Programação para Ciência de Dados

NumPy Avançado - Manipulação e Álgebra Linear

Arthur Casals

16 de Outubro de 2025

Avisos

- Primeira entrega quinzenal: 17/10 (sexta-feira)
- Cobre aulas 1-4 (completo)

Agenda

- Manipulação Avançada de Shape
- Operações Matriciais e Produto Escalar
- Álgebra Linear com NumPy
- Concatenação e Stacking
- I/O de Arrays

Recapitulação Aula 03

O que já sabemos:

- Criação de arrays (zeros, ones, arange, linspace)
- Indexação e slicing avançado
- Boolean indexing e fancy indexing
- Broadcasting automático
- Operações vetorizadas e ufuncs
- Funções de agregação com axis

Hoje vamos aprofundar:

- Manipulação complexa de dimensões
- Operações matriciais e álgebra linear
- Combinação e separação de arrays

Bloco 1

Manipulação Avançada de Shape

Por que Manipular Shape?

Necessidade em Ciência de Dados:

- **Machine Learning:** Preparar dados para modelos
- **Deep Learning:** Ajustar dimensões para redes neurais
- **Visualização:** Transformar dados para plotagem
- **Performance:** Otimizar layout de memória

Operações principais:

- `reshape()`: Mudar dimensões mantendo dados
- `transpose()`: Trocar eixos
- `flatten()`: Achatar para 1D (cópia)
- `ravel()`: Achatar para 1D (view quando possível)
- `squeeze()`: Remover dimensões de tamanho 1
- `expand_dims()`: Adicionar dimensões

reshape(): Mudando Dimensões

O que faz: Reorganiza array em nova forma sem copiar dados

</> Python

```python
# Array 1D com 12 elementos
arr = np.arange(12)
print(arr)
# [ 0  1  2  3  4  5  6  7  8  9 10 11]
print(f"Shape: {arr.shape}")  # (12,)

# Reshape para 2D
arr_2d = arr.reshape(3, 4)
print(arr_2d)
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]
print(f"Shape: {arr_2d.shape}")  # (3, 4)
```

reshape(): Dimensão Automática

Usar -1 para calcular dimensão automaticamente:

</> Python

```python
arr = np.arange(12)

# NumPy calcula automaticamente
arr_2d = arr.reshape(3, -1)  # 3 linhas, ? colunas
print(f"Shape: {arr_2d.shape}")  # (3, 4)

arr_2d = arr.reshape(-1, 4)  # ? linhas, 4 colunas
print(f"Shape: {arr_2d.shape}")  # (3, 4)

# Para 3D
arr_3d = arr.reshape(2, 2, -1)
print(f"Shape: {arr_3d.shape}")  # (2, 2, 3)

```

reshape(): Regras Importantes

Regra fundamental: Número total de elementos deve ser igual

</> Python

```python
arr = np.arange(12)  # 12 elementos

# OK: 3 * 4 = 12
arr.reshape(3, 4)

# OK: 2 * 6 = 12
arr.reshape(2, 6)

# ERRO: 3 * 5 = 15 != 12
# arr.reshape(3, 5)  # ValueError!

# OK: 2 * 2 * 3 = 12
arr.reshape(2, 2, 3)

```

reshape() vs resize()

reshape():

- Retorna view (quando possível)
- Não modifica original
- Deve ter mesmo número de elementos

Python

```python
arr = np.arange(6)
arr_2d = arr.reshape(2, 3)
# Original: [0 1 2 3 4 5]
# Novo: [[0 1 2]
#        [3 4 5]]

```

resize():

- Modifica in-place
- Pode adicionar/remover elementos
- Preenche com zeros se necessário

Python

```python
arr = np.arange(6)
arr.resize(2, 4)
print(arr)
# [[0 1 2 3]
#  [4 5 0 0]]
# Adicionou zeros!

```

transpose(): Trocar Eixos

Operação Essencial em Álgebra Linear

Python

```python
# Matriz 2x3
matriz = np.array([[1, 2, 3],
                   [4, 5, 6]])
print(f"Original shape: {matriz.shape}")  # (2, 3)

# Transpor (linhas  colunas)
transposta = matriz.T
print(f"Transposta shape: {transposta.shape}")  # (3, 2)
print(transposta)
# [[1 4]
#  [2 5]
#  [3 6]]

```

transpose(): Arrays Multidimensionais

</> Python

```python
# Array 3D
arr_3d = np.arange(24).reshape(2, 3, 4)
print(f"Original: {arr_3d.shape}")  # (2, 3, 4)

# Transpor padrão: inverte todos os eixos
transp = arr_3d.T
print(f"Transposto: {transp.shape}")  # (4, 3, 2)

# Especificar ordem dos eixos
transp = np.transpose(arr_3d, (2, 0, 1))
print(f"Personalizado: {transp.shape}")  # (4, 2, 3)
# Eixo 2 -> posicao 0
# Eixo 0 -> posicao 1
# Eixo 1 -> posicao 2

```

swapaxes(): Trocar Dois Eixos

</> Python

```python
# Array 3D
arr = np.arange(24).reshape(2, 3, 4)
print(f"Original: {arr.shape}")  # (2, 3, 4)

# Trocar eixo 0 com eixo 2
trocado = np.swapaxes(arr, 0, 2)
print(f"Trocado: {trocado.shape}")  # (4, 3, 2)

# Equivalente a transpose com ordem especifica
equiv = np.transpose(arr, (2, 1, 0))
print(np.array_equal(trocado, equiv))  # True

```

flatten() vs ravel()

Achatar Arrays

flatten():

- Sempre retorna **cópia**
- Modificações não afetam original
- Mais seguro

Python

```python
matriz = np.array([[1, 2],
                   [3, 4]])
flat = matriz.flatten()
flat[0] = 999
print(matriz[0, 0])
# 1 - nao modificou!
```

ravel():

- Retorna **view** quando possível
- Modificações podem afetar original
- Mais eficiente

Python

```python
matriz = np.array([[1, 2],
                   [3, 4]])
rav = matriz.ravel()
rav[0] = 999
print(matriz[0, 0])
# 999 - modificou!
```

flatten() vs ravel()

⚠ Atenção

Use `flatten()` se precisar garantir independência

squeeze(): Remover Dimensões Unitárias

</> Python

```python
# Array com dimensoes de tamanho 1
arr = np.array([[[1], [2], [3]]])
print(f"Original: {arr.shape}")  # (1, 3, 1)

# Remover todas dimensoes de tamanho 1
squeezed = np.squeeze(arr)
print(f"Squeezed: {squeezed.shape}")  # (3,)
print(squeezed)  # [1 2 3]

# Remover dimensão específica
squeezed = np.squeeze(arr, axis=0)
print(f"Squeeze axis 0: {squeezed.shape}")  # (3, 1)
squeezed = np.squeeze(arr, axis=2)
print(f"Squeeze axis 2: {squeezed.shape}")  # (1, 3)
```

expand_dims(): Adicionar Dimensões

</> Python

```python
# Array 1D
arr = np.array([1, 2, 3])
print(f"Original: {arr.shape}")  # (3,)
# Adicionar dimensao no inicio
expanded = np.expand_dims(arr, axis=0)
print(f"Axis 0: {expanded.shape}")  # (1, 3)
print(expanded)  # [[1 2 3]]
# Adicionar dimensao no final
expanded = np.expand_dims(arr, axis=1)
print(f"Axis 1: {expanded.shape}")  # (3, 1)
print(expanded)
# [[1]
#  [2]
#  [3]]
```

newaxis: Alternativa para expand_dims

Python

```python
arr = np.array([1, 2, 3])
print(f"Original: {arr.shape}")  # (3,)

# Adicionar dimensao com newaxis
expanded = arr[np.newaxis, :]
print(f"newaxis inicio: {expanded.shape}")  # (1, 3)

expanded = arr[:, np.newaxis]
print(f"newaxis final: {expanded.shape}")  # (3, 1)

# Multiplas dimensoes
expanded = arr[np.newaxis, :, np.newaxis]
print(f"Multiplas: {expanded.shape}")  # (1, 3, 1)

```

expand_dims() vs newaxis

expand_dims(array, axis):

- É uma **função**
- Retorna **view** quando possível
- Melhor uso: eixos dinâmicos

np.newaxis:

- É um **atalho** para `None`
- Posições dos eixos são definidas durante a indexação
- Pode ser repetido para múltiplos eixos

Exemplo Prático: Preparar Dados para ML

Python

```python
# Dados: 1000 amostras, 28x28 pixels (imagens)
imagens = np.random.randint(0, 256, (1000, 28, 28))
print(f"Shape original: {imagens.shape}")

# Achatar cada imagem para vetor
# Manter 1000 amostras, mas 784 atributos (28*28)
X = imagens.reshape(1000, -1)
print(f"Shape para ML: {X.shape}")  # (1000, 784)

# Labels (1D)
y = np.random.randint(0, 10, 1000)
print(f"Labels shape: {y.shape}")  # (1000,)

```

Exemplo Prático: Broadcasting com Shape

Python

```python
# Dados: batch de 100 imagens RGB 32x32
batch = np.random.randint(0, 256, (100, 32, 32, 3))
print(f"Batch shape: {batch.shape}")

# Normalizar: subtrair media e dividir por desvio por canal (R, G, B)
mean = np.array([123.68, 116.78, 103.94])
std = np.array([58.39, 57.12, 57.38])

# Reshape para broadcasting: (1, 1, 1, 3)
mean = mean.reshape(1, 1, 1, 3)
std = std.reshape(1, 1, 1, 3)

# Normalizar
batch_norm = (batch - mean) / std
print(f"Normalizado: {batch_norm.shape}")
```

moveaxis(): Mover Eixos

</> Python

```python
# Array de imagens: (batch, height, width, channels)
imagens = np.random.rand(10, 224, 224, 3)
print(f"Original: {imagens.shape}")  # (10, 224, 224, 3)

# Mover canais para segundo eixo
# Formato para PyTorch: (batch, channels, height, width)
imagens_torch = np.moveaxis(imagens, 3, 1)
print(f"PyTorch: {imagens_torch.shape}")  # (10, 3, 224, 224)

# Equivalente com transpose
equiv = np.transpose(imagens, (0, 3, 1, 2))
print(np.array_equal(imagens_torch, equiv))  # True

```

atleast_xd(): Garantir Dimensões Mínimas

</> Python

```python
# Escalar
arr_0d = np.array(5)
arr_1d = np.atleast_1d(arr_0d)
print(f"0D -> 1D: {arr_1d.shape}")  # (1,)

# 1D para 2D
arr = np.array([1, 2, 3])
arr_2d = np.atleast_2d(arr)
print(f"1D -> 2D: {arr_2d.shape}")  # (1, 3)

# 1D para 3D
arr_3d = np.atleast_3d(arr)
print(f"1D -> 3D: {arr_3d.shape}")  # (1, 3, 1)

```

Exemplo Completo: Pipeline de Transformação

</> Python

```python
# Dados brutos: lista de arrays de tamanhos diferentes
dados_brutos = [
    np.array([1, 2, 3]),
    np.array([4, 5]),
    np.array([6, 7, 8, 9])
]

# Normalizar para mesmo tamanho (padding)
max_len = max(len(arr) for arr in dados_brutos)
dados_padded = []
for arr in dados_brutos:
    padded = np.pad(arr, (0, max_len - len(arr)))
    dados_padded.append(padded)

```

Exemplo Completo: Pipeline (cont.)

Python

```python
# Empilhar em matriz
matriz = np.array(dados_padded)
print(f"Matriz: {matriz.shape}")  # (3, 4)

# Adicionar dimensao para batch
batch = np.expand_dims(matriz, axis=0)
print(f"Batch: {batch.shape}")  # (1, 3, 4)

# Transpor para formato (features, samples, batch)
final = np.transpose(batch, (2, 1, 0))
print(f"Final: {final.shape}")  # (4, 3, 1)

```

Bloco 2

Operações Matriciais e Álgebra Linear

Operações Matriciais

Fundamentais para Data Science

Por que importam:

- **Machine Learning:** Regressão linear, PCA, SVD
- **Deep Learning:** Camadas totalmente conectadas
- **Estatística:** Correlação, covariância
- **Processamento de Sinais:** Transformadas
- **Grafos:** Matrizes de adjacência

Operações principais:

- Produto escalar (dot product)
- Produto matricial (matrix multiplication)
- Produto elemento-a-elemento (Hadamard)
- Produto externo (outer product)
- Produto interno (inner product)

Tipos de Produtos

Entendendo as Diferenças

Operação	Símbolo	Resultado
Elemento-a-elemento	`*`	Mesmo shape
Produto escalar	`np.dot()`	Escalar
Produto matricial	`@` ou `np.matmul()`	Matriz
Produto externo	`np.outer()`	Matriz
Produto interno	`np.inner()`	Escalar/Matriz

Nota Importante

Confusão comum: `*` é elemento-a-elemento, não matricial!

Produto Elemento-a-Elemento (Hadamard)

</> Python

```python
a = np.array([[1, 2],
              [3, 4]])
b = np.array([[5, 6],
              [7, 8]])

# Multiplicacao elemento-a-elemento
c = a * b
print(c)
# [[ 5 12]
#  [21 32]]

# Equivalente a
c = np.multiply(a, b)

```

Produto Escalar (Dot Product)

Vetores Escalar

</> Python

```python
# Dois vetores
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Produto escalar
dot = np.dot(a, b)
print(dot)  # 32
# Calculo: 1*4 + 2*5 + 3*6 = 4 + 10 + 18 = 32

# Formas equivalentes
dot = a @ b  # Operador @
dot = np.sum(a * b)  # Manual
dot = (a * b).sum()  # Manual

```

Produto Matricial

Multiplicação de Matrizes

</> Python

```python
# Matriz (2, 3) @ Matriz (3, 2) -> Matriz (2, 2)
A = np.array([[1, 2, 3],
              [4, 5, 6]])  # (2, 3)
B = np.array([[7, 8],
              [9, 10],
              [11, 12]])   # (3, 2)

# Produto matricial
C = A @ B  # Operador @ (Python 3.5+)
print(C)
# [[ 58  64]
#  [139 154]]
print(C.shape)  # (2, 2)
```

Produto Matricial: Dimensões

Regra fundamental: $(m, n) \times (n, p) = (m, p)$

Python

```python
# A dimensao interna (n) deve ser igual
A = np.random.rand(3, 4)  # (3, 4)
B = np.random.rand(4, 5)  # (4, 5)
C = A @ B  # OK: (3, 5)

# ERRO: dimensoes incompativeis
A = np.random.rand(3, 4)  # (3, 4)
B = np.random.rand(5, 2)  # (5, 2)
# C = A @ B  # ValueError: shapes (3,4) and (5,2) not aligned

```

Produto Matricial: Dimensões

⚠ Atenção

Sempre verifique compatibilidade de dimensões antes de multiplicar

np.dot() vs @ vs np.matmul()

</> Python

```python
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

# Tres formas equivalentes para 2D
C1 = np.dot(A, B)
C2 = A @ B
C3 = np.matmul(A, B)

print(np.array_equal(C1, C2))  # True
print(np.array_equal(C2, C3))  # True

# Para arrays 2D, sao identicos
# Para 3D+, comportamento difere ligeiramente

```

np.dot() vs @ vs np.matmul()

np.dot():

- Para arrays 1D: produto escalar (resultado escalar)
- Para arrays 2D: multiplicação matricial tradicional
- Para N-D ($N \geq 3$): soma sobre o último eixo do primeiro array e penúltimo do segundo (pode não conservar a estrutura de lote)
- Não realiza ”batch matrix multiplication”de forma natural

@, np.matmul():

- Para 1D e 2D: igual à multiplicação matricial
- Para N-D ($N \geq 3$): trata os dois últimos eixos como matrizes, realizando multiplicação matricial para cada ”lote”(forma batelada)
- Dimensões anteriores aos dois últimos são interpretadas como lotes/batches, resultado mantém essa estrutura
- Comportamento mais intuitivo para aplicações em aprendizado de máquina

Nota Importante

Prefira `@` - é mais legível e moderno (PEP 465)

Produto Externo (Outer Product)

Python

```python
# Dois vetores
a = np.array([1, 2, 3])
b = np.array([4, 5])

# Produto externo
outer = np.outer(a, b)
print(outer)
# [[ 4  5]
#  [ 8 10]
#  [12 15]]
print(outer.shape)  # (3, 2)

# Calculo manual equivalente
outer_manual = a[:, np.newaxis] * b[np.newaxis, :]
print(np.array_equal(outer, outer_manual))  # True
```

Produto Interno (Inner Product)

</> Python

```python
# Para vetores 1D: igual ao dot product
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

inner = np.inner(a, b)
print(inner)  # 32
# Para matrizes 2D: produto das ultimas dimensoes
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

inner = np.inner(A, B)
print(inner)
# [[17 23]
#  [39 53]]
# Diferente de A @ B !
```

Exemplo: Regressão Linear

</> Python

```python
# Dados: y = 2x + 3 + ruido
np.random.seed(42)
X = np.random.rand(100, 1) * 10
y = 2 * X + 3 + np.random.randn(100, 1)

# Adicionar coluna de 1s para intercepto
X_b = np.c_[np.ones((100, 1)), X]

# Calcular parametros: theta = (X^T X)^-1 X^T y
# Usando algebra linear
theta = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ y
print(f"Intercepto: {theta[0, 0]:.2f}")  # ~3
print(f"Coeficiente: {theta[1, 0]:.2f}")  # ~2
```

Álgebra Linear com NumPy: `np.linalg`

Funções principais:

Função	Descrição
`inv()`	Inversa de matriz
`det()`	Determinante
`eig()`	Autovalores e autovetores
`svd()`	Decomposição SVD
`qr()`	Decomposição QR
`cholesky()`	Decomposição Cholesky
`solve()`	Resolver sistema linear
`lstsq()`	Mínimos quadrados
`norm()`	Norma vetorial/matricial
`matrix_rank()`	Posto da matriz

Inversa de Matriz

Python

```python
# Matriz quadrada
A = np.array([[1, 2],
              [3, 4]])

# Calcular inversa
A_inv = np.linalg.inv(A)
print(A_inv)
# [[-2.   1. ]
#  [ 1.5 -0.5]]

# Verificar: A @ A_inv = I
I = A @ A_inv
print(I)
# [[1. 0.]
#  [0. 1.]]
```

Determinante

Python

```python
A = np.array([[1, 2],
              [3, 4]])

# Calcular determinante
det = np.linalg.det(A)
print(f"Determinante: {det}")  # -2.0

# Verificar se matriz e singular
B = np.array([[1, 2],
              [2, 4]])  # Linhas linearmente dependentes
det_B = np.linalg.det(B)
print(f"Det B: {det_B}")  # ~0 (singular)

# Matriz singular nao tem inversa
# np.linalg.inv(B)  # LinAlgError!
```

Resolver Sistema Linear

</> Python

```python
# Sistema: 2x + y = 8
#          x - y = 1
A = np.array([[2, 1],
              [1, -1]])
b = np.array([8, 1])

# Resolver Ax = b
x = np.linalg.solve(A, b)
print(x)  # [3. 2.]
# x=3, y=2

# Verificar solucao
print(A @ x)  # [8. 1.]
print(np.allclose(A @ x, b))  # True

```

Resolver Sistema Linear

Nota Importante

Mais eficiente que calcular inversa: $x = A^{-1}b$

Autovalores e Autovetores

Python

```python
# Matriz simetrica
A = np.array([[4, 2],
              [2, 3]])
# Calcular autovalores e autovetores
eigenvalues, eigenvectors = np.linalg.eig(A)
print(f"Autovalores: {eigenvalues}")
# [5.37 1.63]
print(f"Autovetores: {eigenvectors}")
# [[ 0.82  0.57]
#  [ 0.57 -0.82]]
# Verificar: A @ v = lambda * v
v = eigenvectors[:, 0]
lam = eigenvalues[0]
print(np.allclose(A @ v, lam * v))  # True

```

Decomposição SVD (Singular Value Decomposition)

</> Python

```python
# Matriz qualquer
A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9],
              [10, 11, 12]])
# Decompor: A = U @ S @ Vt
U, s, Vt = np.linalg.svd(A, full_matrices=False)
print(f"U shape: {U.shape}")    # (4, 3)
print(f"s shape: {s.shape}")    # (3,)
print(f"Vt shape: {Vt.shape}")  # (3, 3)
# Reconstruir matriz
S = np.diag(s)
A_reconstructed = U @ S @ Vt
print(np.allclose(A, A_reconstructed))  # True
```

Normas Vetoriais

</> Python

```python
v = np.array([3, 4])
# Norma L2 (Euclidiana)
norm_2 = np.linalg.norm(v)
print(f"L2: {norm_2}")  # 5.0
# Norma L1 (Manhattan)
norm_1 = np.linalg.norm(v, ord=1)
print(f"L1: {norm_1}")  # 7.0
# Norma infinita (maximo absoluto)
norm_inf = np.linalg.norm(v, ord=np.inf)
print(f"Linf: {norm_inf}")  # 4.0
# Manual
manual = np.sqrt(np.sum(v**2))
print(np.allclose(norm_2, manual))  # True
```

Posto da Matriz (Matrix Rank)

</> Python

```python
# Matriz posto completo
A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 10]])
rank = np.linalg.matrix_rank(A)
print(f"Posto: {rank}")  # 3

# Matriz rank-deficient
B = np.array([[1, 2, 3],
              [2, 4, 6],  # 2x primeira linha
              [3, 6, 9]])  # 3x primeira linha
rank = np.linalg.matrix_rank(B)
print(f"Posto: {rank}")  # 1

```

Exemplo: PCA Manual

Python

```python
# Dados: 100 amostras, 5 features
np.random.seed(42)
X = np.random.randn(100, 5)
# Centralizar dados
X_centered = X - X.mean(axis=0)
# Matriz de covariancia
cov = (X_centered.T @ X_centered) / (X.shape[0] - 1)
# Autovalores e autovetores
eigenvalues, eigenvectors = np.linalg.eig(cov)
# Ordenar por autovalor (maior -> menor)
idx = eigenvalues.argsort()[::-1]
eigenvalues = eigenvalues[idx]
eigenvectors = eigenvectors[:, idx]

```

Exemplo: PCA Manual (cont.)

</> Python

```python
# Projetar em primeiras 2 componentes
n_components = 2
W = eigenvectors[:, :n_components]
X_pca = X_centered @ W

print(f"Original: {X.shape}")      # (100, 5)
print(f"PCA: {X_pca.shape}")       # (100, 2)
print(f"Variancia explicada:")
variance_explained = eigenvalues / eigenvalues.sum()
for i, var in enumerate(variance_explained[:2]):
    print(f"  PC{i+1}: {var*100:.1f}%")

```

Bloco 3

Concatenação e Stacking de Arrays

Combinando Arrays

Diferentes Abordagens

Quando precisamos combinar:

- Juntar datasets de diferentes fontes
- Adicionar features ou amostras
- Construir batches para treinamento
- Criar tensores maiores

Principais funções:

- `concatenate()`: Juntar ao longo de eixo existente
- `stack()`: Empilhar criando novo eixo
- `vstack()`: Empilhar verticalmente (linhas)
- `hstack()`: Empilhar horizontalmente (colunas)
- `dstack()`: Empilhar em profundidade
- `column_stack()`: Empilhar como colunas
- `row_stack()`: Empilhar como linhas

concatenate(): Juntar Arrays

Python

```python
a = np.array([[1, 2],
              [3, 4]])
b = np.array([[5, 6],
              [7, 8]])

# Concatenar ao longo do axis=0 (empilhar linhas)
c = np.concatenate([a, b], axis=0)
print(c)
# [[1 2]
#  [3 4]
#  [5 6]
#  [7 8]]
print(c.shape)  # (4, 2)

```

concatenate(): Diferentes Eixos

Python

```python
a = np.array([[1, 2],
              [3, 4]])
b = np.array([[5, 6],
              [7, 8]])

# Concatenar ao longo do axis=1 (juntar colunas)
c = np.concatenate([a, b], axis=1)
print(c)
# [[1 2 5 6]
#  [3 4 7 8]]
print(c.shape)  # (2, 4)

# Arrays devem ter dimensoes compativeis!

```

vstack() e hstack()

Python

```python
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# vstack: empilhar verticalmente
v = np.vstack([a, b])
print(v)
# [[1 2 3]
#  [4 5 6]]
print(v.shape)  # (2, 3)

# hstack: empilhar horizontalmente
h = np.hstack([a, b])
print(h)  # [1 2 3 4 5 6]
print(h.shape)  # (6,)

```

stack(): Criar Nova Dimensão

Python

```python
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
# stack ao longo de novo eixo 0
s = np.stack([a, b], axis=0)
print(s)
# [[1 2 3]
#  [4 5 6]]
print(s.shape)  # (2, 3)
# stack ao longo de novo eixo 1
s = np.stack([a, b], axis=1)
print(s)
# [[1 4]
#  [2 5]
#  [3 6]]
print(s.shape)  # (3, 2)
```

Diferença: concatenate vs stack

concatenate:

- Usa eixo existente
- Não cria nova dimensão
- Arrays podem ter tamanhos diferentes no axis

Python

```python
a = np.array([1, 2])
b = np.array([3, 4, 5])
# Diferentes tamanhos OK
c = np.concatenate([a, b])
# [1 2 3 4 5]

```

stack:

- Cria novo eixo
- Aumenta dimensionalidade
- Arrays devem ter shapes idênticos

Python

```python
a = np.array([1, 2])
b = np.array([3, 4])
# Mesmo shape necessario
s = np.stack([a, b])
# [[1 2]
#  [3 4]]

```

column_stack() e row_stack()

</> Python

```python
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
# column_stack: tratar 1D como colunas
cs = np.column_stack([a, b])
print(cs)
# [[1 4]
#  [2 5]
#  [3 6]]
print(cs.shape)  # (3, 2)
# row_stack: equivalente a vstack
rs = np.row_stack([a, b])
print(rs)
# [[1 2 3]
#  [4 5 6]]
print(rs.shape)  # (2, 3)
```

dstack(): Empilhar em Profundidade

</> Python

```python
# Arrays 2D
a = np.array([[1, 2],
              [3, 4]])
b = np.array([[5, 6],
              [7, 8]])

# dstack: empilhar ao longo do 3o eixo
d = np.dstack([a, b])
print(d.shape)  # (2, 2, 2)
print(d)
# [[[1 5]
#   [2 6]]
#  [[3 7]
#   [4 8]]]

```

dstack(): Empilhar em Profundidade

Nota Importante

Útil para combinar canais de imagem (R, G, B)

Exemplo: Combinar Atributos

Python

```python
# Features separadas
idades = np.array([25, 30, 35, 40])
salarios = np.array([30000, 45000, 55000, 60000])
anos_exp = np.array([2, 5, 8, 12])

# Combinar em matriz de features
# Cada coluna = uma feature
X = np.column_stack([idades, salarios, anos_exp])
print(X)
# [[   25 30000     2]
#  [   30 45000     5]
#  [   35 55000     8]
#  [   40 60000    12]]
print(X.shape)  # (4, 3)

```

Exemplo: Criar Batches

Python

```python
# Imagens individuais (28x28)
img1 = np.random.rand(28, 28)
img2 = np.random.rand(28, 28)
img3 = np.random.rand(28, 28)

# Criar batch de imagens
batch = np.stack([img1, img2, img3], axis=0)
print(f"Batch shape: {batch.shape}")  # (3, 28, 28)

# Adicionar dimensao de canal
batch = np.expand_dims(batch, axis=-1)
print(f"Com canal: {batch.shape}")  # (3, 28, 28, 1)

```

Separar Arrays: split()

</> Python

```python
# Array grande
arr = np.arange(12)

# Dividir em 3 partes iguais
partes = np.split(arr, 3)
print(f"Numero de partes: {len(partes)}")
for i, parte in enumerate(partes):
    print(f"Parte {i}: {parte}")
# Parte 0: [0 1 2 3]
# Parte 1: [4 5 6 7]
# Parte 2: [8 9 10 11]

# Dividir em indices especificos
partes = np.split(arr, [3, 7])
# Divide em arr[:3], arr[3:7], arr[7:]
```

vsplit() e hsplit()

Python

```python
matriz = np.arange(12).reshape(4, 3)
print(matriz)
# [[ 0  1  2]
#  [ 3  4  5]
#  [ 6  7  8]
#  [ 9 10 11]]

# Dividir verticalmente (por linhas)
top, bottom = np.vsplit(matriz, 2)
print(f"Top shape: {top.shape}")      # (2, 3)
print(f"Bottom shape: {bottom.shape}")  # (2, 3)

# Dividir horizontalmente (por colunas)
left, middle, right = np.hsplit(matriz, 3)
print(f"Left shape: {left.shape}")  # (4, 1)
```

Exemplo: Train/Test Split

</> Python

```python
# Dataset: 1000 amostras
X = np.random.randn(1000, 10)
y = np.random.randint(0, 2, 1000)

# Dividir 80/20
split_idx = int(0.8 * len(X))

X_train = X[:split_idx]
X_test = X[split_idx:]
y_train = y[:split_idx]
y_test = y[split_idx:]

print(f"Train: {X_train.shape}")  # (800, 10)
print(f"Test: {X_test.shape}")    # (200, 10)
```

r_ e c_: Atalhos Convenientes

Python

```python
# np.r_: concatenar por linhas
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
result = np.r_[a, b]
print(result)  # [1 2 3 4 5 6]
# np.c_: concatenar por colunas
result = np.c_[a, b]
print(result)
# [[1 4]
#  [2 5]
#  [3 6]]
# Pode misturar com escalares e slices
result = np.r_[1, 2, 3:7, a]
print(result)  # [1 2 3 4 5 6 1 2 3]
```

Bloco 4

I/O: Salvar e Carregar Arrays

Persistência de Dados

Por que Salvar Arrays?

Casos de uso:

- **Checkpoints:** Salvar progresso de processamento
- **Cache:** Evitar recomputação de dados processados
- **Compartilhamento:** Transferir dados entre sistemas
- **Reprodutibilidade:** Manter datasets exatos
- **Armazenamento:** Arquivar resultados

Formatos disponíveis:

- `.npy`: Binário NumPy (um array)
- `.npz`: Binário NumPy comprimido (múltiplos arrays)
- `.txt`: Texto (legível, mas ineficiente)
- `.csv`: CSV (compatível com outras ferramentas)

save() e load(): Formato .npy

Python

```python
# Criar array
arr = np.random.randn(1000, 100)

# Salvar em formato binario .npy
np.save('dados.npy', arr)
print(f"Array salvo: {arr.shape}")

# Carregar
arr_loaded = np.load('dados.npy')
print(f"Array carregado: {arr_loaded.shape}")

# Verificar igualdade
print(np.array_equal(arr, arr_loaded))  # True

```

savez() e savez_compressed(): Múltiplos Arrays

</> Python

```python
# Multiplos arrays
X_train = np.random.randn(800, 10)
y_train = np.random.randint(0, 2, 800)
X_test = np.random.randn(200, 10)
y_test = np.random.randint(0, 2, 200)
# Salvar todos em um arquivo .npz
np.savez('dataset.npz',
         X_train=X_train,
         y_train=y_train,
         X_test=X_test,
         y_test=y_test)
# Carregar
data = np.load('dataset.npz')
print(data.files)  # ['X_train', 'y_train', 'X_test', 'y_test']
X_train_loaded = data['X_train']
```

Compressão com savez_compressed()

Python

```python
# Array grande
arr = np.random.randn(10000, 1000)
# Salvar sem compressao
np.savez('sem_comp.npz', arr=arr)

# Salvar com compressao
np.savez_compressed('com_comp.npz', arr=arr)

# Comparar tamanhos
import os
size_sem = os.path.getsize('sem_comp.npz')
size_com = os.path.getsize('com_comp.npz')
print(f"Sem compressao: {size_sem/1024/1024:.1f} MB")
print(f"Com compressao: {size_com/1024/1024:.1f} MB")
print(f"Reducao: {(1-size_com/size_sem)*100:.1f}%")
```

savetxt() e loadtxt(): Formato Texto

Python

```python
# Array pequeno
arr = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
# Salvar como texto
np.savetxt('dados.txt', arr, fmt='%d', delimiter=',')
# Arquivo gerado:
# 1,2,3
# 4,5,6
# 7,8,9

# Carregar
arr_loaded = np.loadtxt('dados.txt', delimiter=',')
print(arr_loaded)
```

savetxt() e loadtxt(): Formato Texto

⚠ Atenção

Formato texto: legível mas lento e usa mais espaço

genfromtxt(): Carregar Dados Complexos

Python

```python
# Arquivo CSV com cabecalho e valores faltantes
# nome,idade,salario
# Ana,25,30000
# Bruno,,45000
# Carlos,35,

# Carregar com genfromtxt
data = np.genfromtxt('dados.csv',
                     delimiter=',',
                     skip_header=1,
                     filling_values=0,
                     dtype=None,
                     encoding='utf-8')
print(data)
# Lida automaticamente com valores faltantes
```

Exemplo: Salvar Modelo Treinado

</> Python

```python
# Simular treinamento de modelo
weights = np.random.randn(100, 50)
biases = np.random.randn(50)
history = {
    'loss': [0.5, 0.4, 0.3, 0.2],
    'accuracy': [0.7, 0.8, 0.85, 0.9]
}

# Salvar modelo
np.savez_compressed('modelo.npz',
                    weights=weights,
                    biases=biases,
                    loss=history['loss'],
                    accuracy=history['accuracy'])
print("Modelo salvo com sucesso!")
```

Exemplo: Carregar e Continuar Treinamento

Python

```python
# Carregar modelo salvo
checkpoint = np.load('modelo.npz')

# Recuperar parametros
weights = checkpoint['weights']
biases = checkpoint['biases']
loss_history = checkpoint['loss'].tolist()
acc_history = checkpoint['accuracy'].tolist()

print(f"Modelo carregado!")
print(f"Shape dos pesos: {weights.shape}")
print(f"Ultima loss: {loss_history[-1]}")
print(f"Ultima accuracy: {acc_history[-1]}")
# Continuar treinamento...
```

memmap(): Arrays em Disco

Python

```python
# Criar memmap (array mapeado em disco)
# Usa disco como se fosse RAM
fp = np.memmap('huge_data.dat',
               dtype='float32',
               mode='w+',
               shape=(100000, 10000))

# Preencher em blocos (nao carrega tudo na RAM)
for i in range(0, 100000, 1000):
    fp[i:i+1000] = np.random.randn(1000, 10000)

# Flush para disco
fp.flush()
print("Array gigante criado sem usar RAM!")
```

memmap(): Leitura

Python

```python
# Abrir memmap existente (modo leitura)
fp = np.memmap('huge_data.dat',
               dtype='float32',
               mode='r',
               shape=(100000, 10000))
# Acessar pedacos sem carregar tudo
chunk = fp[0:1000, :]  # Primeiras 1000 linhas
print(f"Chunk shape: {chunk.shape}")

# Processar em batches
for i in range(0, len(fp), 1000):
    batch = fp[i:i+1000]
    # Processar batch...
    print(f"Processando batch {i//1000}")
```

memmap(): Leitura

Nota Importante

Essencial para big data que não cabe na RAM

Comparação de Formatos

Formato	Velocidade	Tamanho	Legível	Uso
.npy	Muito rápida	Médio	Não	Um array
.npz	Rápida	Médio	Não	Múltiplos arrays
.npz (comp)	Média	Pequeno	Não	Arquivamento
.txt	Lenta	Grande	Sim	Debug, compartilhar
.csv	Lenta	Grande	Sim	Excel, R, etc
memmap	Rápida	Médio	Não	Big data

Recomendação geral:

- Desenvolvimento: `.npz`
- Produção: `.npz_compressed`
- Big data: `memmap`
- Compartilhamento: `.csv`

Benchmark de Performance

</> Python

```python
import time
# Array de teste
arr = np.random.randn(10000, 1000)

# .npy
start = time.time()
np.save('test.npy', arr)
time_npy_save = time.time() - start
start = time.time()
_ = np.load('test.npy')
time_npy_load = time.time() - start
```

Benchmark de Performance (cont.)

Python

```python
# .txt
start = time.time()
np.savetxt('test.txt', arr)
time_txt_save = time.time() - start
start = time.time()
_ = np.loadtxt('test.txt')
time_txt_load = time.time() - start

```

Benchmark: Resultados

Python

```python
print("=== SALVAR ===")
print(f".npy: {time_npy_save:.3f}s")
print(f".txt: {time_txt_save:.3f}s")
print(f"Speedup: {time_txt_save/time_npy_save:.1f}x")

print("\n=== CARREGAR ===")
print(f".npy: {time_npy_load:.3f}s")
print(f".txt: {time_txt_load:.3f}s")
print(f"Speedup: {time_txt_load/time_npy_load:.1f}x")

# Resultado tipico:
# .npy é 10-50x mais rápido que .txt!

```

Tópicos Avançados (Bônus)

Para Exploração Futura

Conceitos avançados não cobertos hoje:

1. **Structured Arrays:** Arrays com campos nomeados
   - Como dicionários em arrays
   - Útil para dados heterogêneos
2. **Record Arrays:** Acesso por atributo
   - `arr.name` em vez de `arr['name']`
3. **Masked Arrays:** Dados com valores faltantes
   - Lidar com NaN e valores inválidos
4. **NumPy para GPU:** CuPy
   - NumPy com aceleração CUDA
5. **Numba:** JIT compilation
   - Compilar Python para velocidade C

Preview: Structured Arrays

Python

```python
# Definir estrutura
dtype = np.dtype([('nome', 'U10'),
                  ('idade', 'i4'),
                  ('salario', 'f8')])
# Criar array estruturado
pessoas = np.array([
    ('Ana', 25, 30000.0),
    ('Bruno', 30, 45000.0),
    ('Carlos', 35, 55000.0)
], dtype=dtype)

# Acessar campos
print(pessoas['nome'])      # ['Ana' 'Bruno' 'Carlos']
print(pessoas['idade'])     # [25 30 35]
print(pessoas[0])           # ('Ana', 25, 30000.)
```

Revisão da Aula

Bloco 1 - Manipulação de Shape:

- reshape, transpose, flatten, ravel
- squeeze, expand_dims, newaxis
- Preparação de dados para ML/DL

Bloco 2 - Álgebra Linear:

- Produtos: escalar, matricial, externo, interno
- Operações matriciais: inv, det, solve
- Autovalores, SVD, normas
- Aplicações: regressão linear, PCA

Revisão da Aula (cont.)

Bloco 3 - Concatenação:

- concatenate, stack, vstack, hstack
- Diferenças entre concatenate e stack
- split, vsplit, hsplit
- Combinação de datasets

Bloco 4 - I/O:

- save/load (.npy)
- savez/savez_compressed (.npz)
- savetxt/loadtxt (texto)
- memmap para big data
- Performance de diferentes formatos

Conceitos-Chave para Lembrar

1. **reshape(-1, x):** NumPy calcula dimensão automaticamente
2. **@ é produto matricial**, * é elemento-a-elemento
3. **Views vs Copies:** reshape retorna view, flatten retorna copy
4. **Broadcasting:** Fundamental para operações eficientes
5. **stack cria nova dimensão**, concatenate usa existente
6. **.npz é melhor formato** para persistência
7. **memmap para big data** que não cabe na RAM
8. **Sempre verificar shapes** antes de operações

Nota Importante

Estes conceitos são essenciais para trabalhar com dados reais!

NumPy na Prática: Workflow Completo

Pipeline típico de Data Science:

1. **Carregar:** `np.load()`, `np.loadtxt()`
2. **Explorar:** `.shape`, `.dtype`, estatísticas
3. **Limpar:** Boolean indexing, `np.where()`
4. **Transformar:** reshape, broadcasting, operações
5. **Combinar:** concatenate, stack, merge datasets
6. **Analisar:** Álgebra linear, estatísticas
7. **Preparar:** Train/test split, normalização
8. **Salvar:** `np.savez_compressed()`
9. **Modelar:** Passar para Scikit-learn, TensorFlow, etc.

Integrando com Outras Bibliotecas

NumPy como ponte:

- **Pandas:** `df.values` → NumPy array
- **Matplotlib:** Recebe arrays diretamente
- **Scikit-learn:** Todas funções usam arrays
- **TensorFlow:** `tf.constant(arr)`
- **PyTorch:** `torch.from_numpy(arr)`
- **PIL/OpenCV:** Imagens como arrays
- **SciPy:** Extensão científica do NumPy

Nota Importante

NumPy é a **lingua franca** do ecossistema Python científico

Dicas de Performance

Como otimizar código NumPy:

1. **Vetorize tudo:** Evite loops Python
2. **Use views:** reshape, transpose não copiam
3. **Broadcasting:** Em vez de loops ou tile
4. **Operações in-place:** `+=`, `*=` quando possível
5. **Tipo adequado:** float32 vs float64
6. **Contiguous arrays:** `np.ascontiguousarray()`
7. **Evite criar temporários:** Use `out` parameter
8. **Profile código:** Use `%timeit` no Jupyter

Debugging NumPy

Python

```python
# Sempre inspecione arrays
arr = some_function()
print(f"Shape: {arr.shape}")
print(f"Dtype: {arr.dtype}")
print(f"Min/Max: {arr.min()}/{arr.max()}")
print(f"Mean/Std: {arr.mean():.2f}/{arr.std():.2f}")
# Verificar valores invalidos
print(f"NaN? {np.any(np.isnan(arr))}")
print(f"Inf? {np.any(np.isinf(arr))}")
# Visualizar
print(f"Primeiros: {arr[:5]}")
print(f"Ultimos: {arr[-5:]}")
# Shape mismatch? Adicione prints!
print(f"A: {A.shape}, B: {B.shape}")
```

Erros Comuns e Soluções

1. **ValueError: shapes not aligned**
   - Verificar dimensões antes de multiplicar
   - Usar transpose ou reshape
2. **Modificações inesperadas (views)**
   - Usar `.copy()` quando necessário
3. **Broadcasting errado**
   - Usar `np.newaxis` para ajustar dimensões
4. **Dtype errado**
   - Especificar dtype explicitamente
   - Usar `.astype()` para converter
5. **Memória insuficiente**
   - Processar em batches
   - Usar memmap ou dtype menor

Recursos para Aprender Mais

Documentação:

- `https://numpy.org/doc/` - Documentação oficial
- `https://numpy.org/numpy-tutorials/` - Tutoriais

Livros:

- ”Python for Data Analysis- Wes McKinney
- ”Python Data Science Handbook- Jake VanderPlas

Prática:

- Kaggle datasets
- Project Euler (problemas matemáticos)
- Implementar algoritmos do zero

Comunidade:

- Stack Overflow - tag [numpy]
- GitHub - projetos open source

Próximos Passos

Depois de NumPy:

1. **Pandas:** DataFrames e análise de dados
   - Construído sobre NumPy
   - Manipulação de dados tabulares
2. **Matplotlib/Seaborn:** Visualização
   - Criar gráficos de arrays
   - Exploração visual de dados
3. **Scikit-learn:** Machine Learning
   - Modelos que usam arrays NumPy
   - Pipeline completo de ML
4. **SciPy:** Computação científica avançada
   - Otimização, integração, sinais

Preparação para Entrega

Checklist para entrega quinzenal (17/10):

- ☐ Revisar Aulas 1-4
- ☐ Completar todos os notebooks de prática
- ☐ Testar código localmente
- ☐ Verificar formatação e comentários
- ☐ Submeter via Moodle

Dicas:

- Teste incrementalmente
- Revise os slides das aulas

Resumo Final: NumPy Completo

Aulas 03 + 04 cobrimos:

Fundamentos:

- Criação de arrays
- Indexação e slicing
- Boolean/fancy indexing
- Broadcasting
- Operações vetorizadas
- Funções de agregação

Avançado:

- Manipulação de shape
- Álgebra linear
- Operações matriciais
- Concatenação e split
- I/O e persistência
- Otimização

Nota Importante

Com este conhecimento, você está pronto para **qualquer** tarefa de Data Science!

A Seguir: Aula Prática

✋ Exercício Prático

Tempo: 60 minutos
Entrega: via Moodle (notebook)
Tarefa: (atualizada em aula)

Obrigado!

Próxima aula teórica: Introdução ao Pandas
Terça-feira, 21/10