulativo.

## ### Lidando com Valores Duplicados

- \*\*`df.drop\_duplicates(subset=['coluna'])`\*\*: Remove linhas duplicadas com base em uma coluna específica.

- \*\*`df['coluna'].value\_counts(normalize=True)`\*\*: Calcula a porcentagem de recorrência dos valores em uma coluna.

## Agrupamento de Dados

- \*\*`df.groupby('coluna')['outra\_coluna'].mean()`\*\*: Calcula a média de uma coluna agrupada por outra.

- \*\*`df.groupby(['coluna1', 'coluna2']).agg({'coluna3': ['min', 'max', 'sum']})`\*\*: Agrupa por múltiplas colunas e aplica várias funções de agregação.

## Tabelas Pivot

- \*\*`df.pivot\_table(values='coluna', index='indice', aggfunc=np.mean)`\*\*: Cria uma tabela dinâmica.

- \*\*`df.pivot\_table(values='coluna', index='indice', aggfunc=[np.mean, np.median])`\*\*: Aplica múltiplas funções de agregação.

## Índices e Fatiamento de um DataFrame

- \*\*Alterando um Índice\*\*: `df.set\_index('coluna')` ou `df.reset\_index()`.

- \*\*Fatiamento com `.loc`\*\*: `df.loc[condição]` para selecionar linhas e colunas por rótulo.

- \*\*Fatiamento com `.iloc`\*\*: `df.iloc[linhas, colunas]` para selecionar por posição numérica.

## Trabalhando com Datas

- \*\*Convertendo Strings em Datas\*\*: `df['data'] = pd.to\_datetime(df['data'])`.

- \*\*Ordenando por Data\*\*: `df.set\_index('data').sort\_index()` ou `df.sort\_values('data')`.

## Gráficos e Valores Faltosos

- \*\*Plotando Gráficos\*\*: `df.plot(kind='bar')`.

- \*\*Lidando com Valores Faltosos\*\*:

- \*\*`df.isna()`\*\*: Identifica valores nulos.

- \*\*`df.dropna()`\*\*: Remove linhas ou colunas com valores nulos.

- \*\*`df.fillna(valor)`\*\*: Substitui valores nulos por um valor específico.

```python

import matplotlib.pyplot as plt

# Exemplo de gráfico de barras

df['coluna'].plot(kind='bar')

plt.show()

# Identificando valores nulos

print(df.isna().sum())

# Removendo valores nulos

df = df.dropna()

# Substituindo valores nulos pela mediana

df = df.fillna(df.median(numeric\_only=True))

# Tratamento de Valores Ausentes

Valores ausentes podem causar problemas significativos durante a análise de dados. É essencial preencher (imputar) ou remover esses valores para garantir a integridade dos resultados. Métodos comuns incluem a imputação com a média, mediana ou moda, ou a remoção de linhas ou colunas inteiras que contêm valores ausentes.

# Codificação de Variáveis Categóricas

Variáveis categóricas precisam ser convertidas em uma forma numérica para serem utilizadas em algoritmos de machine learning. Uma técnica popular é o one-hot encoding, que cria colunas binárias para cada categoria, permitindo que os algoritmos interpretem corretamente os dados categóricos.

# Normalização ou Padronização

Para garantir que os valores numéricos estejam em uma escala comum, utilizamos técnicas de normalização ou padronização. A normalização escala os valores para um intervalo de 0 a 1, enquanto a padronização ajusta os dados para que tenham média 0 e desvio padrão 1. Isso é crucial para algoritmos que são sensíveis à escala dos dados.

# Conversão de Tipos de Dados

Garantir que os tipos de dados das colunas estão corretos é fundamental para a análise. Isso pode incluir a conversão de strings em datas, números inteiros, floats, entre outros. Ferramentas como pandas facilitam essas conversões com métodos como pd.to\_datetime() e pd.to\_numeric().

# drop\_duplicates()

Dados duplicados podem distorcer os resultados da análise. Identificar e remover essas entradas é essencial para manter a precisão. Em pandas, o método drop\_duplicates() é frequentemente utilizado para essa finalidade.

# groupby() e agg()

Agrupar dados com base em determinadas características e calcular estatísticas agregadas, como média, soma, mínimo e máximo, é uma prática comum. Isso ajuda a resumir e entender grandes conjuntos de dados. Em pandas, métodos como groupby() e agg() são utilizados para essas operações.

# Criação de Novas Variáveis (Feature Engineering)

Criar novas variáveis que possam ser mais informativas para a análise é uma técnica poderosa. Por exemplo, calcular a idade a partir da data de nascimento ou criar indicadores binários a partir de variáveis categóricas. Isso pode melhorar significativamente a performance dos modelos de machine learning.

# Divisão de Dados

Separar conjuntos de dados em subconjuntos com base em condições específicas é útil para análises segmentadas. Isso pode incluir a divisão de dados em conjuntos de treino e teste, ou a segmentação por categorias específicas.

rename()

Modificar os nomes das colunas para torná-los mais descritivos e compreensíveis facilita a interpretação dos dados. Em pandas, o método rename() é utilizado para renomear colunas de forma eficiente.

# Mudança do Formato do DataFrame

Reorganizar ou pivotar o DataFrame pode facilitar a análise ou a criação de visualizações. Ferramentas como pivot\_table() e melt() em pandas são úteis para essas transformações.

pivot\_table()

A função pivot\_table() do pandas é uma ferramenta poderosa para resumir e reorganizar dados em um DataFrame. Ela permite criar tabelas dinâmicas que agregam dados de maneira flexível, facilitando a análise e a visualização de informações.

O Que é uma Tabela Pivot?

Uma tabela pivot é uma tabela que permite reorganizar e resumir dados de um DataFrame, agrupando-os por uma ou mais colunas e aplicando funções de agregação, como média, soma, contagem, entre outras. Isso é especialmente útil para explorar grandes conjuntos de dados e extrair insights significativos.

Sintaxe Básica

A sintaxe básica da função pivot\_table() é a seguinte:

import pandas as pd

# Exemplo de DataFrame

data = {

'Produto': ['A', 'B', 'A', 'B'],

'Vendas': [100, 150, 200, 250],

'Região': ['Norte', 'Sul', 'Norte', 'Sul']

}

df = pd.DataFrame(data)

# Criando uma tabela pivot

pivot = df.pivot\_table(values='Vendas', index='Produto', columns='Região', aggfunc='sum')

print(pivot)

Parâmetros Principais

values: Coluna(s) que contêm os valores a serem agregados.

index: Coluna(s) a serem usadas como índice(s) na tabela pivot.

columns: Coluna(s) a serem usadas como colunas na tabela pivot.

aggfunc: Função de agregação a ser aplicada aos dados (por exemplo, sum, mean, count).

Exemplo Prático

Vamos criar uma tabela pivot que mostra a soma das vendas por produto e região:

pivot = df.pivot\_table(values='Vendas', index='Produto', columns='Região', aggfunc='sum')

print(pivot)

Isso resultará em uma tabela onde as linhas representam os produtos, as colunas representam as regiões e os valores são a soma das vendas.

Aplicações

Análise de Vendas: Resumir vendas por produto, região, período, etc.

Análise Financeira: Agregar dados financeiros por categoria, departamento, etc.

Análise de Marketing: Agrupar dados de campanhas por canal, segmento de clientes, etc.

melt()

A função melt() do pandas é usada para transformar ou "derreter" um DataFrame de formato largo para formato longo. Isso é útil quando você precisa reorganizar seus dados para análise ou visualização.

O Que é o Formato Longo?

No formato longo, cada variável é uma coluna e cada observação é uma linha. Isso é o oposto do formato largo, onde cada variável pode ser espalhada por várias colunas.

Sintaxe Básica

A sintaxe básica da função melt() é a seguinte:

import pandas as pd

# Exemplo de DataFrame

data = {

'Produto': ['A', 'B'],

'Vendas\_Jan': [100, 150],

'Vendas\_Fev': [200, 250]

}

df = pd.DataFrame(data)

# Transformando o DataFrame de formato largo para longo

melted = df.melt(id\_vars='Produto', var\_name='Mês', value\_name='Vendas')

print(melted)

Parâmetros Principais

id\_vars: Coluna(s) a serem mantidas como identificadores.

var\_name: Nome da coluna que conterá os nomes das variáveis.

value\_name: Nome da coluna que conterá os valores das variáveis.

Exemplo Prático

Vamos transformar o DataFrame de formato largo para longo:

melted = df.melt(id\_vars='Produto', var\_name='Mês', value\_name='Vendas')

print(melted)

Isso resultará em um DataFrame onde cada linha representa uma combinação de produto e mês, com as vendas correspondentes.

Aplicações

Preparação de Dados para Visualização: Transformar dados para formatos compatíveis com bibliotecas de visualização.

Análise Temporal: Reorganizar dados para facilitar a análise de séries temporais.

Transformação de Dados: Ajustar o formato dos dados para atender aos requisitos de diferentes ferramentas de análise.

# str.replace()

Expressões regulares são poderosas para remover caracteres especiais de nomes e telefones, garantindo que os dados estejam limpos e padronizados. Em pandas, o método str.replace() pode ser utilizado em conjunto com expressões regulares.

# get\_dummies e sep

O método get\_dummies() em pandas é utilizado para realizar one-hot encoding de variáveis categóricas. O parâmetro sep pode ser utilizado para especificar o separador em operações de leitura de arquivos, como pd.read\_csv().