Contents

Curso Completo de Python: Do Básico à Inteligência Artificial

Figure 1: Python Logo

Do Zero ao Avançado: Fundamentos, Programação Avançada, Ciência de Dados, Machine Learning e Redes Neurais

Abril 2025

Este material foi desenvolvido para fornecer um guia completo de aprendizado de Python, desde os conceitos mais básicos até aplicações avançadas em Inteligência Artificial.

Estrutura do Curso

1. **Fundamentos de Python**
   - Introdução à linguagem Python
   - Instalação e configuração do ambiente
   - Variáveis e tipos de dados
   - Operadores
   - Estruturas condicionais
   - Estruturas de repetição
   - Funções

- Exercícios práticos

2. **Python Intermediário**
   - Estruturas de dados (listas, tuplas, dicionários, conjuntos)
   - Compreensão de listas
   - Manipulação de arquivos
   - Módulos e pacotes
   - Tratamento de exceções
   - Programação orientada a objetos
   - Exercícios práticos
3. **Python Avançado**
   - Decoradores
   - Geradores e iteradores
   - Expressões regulares
   - Programação funcional
   - Concorrência e paralelismo
   - Testes unitários
   - Exercícios práticos
4. **Ciência de Dados e Machine Learning**
   - Bibliotecas fundamentais (NumPy, Pandas)
   - Visualização de dados (Matplotlib, Seaborn)
   - Manipulação e análise de dados
   - Conceitos básicos de Machine Learning
   - Algoritmos de aprendizado supervisionado
   - Algoritmos de aprendizado não supervisionado
5. **Redes Neurais e Deep Learning**
   - Fundamentos de redes neurais
   - Redes neurais convolucionais (CNNs)
   - Redes neurais recorrentes (RNNs)
   - Transformers e modelos de linguagem
   - GANs e modelos generativos
   - Aplicações práticas e projetos

Índice

Módulo 1: Fundamentos de Python

Módulo 2: Python Intermediário

Módulo 3: Python Avançado

Módulo 4: Ciência de Dados e Machine Learning

Módulo 5: Redes Neurais e Deep Learning

1. Fundamentos de redes neurais
2. Redes neurais convolucionais
3. Redes neurais recorrentes
4. Transformers e modelos de linguagem
5. GANs e modelos generativos
6. Aplicações práticas e projetos

Referências

Módulo 1: Fundamentos de Python

Módulo 1: Fundamentos de Python

Introdução à Linguagem Python

Python é uma linguagem de programação de alto nível, interpretada, de script, imperativa, orientada a objetos, funcional, de tipagem dinâmica e forte. Foi lançada por Guido van Rossum em 1991 e desenvolvida por uma comunidade que determina seu direcionamento.

A filosofia de Python enfatiza a legibilidade do código com sua notável sintaxe clara. É uma linguagem que permite expressar conceitos em menos linhas de código do que seria possível em linguagens como C++ ou Java. Esta simplicidade torna Python uma excelente escolha tanto para iniciantes quanto para desenvolvedores experientes.

Python é versátil e encontra aplicações em diversos campos:

- Desenvolvimento web (Django, Flask)
- Ciência de dados e análise estatística
- Inteligência artificial e machine learning
- Automação e scripting
- Desenvolvimento de jogos
- Aplicações desktop

Uma das maiores vantagens do Python é seu ecossistema rico em bibliotecas e frameworks que facilitam o desenvolvimento em praticamente qualquer área. Além disso, possui uma comunidade ativa e acolhedora, sempre disposta a ajudar novos programadores.

Ao longo deste curso, exploraremos Python desde seus conceitos mais básicos até aplicações avançadas em inteligência artificial e machine learning. Vamos começar nossa jornada entendendo como instalar e configurar o ambiente Python.

Módulo 1: Fundamentos de Python

Instalação e Configuração do Ambiente

Para começar a programar em Python, precisamos primeiro configurar o ambiente de desenvolvimento. Python está disponível para todos os principais sistemas operacionais: Windows, macOS e Linux. Vamos ver como instalar e configurar o Python em cada um deles.

Instalação no Windows

1. Acesse o site oficial do Python (python.org) e baixe a versão mais recente para Windows.
2. Execute o instalador baixado. Importante: marque a opção "Add Python to PATH" antes de prosseguir com a instalação.
3. Clique em "Install Now" para uma instalação padrão ou "Customize installation" para personalizar.
4. Após a conclusão, abra o Prompt de Comando e digite `python --version` para verificar se a instalação foi bem-sucedida.

Instalação no macOS

O macOS geralmente vem com Python pré-instalado, mas pode ser uma versão mais antiga. Para instalar a versão mais recente:

1. A maneira recomendada é usar o Homebrew. Se não tiver o Homebrew instalado, instale-o executando:

   ```
   /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
   ```

2. Depois, instale o Python com:

   ```
   brew install python
   ```

3. Verifique a instalação com `python3 --version`.

Instalação no Linux

A maioria das distribuições Linux já vem com Python instalado. Para instalar a versão mais recente:

1. No Ubuntu/Debian:

   ```
   sudo apt update
   sudo apt install python3 python3-pip
   ```

2. No Fedora:

   ```
   sudo dnf install python3 python3-pip
   ```

3. Verifique a instalação com `python3 --version`.

Ambientes Virtuais

Uma prática recomendada é usar ambientes virtuais para isolar dependências de projetos diferentes. Para criar um ambiente virtual:

1. Instale o pacote virtualenv (se ainda não estiver instalado):

   ```
   pip install virtualenv
   ```

2. Crie um novo ambiente virtual:

   ```
   virtualenv meu_ambiente
   ```

3. Ative o ambiente virtual:

   - No Windows: `meu_ambiente\Scripts\activate`
   - No macOS/Linux: `source meu_ambiente/bin/activate`

IDEs e Editores de Código

Existem várias IDEs (Ambientes de Desenvolvimento Integrado) e editores de código que facilitam o desenvolvimento em Python:

1. **PyCharm**: Uma IDE completa com muitos recursos, disponível em versões gratuita (Community) e paga (Professional).
2. **Visual Studio Code**: Um editor de código leve e altamente personalizável com excelente suporte para Python através de extensões.
3. **Jupyter Notebook**: Ideal para ciência de dados e aprendizado de máquina, permite combinar código, visualizações e texto explicativo.
4. **IDLE**: A IDE básica que vem com a instalação padrão do Python.

Para iniciantes, recomendo o Visual Studio Code ou o PyCharm Community Edition, pois oferecem um bom equilíbrio entre recursos e facilidade de uso.

Agora que temos nosso ambiente configurado, estamos prontos para começar a programar em Python!

Módulo 1: Fundamentos de Python

Variáveis e Tipos de Dados

Em Python, as variáveis são espaços na memória que armazenam valores. Uma característica importante do Python é que não precisamos declarar o tipo de uma variável explicitamente - o interpretador infere o tipo com base no valor atribuído. Isso é conhecido como tipagem dinâmica.

Criando Variáveis

Para criar uma variável em Python, simplesmente atribuímos um valor a ela usando o operador de atribuição `=`:

```
nome = "Maria"
idade = 25
altura = 1.65
```

Neste exemplo, criamos três variáveis: `nome` (uma string), `idade` (um inteiro) e `altura` (um número de ponto flutuante).

Regras para Nomes de Variáveis

Em Python, os nomes de variáveis devem seguir algumas regras:

1. Podem conter letras, números e o caractere underscore (_)
2. Devem começar com uma letra ou underscore, nunca com um número
3. São sensíveis a maiúsculas e minúsculas (case-sensitive)
4. Não podem ser palavras reservadas do Python (como `if`, `for`, `while`, etc.)

Exemplos válidos:

```
nome = "João"
_idade = 30
valor_total = 1500.50
minhaVariavel = True
```

Exemplos inválidos:

```
2nome = "Maria"  # Começa com número
meu-nome = "Pedro"  # Contém hífen
if = 10  # Palavra reservada
```

Tipos de Dados Básicos

Python possui vários tipos de dados integrados. Os mais comuns são:

Números

- **int**: Números inteiros, como 1, 100, -10
- **float**: Números de ponto flutuante, como 3.14, -0.001, 2.0
- **complex**: Números complexos, como 1+2j

```
inteiro = 42
flutuante = 3.14159
complexo = 2 + 3j
```

Strings

Strings são sequências de caracteres, delimitadas por aspas simples ou duplas:

```
nome = "Ana"
mensagem = 'Olá, mundo!'
texto_longo = """Este é um texto
que ocupa várias
linhas."""
```

Podemos acessar caracteres individuais de uma string usando índices:

```
primeira_letra = nome[0]  # 'A'
ultima_letra = nome[-1]  # 'a'
```

E podemos obter substrings usando slicing:

```
primeiras_duas = nome[0:2]  # 'An'
```

Booleanos

O tipo booleano (`bool`) pode ter apenas dois valores: `True` ou `False`:

```
esta_chovendo = False
sol_brilhando = True
```

None

Python tem um valor especial, `None`, que representa a ausência de valor:

```
resultado = None
```

Verificando Tipos

Podemos verificar o tipo de uma variável usando a função `type()`:

```
x = 10
print(type(x))  # <class 'int'>

y = "Olá"
print(type(y))  # <class 'str'>
```

Conversão de Tipos

Python permite converter entre diferentes tipos de dados:

```
# Conversão para inteiro
idade_str = "25"
idade_int = int(idade_str)  # 25
```

```python
# Conversão para float
numero_str = "3.14"
numero_float = float(numero_str)  # 3.14

# Conversão para string
valor = 42
valor_str = str(valor)  # "42"

# Conversão para booleano
numero = 1
booleano = bool(numero)  # True (qualquer número diferente de 0 é True)
```

Variáveis Mutáveis e Imutáveis

Em Python, alguns tipos de dados são imutáveis (não podem ser alterados após a criação), enquanto outros são mutáveis (podem ser modificados).

Tipos imutáveis: - Números (int, float, complex) - Strings - Tuplas - Booleanos

Tipos mutáveis: - Listas - Dicionários - Conjuntos

Entender a diferença entre tipos mutáveis e imutáveis é importante para evitar comportamentos inesperados em seu código.

No próximo tópico, exploraremos os operadores em Python e como utilizá-los para realizar operações com nossas variáveis.

Módulo 1: Fundamentos de Python

Operadores

Os operadores são símbolos especiais que realizam operações em variáveis e valores. Python suporta vários tipos de operadores que nos permitem realizar diferentes tipos de operações.

Operadores Aritméticos

Os operadores aritméticos são usados para realizar operações matemáticas básicas:

Operador	Nome	Exemplo
+	Adição	x + y
-	Subtração	x - y
	Multiplicação	x * y
/	Divisão	x / y
%	Módulo (resto da divisão)	x % y
	Exponenciação	x ** y
//	Divisão inteira	x // y

Exemplos:

```
a = 10
b = 3

soma = a + b  # 13
subtracao = a - b  # 7
multiplicacao = a * b  # 30
divisao = a / b  # 3.3333...
modulo = a % b  # 1 (resto da divisão de 10 por 3)
exponenciacao = a ** b  # 1000 (10 elevado a 3)
divisao_inteira = a // b  # 3 (parte inteira da divisão de 10 por 3)
```

Um detalhe importante: em Python 3, a divisão (`/`) sempre retorna um número de ponto flutuante, mesmo quando o resultado é um número inteiro. Se você quiser obter apenas a parte inteira da divisão, use o operador de divisão inteira (`//`).

Operadores de Atribuição

Os operadores de atribuição são usados para atribuir valores a variáveis:

Operador	Exemplo	Equivalente a
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
=	x **= 3	x = x ** 3
//=	x //= 3	x = x // 3

Exemplos:

```
x = 10
x += 5  # x agora é 15
x -= 3  # x agora é 12
x *= 2  # x agora é 24
x /= 4  # x agora é 6.0
```

Operadores de Comparação

Os operadores de comparação são usados para comparar dois valores:

Operador	Nome	Exemplo
==	Igual a	x == y
!=	Diferente de	x != y
>	Maior que	x > y
<	Menor que	x < y
>=	Maior ou igual a	x >= y
<=	Menor ou igual a	x <= y

Exemplos:

```
a = 10
b = 5

print(a == b)  # False
print(a != b)  # True
print(a > b)   # True
print(a < b)   # False
print(a >= b)  # True
print(a <= b)  # False
```

Estes operadores retornam valores booleanos (`True` ou `False`) e são frequentemente usados em estruturas condicionais.

Operadores Lógicos

Os operadores lógicos são usados para combinar expressões condicionais:

Operador	Descrição	Exemplo
and	Retorna True se ambas as expressões forem verdadeiras	x < 5 and x < 10
or	Retorna True se uma das expressões for verdadeira	x < 5 or x < 4
not	Inverte o resultado, retorna False se o resultado for True	not(x < 5 and x < 10)

Exemplos:

```
x = 5
y = 10

print(x > 3 and y < 15)  # True (ambas as condições são verdadeiras)
print(x > 7 or y < 15)   # True (a segunda condição é verdadeira)
print(not(x > 3))        # False (inverte True para False)
```

Operadores de Identidade

Os operadores de identidade são usados para comparar objetos, verificando se eles são o mesmo objeto com o mesmo local de memória:

Operador	Descrição	Exemplo
is	Retorna True se ambas as variáveis são o mesmo objeto	x is y
is not	Retorna True se as variáveis não são o mesmo objeto	x is not y

Exemplos:

```
a = [1, 2, 3]
b = [1, 2, 3]
c = a

print(a is c)  # True (a e c referem-se ao mesmo objeto)
print(a is b)  # False (a e b são objetos diferentes, mesmo tendo o mesmo conteúdo)
print(a == b)  # True (a e b têm o mesmo conteúdo)
```

Operadores de Associação

Os operadores de associação são usados para verificar se um valor está presente em uma sequência (string, lista, tupla, conjunto ou dicionário):

Operador	Descrição	Exemplo
in	Retorna True se o valor estiver presente na sequência	x in y
not in	Retorna True se o valor não estiver presente na sequência	x not in y

Exemplos:

```python
frutas = ["maçã", "banana", "laranja"]
print("banana" in frutas)      # True
print("abacaxi" in frutas)     # False
print("abacaxi" not in frutas) # True

texto = "Python é incrível"
print("Python" in texto)       # True
print("Java" in texto)         # False
```

Precedência de Operadores

Assim como na matemática, Python segue uma ordem de precedência para avaliar expressões com múltiplos operadores:

1. Parênteses `()`
2. Exponenciação `**`
3. Operadores unários `+x`, `-x`, `~x`
4. Multiplicação, divisão, módulo, divisão inteira `*`, `/`, `%`, `//`
5. Adição e subtração `+`, `-`
6. Operadores de deslocamento de bits `<<`, `>>`
7. Operador bitwise AND `&`
8. Operador bitwise XOR `^`
9. Operador bitwise OR `|`
10. Operadores de comparação `==`, `!=`, `>`, `>=`, `<`, `<=`, `is`, `is not`, `in`, `not in`
11. Operadores lógicos `not`, `and`, `or`

Se você tiver dúvidas sobre a ordem de avaliação, use parênteses para tornar suas intenções explícitas.

No próximo tópico, exploraremos as estruturas condicionais em Python, que nos permitem executar diferentes blocos de código com base em condições.

Módulo 1: Fundamentos de Python

Estruturas Condicionais

As estruturas condicionais permitem que seu programa tome decisões e execute diferentes blocos de código com base em condições específicas. Em Python, as principais estruturas condicionais são `if`, `elif` e `else`.

Declaração if

A declaração `if` é a estrutura condicional mais básica. Ela executa um bloco de código se uma condição for avaliada como verdadeira:

```
idade = 18

if idade >= 18:
    print("Você é maior de idade.")
```

Neste exemplo, a mensagem "Você é maior de idade" será exibida apenas se a variável `idade` for maior ou igual a 18.

Declaração if-else

A declaração `if-else` permite executar um bloco de código se a condição for verdadeira e outro bloco se a condição for falsa:

```
idade = 16

if idade >= 18:
    print("Você é maior de idade.")
else:
    print("Você é menor de idade.")
```

Neste exemplo, como a idade é 16, a mensagem "Você é menor de idade" será exibida.

Declaração if-elif-else

A declaração `if-elif-else` permite verificar múltiplas condições em sequência:

```
nota = 85

if nota >= 90:
    print("Conceito A")
elif nota >= 80:
    print("Conceito B")
elif nota >= 70:
    print("Conceito C")
```

```
elif nota >= 60:
    print("Conceito D")
else:
    print("Conceito F")
```

Neste exemplo, como a nota é 85, a mensagem "Conceito B" será exibida. O Python verifica cada condição em ordem e executa o bloco de código associado à primeira condição verdadeira que encontrar.

Condições Aninhadas

Podemos aninhar estruturas condicionais dentro de outras estruturas condicionais:

```
idade = 25
tem_carteira = True

if idade >= 18:
    if tem_carteira:
        print("Você pode dirigir.")
    else:
        print("Você tem idade para dirigir, mas precisa de uma carteira.")
else:
    print("Você é muito jovem para dirigir.")
```

Neste exemplo, como a idade é 25 e `tem_carteira` é `True`, a mensagem "Você pode dirigir" será exibida.

Operadores Lógicos em Condições

Podemos usar operadores lógicos (`and`, `or`, `not`) para combinar ou modificar condições:

```
idade = 25
tem_carteira = True

if idade >= 18 and tem_carteira:
    print("Você pode dirigir.")
elif idade >= 18 and not tem_carteira:
    print("Você tem idade para dirigir, mas precisa de uma carteira.")
else:
    print("Você é muito jovem para dirigir.")
```

Este código produz o mesmo resultado que o exemplo anterior, mas de uma forma mais concisa.

Expressões Condicionais (Operador Ternário)

Python oferece uma forma compacta de escrever condições simples, conhecida como operador ternário:

```
idade = 20
status = "maior de idade" if idade >= 18 else "menor de idade"
print(f"Você é {status}.")
```

Esta linha equivale a:

```
if idade >= 18:
    status = "maior de idade"
else:
    status = "menor de idade"
print(f"Você é {status}.")
```

Verificando Valores Vazios ou Nulos

Em Python, os seguintes valores são avaliados como `False` em um contexto booleano: - `False` - `None` - Zero de qualquer tipo numérico (`0`, `0.0`, `0j`) - Sequências vazias (`''`, `[]`, `()`, `{}`, `set()`)

Isso permite verificações simplificadas:

```
nome = ""

if nome:
    print(f"Olá, {nome}!")
else:
    print("Olá, visitante!")
```

Como `nome` é uma string vazia, a condição é avaliada como `False` e a mensagem "Olá, visitante!" será exibida.

Boas Práticas

1. **Indentação**: Python usa indentação para definir blocos de código. A convenção é usar 4 espaços para cada nível de indentação.
2. **Clareza**: Escreva condições claras e legíveis. Se uma condição for muito complexa, considere dividi-la em partes menores ou usar variáveis intermediárias.
3. **Evite Aninhamento Excessivo**: Muitos níveis de aninhamento podem tornar o código difícil de ler. Considere reestruturar o código ou usar funções para reduzir o aninhamento.
4. **Use Parênteses para Clareza**: Quando combinar múltiplos operadores lógicos, use parênteses para tornar a precedência explícita.

As estruturas condicionais são fundamentais para criar programas que podem tomar decisões com base em diferentes situações. No próximo tópico, exploraremos as estruturas de repetição, que nos permitem executar blocos de código repetidamente.

Módulo 1: Fundamentos de Python

Estruturas de Repetição

As estruturas de repetição, também conhecidas como loops, permitem executar um bloco de código várias vezes. Isso é fundamental para automatizar tarefas repetitivas e processar coleções de dados. Python oferece três tipos principais de estruturas de repetição: `while`, `for` e compreensões.

Loop while

O loop `while` executa um bloco de código enquanto uma condição específica for verdadeira:

```
contador = 1
while contador <= 5:
    print(f"Contagem: {contador}")
    contador += 1
```

Este código imprimirá:

```
Contagem: 1
Contagem: 2
Contagem: 3
Contagem: 4
Contagem: 5
```

O loop `while` verifica a condição antes de cada iteração. Se a condição for falsa desde o início, o bloco de código não será executado nenhuma vez.

É importante garantir que a condição eventualmente se torne falsa, caso contrário, você criará um "loop infinito" que continuará executando até que o programa seja forçado a parar.

Loop for

O loop `for` em Python é usado para iterar sobre uma sequência (como uma lista, tupla, dicionário, conjunto ou string):

```
frutas = ["maçã", "banana", "laranja", "uva"]
for fruta in frutas:
    print(f"Eu gosto de {fruta}")
```

Este código imprimirá:

```
Eu gosto de maçã
Eu gosto de banana
Eu gosto de laranja
Eu gosto de uva
```

O loop `for` é especialmente útil quando você sabe antecipadamente quantas vezes deseja executar um bloco de código ou quando deseja processar cada item em uma coleção.

Função range()

A função `range()` é frequentemente usada com loops `for` para gerar uma sequência de números:

```
# range(stop) - gera números de 0 até stop-1
for i in range(5):
    print(i)  # Imprime 0, 1, 2, 3, 4

# range(start, stop) - gera números de start até stop-1
for i in range(2, 6):
    print(i)  # Imprime 2, 3, 4, 5

# range(start, stop, step) - gera números de start até stop-1 com incremento step
for i in range(1, 10, 2):
    print(i)  # Imprime 1, 3, 5, 7, 9
```

Declarações break e continue

Python fornece duas declarações especiais para controlar o comportamento dos loops:

- `break`: Sai imediatamente do loop
- `continue`: Pula para a próxima iteração do loop

Exemplo com `break`:

```
for i in range(1, 11):
    if i == 5:
        break
    print(i)
```

Este código imprimirá apenas os números 1, 2, 3 e 4, pois o loop é interrompido quando `i` é igual a 5.

Exemplo com `continue`:

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
```

Este código imprimirá os números 1, 2, 4 e 5, pulando o 3.

Loop else

Em Python, os loops `for` e `while` podem ter uma cláusula `else` opcional. O bloco de código no `else` é executado quando o loop termina normalmente (ou seja, não foi interrompido por um `break`):

```
for i in range(1, 6):
    print(i)
else:
    print("Loop concluído com sucesso!")
```

Este código imprimirá os números de 1 a 5 e, em seguida, a mensagem "Loop concluído com sucesso!".

Se o loop for interrompido por um `break`, o bloco `else` não será executado:

```python
for i in range(1, 6):
    if i == 3:
        break
    print(i)
else:
    print("Loop concluído com sucesso!")
```

Este código imprimirá apenas os números 1 e 2, e a mensagem "Loop concluído com sucesso!" não será exibida.

Loops Aninhados

Podemos aninhar loops dentro de outros loops:

```python
for i in range(1, 4):
    for j in range(1, 4):
        print(f"({i}, {j})")
```

Este código imprimirá todas as combinações de pares ordenados (i, j) onde i e j variam de 1 a 3.

Compreensões de Lista

As compreensões de lista oferecem uma sintaxe concisa para criar listas baseadas em listas existentes:

```python
# Cria uma lista com os quadrados dos números de 1 a 5
quadrados = [x**2 for x in range(1, 6)]
print(quadrados)  # [1, 4, 9, 16, 25]

# Cria uma lista apenas com números pares de 1 a 10
pares = [x for x in range(1, 11) if x % 2 == 0]
print(pares)  # [2, 4, 6, 8, 10]
```

As compreensões de lista são mais concisas e muitas vezes mais legíveis do que loops `for` tradicionais para operações simples.

Boas Práticas

1. **Escolha o Loop Adequado**: Use `for` quando souber o número de iterações antecipadamente ou estiver iterando sobre uma coleção. Use `while` quando precisar continuar até que uma condição seja atendida.
2. **Evite Loops Infinitos**: Certifique-se de que a condição em um loop `while` eventualmente se torne falsa.
3. **Use break e continue com Moderação**: Embora úteis, o uso excessivo de `break` e `continue` pode tornar o código difícil de entender.
4. **Prefira Compreensões para Operações Simples**: Para transformações simples de listas, as compreensões de lista são geralmente mais legíveis e eficientes.
5. **Limite o Aninhamento**: Loops profundamente aninhados podem ser difíceis de entender. Considere refatorar o código ou usar funções para reduzir o aninhamento.

As estruturas de repetição são essenciais para automatizar tarefas e processar dados em Python. No próximo tópico, exploraremos as funções, que nos permitem organizar o código em blocos reutilizáveis.

Módulo 1: Fundamentos de Python

Funções

As funções são blocos de código reutilizáveis que realizam uma tarefa específica. Elas permitem organizar o código, evitar repetição e tornar o programa mais modular e fácil de manter. Em Python, as funções são definidas usando a palavra-chave `def`.

Definindo Funções

A sintaxe básica para definir uma função em Python é:

```
def nome_da_funcao(parametro1, parametro2, ...):
    """Docstring - documentação da função."""
    # Corpo da função
    # Código a ser executado
    return valor  # Opcional
```

Exemplo de uma função simples:

```
def saudacao(nome):
    """Retorna uma mensagem de saudação personalizada."""
    return f"Olá, {nome}! Bem-vindo ao curso de Python."

# Chamando a função
mensagem = saudacao("Maria")
print(mensagem)  # Imprime: Olá, Maria! Bem-vindo ao curso de Python.
```

Parâmetros e Argumentos

Os termos "parâmetro" e "argumento" são frequentemente usados de forma intercambiável, mas há uma diferença sutil:

- **Parâmetros** são as variáveis listadas na definição da função.
- **Argumentos** são os valores reais passados para a função quando ela é chamada.

Python suporta diferentes tipos de parâmetros:

Parâmetros Posicionais

São os parâmetros mais comuns, onde a ordem dos argumentos importa:

```
def potencia(base, expoente):
    return base ** expoente

resultado = potencia(2, 3)  # 2³ = 8
```

Parâmetros com Valores Padrão

Podemos definir valores padrão para parâmetros, que serão usados se nenhum argumento for fornecido:

```
def potencia(base, expoente=2):
    return base ** expoente

resultado1 = potencia(2, 3)  # 2³ = 8
resultado2 = potencia(2)     # 2² = 4 (usa o valor padrão para expoente)
```

Parâmetros Nomeados

Podemos especificar os argumentos pelo nome do parâmetro, o que torna a chamada da função mais clara e permite ignorar a ordem:

```
def saudacao(nome, mensagem="Olá"):
    return f"{mensagem}, {nome}!"

# Usando argumentos posicionais
print(saudacao("João"))  # Olá, João!

# Usando argumentos nomeados
print(saudacao(mensagem="Bom dia", nome="João"))  # Bom dia, João!
```

Número Variável de Argumentos

Python permite definir funções que aceitam um número variável de argumentos:

- `*args`: Coleta argumentos posicionais extras em uma tupla
- `**kwargs`: Coleta argumentos nomeados extras em um dicionário

```
def soma(*numeros):
    """Soma um número variável de argumentos."""
    resultado = 0
    for numero in numeros:
        resultado += numero
    return resultado

print(soma(1, 2))          # 3
print(soma(1, 2, 3, 4, 5)) # 15

def info_pessoa(nome, idade, **dados_adicionais):
    """Exibe informações sobre uma pessoa."""
    print(f"Nome: {nome}")
    print(f"Idade: {idade}")
    for chave, valor in dados_adicionais.items():
        print(f"{chave.capitalize()}: {valor}")

info_pessoa("Ana", 30, profissao="Engenheira", cidade="São Paulo")
# Nome: Ana
# Idade: 30
# Profissao: Engenheira
# Cidade: São Paulo
```

Valor de Retorno

As funções podem retornar valores usando a instrução `return`. Se nenhum valor for especificado após `return` ou se não houver uma instrução `return`, a função retornará `None`.

```
def quadrado(numero):
    return numero ** 2

def sem_retorno():
    print("Esta função não retorna nada explicitamente")

resultado1 = quadrado(5)      # resultado1 = 25
resultado2 = sem_retorno()    # resultado2 = None
```

Uma função pode retornar múltiplos valores, que são empacotados em uma tupla:

```
def operacoes(a, b):
    soma = a + b
    diferenca = a - b
    produto = a * b
    quociente = a / b
    return soma, diferenca, produto, quociente

s, d, p, q = operacoes(10, 2)
print(f"Soma: {s}, Diferença: {d}, Produto: {p}, Quociente: {q}")
# Soma: 12, Diferença: 8, Produto: 20, Quociente: 5.0
```

Escopo de Variáveis

O escopo de uma variável determina onde ela pode ser acessada no código:

- **Variáveis locais**: Definidas dentro de uma função e acessíveis apenas dentro dessa função.
- **Variáveis globais**: Definidas no nível principal do programa e acessíveis em todo o programa.

```
x = 10  # Variável global

def funcao():
    y = 5  # Variável local
    print(x)  # Pode acessar a variável global
    print(y)  # Pode acessar a variável local

funcao()
print(x)  # Pode acessar a variável global
# print(y)  # Erro! Não pode acessar a variável local fora da função
```

Se você quiser modificar uma variável global dentro de uma função, deve usar a palavra-chave `global`:

```
contador = 0  # Variável global

def incrementar():
    global contador
    contador += 1
    print(f"Contador: {contador}")

incrementar()  # Contador: 1
incrementar()  # Contador: 2
```

Funções Anônimas (Lambda)

Python permite criar pequenas funções anônimas usando a palavra-chave `lambda`. Estas funções são limitadas a uma única expressão:

```
# Função tradicional
def quadrado(x):
    return x ** 2

# Equivalente usando lambda
quadrado_lambda = lambda x: x ** 2

print(quadrado(5))        # 25
print(quadrado_lambda(5)) # 25
```

As funções lambda são frequentemente usadas com funções como `map()`, `filter()` e `sorted()`:

```
# Usando map() com lambda para aplicar uma função a cada item de uma lista
numeros = [1, 2, 3, 4, 5]
quadrados = list(map(lambda x: x ** 2, numeros))
print(quadrados)  # [1, 4, 9, 16, 25]

# Usando filter() com lambda para filtrar itens de uma lista
pares = list(filter(lambda x: x % 2 == 0, numeros))
print(pares)  # [2, 4]

# Usando sorted() com lambda para ordenar uma lista de tuplas pelo segundo elemento
pessoas = [("João", 25), ("Maria", 30), ("Pedro", 20)]
pessoas_ordenadas = sorted(pessoas, key=lambda pessoa: pessoa[1])
print(pessoas_ordenadas)  # [('Pedro', 20), ('João', 25), ('Maria', 30)]
```

Docstrings

As docstrings são strings de documentação que descrevem o que uma função faz. Elas são colocadas logo após a definição da função e são delimitadas por aspas triplas:

```
def area_retangulo(largura, altura):
    """
    Calcula a área de um retângulo.

    Parâmetros:
    largura (float): A largura do retângulo
    altura (float): A altura do retângulo

    Retorna:
    float: A área do retângulo
    """
    return largura * altura
```

As docstrings podem ser acessadas usando o atributo `__doc__` ou a função `help()`:

```
print(area_retangulo.__doc__)
help(area_retangulo)
```

Funções Recursivas

Uma função recursiva é uma função que chama a si mesma:

```python
def fatorial(n):
    """Calcula o fatorial de um número usando recursão."""
    if n == 0 or n == 1:
        return 1
    else:
        return n * fatorial(n - 1)

print(fatorial(5))  # 5! = 5 * 4 * 3 * 2 * 1 = 120
```

A recursão deve ser usada com cuidado, pois pode levar a um estouro de pilha se a função se chamar muitas vezes.

Boas Práticas

1. **Nomes Descritivos**: Use nomes de funções que descrevam claramente o que a função faz.
2. **Uma Tarefa por Função**: Cada função deve realizar uma única tarefa bem definida.
3. **Funções Pequenas**: Mantenha as funções pequenas e focadas. Se uma função estiver fazendo muitas coisas, considere dividi-la.
4. **Docstrings**: Documente suas funções com docstrings para explicar o que elas fazem, seus parâmetros e valores de retorno.
5. **Valores Padrão Imutáveis**: Use apenas valores imutáveis (como números, strings ou None) como valores padrão para parâmetros.
6. **Evite Efeitos Colaterais**: Prefira funções que retornem valores em vez de modificar variáveis globais ou parâmetros mutáveis.

As funções são um dos conceitos mais importantes em programação, permitindo criar código modular, reutilizável e mais fácil de manter. Dominar o uso de funções é essencial para se tornar um programador Python eficiente.

No próximo módulo, exploraremos tópicos intermediários de Python, incluindo estruturas de dados mais avançadas, manipulação de arquivos e programação orientada a objetos.

Módulo 1: Fundamentos de Python

Exercícios Práticos

Nesta seção, apresentamos uma série de exercícios práticos para consolidar os conhecimentos adquiridos no Módulo 1. A prática é fundamental para o aprendizado de programação, pois permite aplicar os conceitos teóricos em situações concretas.

Exercício 1: Variáveis e Tipos de Dados

Crie um programa que solicite ao usuário seu nome, idade e altura. Em seguida, exiba uma mensagem formatada com essas informações.

```
# Solução
nome = input("Digite seu nome: ")
idade = int(input("Digite sua idade: "))
altura = float(input("Digite sua altura em metros: "))

print(f"Olá, {nome}! Você tem {idade} anos e mede {altura:.2f}m.")
```

Exercício 2: Operadores

Escreva um programa que calcule a área e o perímetro de um retângulo. O usuário deve fornecer a largura e o comprimento.

```
# Solução
largura = float(input("Digite a largura do retângulo: "))
comprimento = float(input("Digite o comprimento do retângulo: "))

area = largura * comprimento
perimetro = 2 * (largura + comprimento)

print(f"Área do retângulo: {area:.2f} unidades quadradas")
print(f"Perímetro do retângulo: {perimetro:.2f} unidades")
```

Exercício 3: Estruturas Condicionais

Crie um programa que determine se um ano fornecido pelo usuário é bissexto ou não. Um ano é bissexto se for divisível por 4, exceto para anos centenários (divisíveis por 100) que não são divisíveis por 400.

```
# Solução
ano = int(input("Digite um ano: "))

if (ano % 4 == 0 and ano % 100 != 0) or (ano % 400 == 0):
    print(f"{ano} é um ano bissexto.")
```

```
else:
    print(f"{ano} não é um ano bissexto.")
```

Exercício 4: Estruturas de Repetição

Escreva um programa que imprima a tabuada de um número fornecido pelo usuário.

```
# Solução
numero = int(input("Digite um número para ver sua tabuada: "))

print(f"Tabuada do {numero}:")
for i in range(1, 11):
    resultado = numero * i
    print(f"{numero} x {i} = {resultado}")
```

Exercício 5: Funções

Crie uma função que receba uma lista de números e retorne a média, o maior valor e o menor valor.

```
# Solução
def analisar_numeros(lista):
    """
    Analisa uma lista de números e retorna a média, o maior e o menor valor.

    Parâmetros:
    lista (list): Lista de números a ser analisada

    Retorna:
    tuple: (média, maior valor, menor valor)
    """
    if not lista:
        return None, None, None

    media = sum(lista) / len(lista)
    maior = max(lista)
    menor = min(lista)

    return media, maior, menor

# Teste da função
numeros = [10, 5, 7, 2, 9, 15, 3]
media, maior, menor = analisar_numeros(numeros)

print(f"Lista: {numeros}")
print(f"Média: {media:.2f}")
print(f"Maior valor: {maior}")
print(f"Menor valor: {menor}")
```

Exercício 6: Calculadora Simples

Crie uma calculadora simples que permita ao usuário realizar operações básicas (adição, subtração, multiplicação e divisão) entre dois números.

```
# Solução
def calculadora():
```

```
    """Função que implementa uma calculadora simples."""
    print("Calculadora Simples")
    print("Operações disponíveis:")
    print("1. Adição (+)")
    print("2. Subtração (-)")
    print("3. Multiplicação (*)")
    print("4. Divisão (/)")

    operacao = input("Escolha uma operação (1/2/3/4): ")

    num1 = float(input("Digite o primeiro número: "))
    num2 = float(input("Digite o segundo número: "))

    if operacao == "1":
        resultado = num1 + num2
        simbolo = "+"
    elif operacao == "2":
        resultado = num1 - num2
        simbolo = "-"
    elif operacao == "3":
        resultado = num1 * num2
        simbolo = "*"
    elif operacao == "4":
        if num2 == 0:
            return "Erro: Divisão por zero!"
        resultado = num1 / num2
        simbolo = "/"
    else:
        return "Operação inválida!"

    return f"{num1} {simbolo} {num2} = {resultado}"

print(calculadora())
```

Exercício 7: Jogo de Adivinhação

Crie um jogo simples onde o computador escolhe um número aleatório entre 1 e 100, e o usuário tenta adivinhar qual é esse número. O programa deve informar se o palpite está alto, baixo ou correto.

```
# Solução
import random

def jogo_adivinhacao():
    """Implementa um jogo de adivinhação de números."""
    numero_secreto = random.randint(1, 100)
    tentativas = 0
    max_tentativas = 10

    print("Bem-vindo ao Jogo de Adivinhação!")
    print(f"Estou pensando em um número entre 1 e 100. Você tem {max_tentativas} tentativas.")

    while tentativas < max_tentativas:
        tentativas += 1
```

```
        palpite = int(input(f"Tentativa {tentativas}: Digite seu palpite: "))

        if palpite < numero_secreto:
            print("Muito baixo! Tente um número maior.")
        elif palpite > numero_secreto:
            print("Muito alto! Tente um número menor.")
        else:
            print(f"Parabéns! Você acertou o número {numero_secreto} em {tentativas} tentativas!")
            return

    print(f"Suas tentativas acabaram! O número secreto era {numero_secreto}.")

jogo_adivinhacao()
```

Exercício 8: Conversor de Temperatura

Crie um programa que converta temperaturas entre Celsius, Fahrenheit e Kelvin.

```
# Solução
def converter_temperatura():
    """Converte temperaturas entre Celsius, Fahrenheit e Kelvin."""
    print("Conversor de Temperatura")
    print("Escolha a conversão:")
    print("1. Celsius para Fahrenheit")
    print("2. Celsius para Kelvin")
    print("3. Fahrenheit para Celsius")
    print("4. Fahrenheit para Kelvin")
    print("5. Kelvin para Celsius")
    print("6. Kelvin para Fahrenheit")

    opcao = input("Digite sua opção (1-6): ")

    if opcao not in ["1", "2", "3", "4", "5", "6"]:
        return "Opção inválida!"

    temperatura = float(input("Digite a temperatura: "))

    if opcao == "1":  # Celsius para Fahrenheit
        resultado = (temperatura * 9/5) + 32
        return f"{temperatura}°C = {resultado:.2f}°F"

    elif opcao == "2":  # Celsius para Kelvin
        resultado = temperatura + 273.15
        return f"{temperatura}°C = {resultado:.2f}K"

    elif opcao == "3":  # Fahrenheit para Celsius
        resultado = (temperatura - 32) * 5/9
        return f"{temperatura}°F = {resultado:.2f}°C"

    elif opcao == "4":  # Fahrenheit para Kelvin
        resultado = (temperatura - 32) * 5/9 + 273.15
        return f"{temperatura}°F = {resultado:.2f}K"

    elif opcao == "5":  # Kelvin para Celsius
```

```
        resultado = temperatura - 273.15
        return f"{temperatura}K = {resultado:.2f}°C"

    elif opcao == "6":  # Kelvin para Fahrenheit
        resultado = (temperatura - 273.15) * 9/5 + 32
        return f"{temperatura}K = {resultado:.2f}°F"

print(converter_temperatura())
```

Exercício 9: Verificador de Palíndromos

Crie uma função que verifique se uma palavra ou frase é um palíndromo (lê-se igual de trás para frente, desconsiderando espaços e pontuação).

```
# Solução
def eh_palindromo(texto):
    """
    Verifica se um texto é um palíndromo.

    Parâmetros:
    texto (str): Texto a ser verificado

    Retorna:
    bool: True se for palíndromo, False caso contrário
    """
    # Remove espaços e converte para minúsculas
    texto = texto.lower().replace(" ", "")

    # Remove pontuação
    for char in ",.!?;:":
        texto = texto.replace(char, "")

    # Verifica se o texto é igual ao seu inverso
    return texto == texto[::-1]

# Teste da função
frases = [
    "Ana",
    "A sacada da casa",
    "Socorram-me, subi no ônibus em Marrocos",
    "Python é incrível"
]

for frase in frases:
    if eh_palindromo(frase):
        print(f'"{frase}" é um palíndromo.')
    else:
        print(f'"{frase}" não é um palíndromo.')
```

Exercício 10: Contador de Palavras

Crie um programa que conte o número de palavras em um texto fornecido pelo usuário.

```python
# Solução
def contar_palavras(texto):
    """
    Conta o número de palavras em um texto.

    Parâmetros:
    texto (str): Texto a ser analisado

    Retorna:
    int: Número de palavras no texto
    """
    # Remove espaços extras e divide o texto em palavras
    palavras = texto.strip().split()
    return len(palavras)

# Teste da função
texto = input("Digite um texto: ")
num_palavras = contar_palavras(texto)

if num_palavras == 1:
    print(f"O texto contém 1 palavra.")
else:
    print(f"O texto contém {num_palavras} palavras.")
```

Estes exercícios foram projetados para reforçar os conceitos fundamentais de Python que aprendemos neste módulo. Recomendo que você tente resolver cada exercício por conta própria antes de olhar a solução. A prática constante é a chave para se tornar um programador proficiente.

No próximo módulo, exploraremos tópicos intermediários de Python, incluindo estruturas de dados mais avançadas, manipulação de arquivos e programação orientada a objetos.

Módulo 2: Python Intermediário

Módulo 2: Python Intermediário

Estruturas de Dados

As estruturas de dados são fundamentais na programação, pois permitem organizar e manipular dados de forma eficiente. Python oferece várias estruturas de dados integradas, cada uma com características e usos específicos. Neste tópico, exploraremos em profundidade as principais estruturas de dados em Python: listas, tuplas, dicionários e conjuntos.

Listas

As listas são uma das estruturas de dados mais versáteis em Python. Elas são coleções ordenadas e mutáveis de itens, que podem ser de diferentes tipos.

Criação de Listas

```
# Lista vazia
lista_vazia = []

# Lista com elementos
numeros = [1, 2, 3, 4, 5]
frutas = ["maçã", "banana", "laranja"]
misturada = [1, "Python", True, 3.14]

# Lista usando a função list()
caracteres = list("Python")  # ['P', 'y', 't', 'h', 'o', 'n']
```

Acesso a Elementos

Os elementos de uma lista são acessados por índices, que começam em 0:

```
frutas = ["maçã", "banana", "laranja", "uva", "pera"]

# Acesso por índice positivo (do início para o fim)
print(frutas[0])  # maçã
print(frutas[2])  # laranja

# Acesso por índice negativo (do fim para o início)
print(frutas[-1])  # pera
print(frutas[-3])  # laranja

# Slicing (fatiamento)
print(frutas[1:4])    # ['banana', 'laranja', 'uva']
print(frutas[:3])     # ['maçã', 'banana', 'laranja']
```

```
print(frutas[2:])     # ['laranja', 'uva', 'pera']
print(frutas[::2])    # ['maçã', 'laranja', 'pera']
print(frutas[::-1])   # ['pera', 'uva', 'laranja', 'banana', 'maçã']
```

Métodos de Listas

Python oferece vários métodos para manipular listas:

```
frutas = ["maçã", "banana", "laranja"]

# Adicionar elementos
frutas.append("uva")          # Adiciona ao final: ['maçã', 'banana', 'laranja', 'uva']
frutas.insert(1, "abacaxi")   # Insere na posição 1: ['maçã', 'abacaxi', 'banana', 'laranja', 'uva']
frutas.extend(["pera", "kiwi"])  # Estende a lista: ['maçã', 'abacaxi', 'banana', 'laranja', 'uva', 'pe

# Remover elementos
frutas.remove("banana")       # Remove a primeira ocorrência: ['maçã', 'abacaxi', 'laranja', 'uva', 'pe
elemento = frutas.pop(2)      # Remove e retorna o elemento no índice 2: 'laranja'
del frutas[0]                 # Remove o elemento no índice 0: ['abacaxi', 'uva', 'pera', 'kiwi']

# Outras operações
frutas.sort()                 # Ordena a lista: ['abacaxi', 'kiwi', 'pera', 'uva']
frutas.reverse()              # Inverte a lista: ['uva', 'pera', 'kiwi', 'abacaxi']
indice = frutas.index("pera") # Retorna o índice da primeira ocorrência: 1
contagem = frutas.count("uva") # Conta ocorrências: 1
frutas.clear()                # Remove todos os elementos: []
```

Listas Aninhadas (Multidimensionais)

As listas podem conter outras listas, criando estruturas multidimensionais:

```
# Matriz 3x3
matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Acesso a elementos
print(matriz[1][2])  # 6 (elemento na linha 1, coluna 2)

# Iteração sobre uma matriz
for linha in matriz:
    for elemento in linha:
        print(elemento, end=" ")
    print()  # Nova linha após cada linha da matriz
```

Tuplas

As tuplas são semelhantes às listas, mas são imutáveis, ou seja, não podem ser modificadas após a criação. Elas são usadas para armazenar coleções de itens que não devem ser alterados.

Criação de Tuplas

```
# Tupla vazia
tupla_vazia = ()

# Tupla com elementos
coordenadas = (10, 20)
pessoa = ("João", 30, "Engenheiro")

# Tupla com um único elemento (observe a vírgula)
singleton = (42,)

# Tupla usando a função tuple()
letras = tuple("Python")  # ('P', 'y', 't', 'h', 'o', 'n')
```

Acesso a Elementos

O acesso a elementos em tuplas é semelhante ao das listas:

```
pessoa = ("João", 30, "Engenheiro")

print(pessoa[0])    # João
print(pessoa[-1])   # Engenheiro
print(pessoa[1:])   # (30, 'Engenheiro')
```

Métodos de Tuplas

Como as tuplas são imutáveis, elas têm menos métodos que as listas:

```
numeros = (1, 2, 3, 2, 4, 2)

contagem = numeros.count(2)  # Conta ocorrências de 2: 3
indice = numeros.index(3)    # Retorna o índice da primeira ocorrência de 3: 2
```

Desempacotamento de Tuplas

Uma característica útil das tuplas é o desempacotamento, que permite atribuir os elementos de uma tupla a variáveis individuais:

```
coordenadas = (10, 20, 30)
x, y, z = coordenadas

print(x)  # 10
print(y)  # 20
print(z)  # 30

# Troca de valores usando desempacotamento
a, b = 5, 10
a, b = b, a  # Agora a = 10 e b = 5
```

Dicionários

Os dicionários são coleções não ordenadas de pares chave-valor. Eles são otimizados para recuperar valores quando a chave é conhecida.

Criação de Dicionários

```
# Dicionário vazio
dict_vazio = {}

# Dicionário com elementos
pessoa = {
    "nome": "Maria",
    "idade": 25,
    "profissao": "Desenvolvedora"
}

# Dicionário usando a função dict()
pessoa2 = dict(nome="João", idade=30, profissao="Engenheiro")

# Dicionário a partir de uma lista de tuplas
itens = [("a", 1), ("b", 2), ("c", 3)]
dict_de_tuplas = dict(itens)
```

Acesso a Elementos

Os elementos de um dicionário são acessados por suas chaves:

```
pessoa = {
    "nome": "Maria",
    "idade": 25,
    "profissao": "Desenvolvedora"
}

print(pessoa["nome"])      # Maria
print(pessoa.get("idade")) # 25

# Usando get() com valor padrão para chaves inexistentes
print(pessoa.get("cidade", "Não informada"))  # Não informada
```

Métodos de Dicionários

Python oferece vários métodos para manipular dicionários:

```
pessoa = {
    "nome": "Maria",
    "idade": 25,
    "profissao": "Desenvolvedora"
}

# Adicionar ou modificar elementos
pessoa["cidade"] = "São Paulo"  # Adiciona um novo par chave-valor
pessoa["idade"] = 26            # Modifica um valor existente

# Remover elementos
cidade = pessoa.pop("cidade")   # Remove e retorna o valor: 'São Paulo'
del pessoa["profissao"]         # Remove o par chave-valor

# Outras operações
chaves = pessoa.keys()          # Retorna um objeto dict_keys com as chaves
```

```python
valores = pessoa.values()       # Retorna um objeto dict_values com os valores
itens = pessoa.items()          # Retorna um objeto dict_items com pares (chave, valor)

# Verificar se uma chave existe
if "nome" in pessoa:
    print("A chave 'nome' existe no dicionário")

# Atualizar um dicionário com outro
pessoa.update({"cidade": "Rio de Janeiro", "estado": "RJ"})

# Limpar o dicionário
pessoa.clear()  # Remove todos os elementos
```

Dicionários Aninhados

Os dicionários podem conter outros dicionários, criando estruturas hierárquicas:

```python
funcionarios = {
    "F001": {
        "nome": "Ana",
        "departamento": "TI",
        "salario": 5000
    },
    "F002": {
        "nome": "Carlos",
        "departamento": "RH",
        "salario": 4500
    }
}

# Acesso a elementos aninhados
print(funcionarios["F001"]["nome"])  # Ana
print(funcionarios["F002"]["salario"])  # 4500
```

Conjuntos (Sets)

Os conjuntos são coleções não ordenadas de itens únicos. Eles são úteis para operações matemáticas de conjuntos, como união, interseção e diferença.

Criação de Conjuntos

```python
# Conjunto vazio (observe que {} cria um dicionário vazio)
conjunto_vazio = set()

# Conjunto com elementos
frutas = {"maçã", "banana", "laranja"}

# Conjunto usando a função set()
letras = set("abracadabra")  # {'a', 'b', 'c', 'd', 'r'}
```

Métodos de Conjuntos

Python oferece vários métodos para manipular conjuntos:

```python
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# Adicionar e remover elementos
A.add(6)           # Adiciona um elemento: {1, 2, 3, 4, 5, 6}
A.remove(2)        # Remove um elemento (gera erro se não existir): {1, 3, 4, 5, 6}
A.discard(10)      # Remove um elemento (não gera erro se não existir): {1, 3, 4, 5, 6}
elemento = A.pop() # Remove e retorna um elemento arbitrário

# Operações de conjuntos
uniao = A | B      # ou A.union(B): {1, 3, 4, 5, 6, 7, 8}
intersecao = A & B # ou A.intersection(B): {4, 5, 6}
diferenca = A - B  # ou A.difference(B): {1, 3}
dif_simetrica = A ^ B # ou A.symmetric_difference(B): {1, 3, 7, 8}

# Verificar subconjuntos e superconjuntos
C = {4, 5}
print(C.issubset(A))      # True (C é subconjunto de A)
print(A.issuperset(C))    # True (A é superconjunto de C)

# Verificar disjunção
D = {9, 10}
print(A.isdisjoint(D))    # True (A e D não têm elementos em comum)
```

Frozensets

Um `frozenset` é uma versão imutável de um conjunto. Uma vez criado, não pode ser modificado:

```python
fs = frozenset([1, 2, 3, 4])
# fs.add(5)  # Isso geraria um erro, pois frozensets são imutáveis
```

Escolhendo a Estrutura de Dados Adequada

Cada estrutura de dados tem suas próprias características e casos de uso ideais:

- **Listas**: Use quando precisar de uma coleção ordenada que pode ser modificada.
- **Tuplas**: Use quando precisar de uma coleção ordenada que não deve ser modificada.
- **Dicionários**: Use quando precisar mapear chaves para valores.
- **Conjuntos**: Use quando precisar armazenar itens únicos e realizar operações de conjuntos.

Boas Práticas

1. **Escolha a Estrutura Adequada**: Selecione a estrutura de dados que melhor se adapta ao seu problema.
2. **Evite Modificar Coleções Durante Iteração**: Modificar uma coleção enquanto itera sobre ela pode levar a comportamentos inesperados.
3. **Use Compreensões para Código Conciso**: Compreensões de lista, dicionário e conjunto podem tornar o código mais conciso e legível.
4. **Considere o Desempenho**: Diferentes estruturas têm diferentes características de desempenho. Por exemplo, verificar se um elemento está em um conjunto é mais rápido do que em uma lista.
5. **Imutabilidade para Segurança**: Use estruturas imutáveis (tuplas, frozensets) quando os dados não devem ser alterados.

As estruturas de dados são ferramentas fundamentais para qualquer programador Python. Dominar essas estruturas e saber quando usar cada uma delas é essencial para escrever código eficiente e elegante.

No próximo tópico, exploraremos as compreensões de listas, dicionários e conjuntos, que são formas concisas e poderosas de criar e transformar essas estruturas de dados.

Módulo 2: Python Intermediário

Compreensão de Listas

A compreensão de listas (list comprehension) é uma característica poderosa e elegante do Python que permite criar listas de forma concisa. Ela oferece uma sintaxe mais compacta e muitas vezes mais legível do que os loops tradicionais para criar ou transformar listas. Além da compreensão de listas, Python também suporta compreensões de dicionários e conjuntos.

Compreensão de Listas Básica

A sintaxe básica de uma compreensão de lista é:

```
[expressão for item in iterável]
```

Onde: - `expressão` é uma operação aplicada a cada `item` - `item` é a variável que representa cada elemento do `iterável` - `iterável` é uma sequência, como lista, tupla, string, etc.

Exemplo simples: criar uma lista com os quadrados dos números de 0 a 9:

```
# Usando um loop for tradicional
quadrados = []
for i in range(10):
    quadrados.append(i ** 2)
print(quadrados)  # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

# Usando compreensão de lista
quadrados = [i ** 2 for i in range(10)]
print(quadrados)  # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Como podemos ver, a compreensão de lista é mais concisa e, para muitos programadores, mais legível.

Compreensão de Listas com Condicionais

Podemos adicionar condicionais às compreensões de listas para filtrar elementos:

```
[expressão for item in iterável if condição]
```

Exemplo: criar uma lista apenas com os números pares de 0 a 9:

```
# Usando um loop for tradicional
pares = []
for i in range(10):
    if i % 2 == 0:
        pares.append(i)
print(pares)  # [0, 2, 4, 6, 8]
```

```python
# Usando compreensão de lista
pares = [i for i in range(10) if i % 2 == 0]
print(pares)  # [0, 2, 4, 6, 8]
```

Também podemos usar o `if` como parte da expressão, criando uma estrutura condicional:

```
[expressão_if_verdadeiro if condição else expressão_if_falso for item in iterável]
```

Exemplo: criar uma lista que classifica números como "par" ou "ímpar":

```python
classificacao = ["par" if i % 2 == 0 else "ímpar" for i in range(5)]
print(classificacao)  # ['par', 'ímpar', 'par', 'ímpar', 'par']
```

Compreensões de Listas Aninhadas

Podemos aninhar múltiplos loops em uma compreensão de lista:

```
[expressão for item1 in iterável1 for item2 in iterável2]
```

Isso é equivalente a:

```
for item1 in iterável1:
    for item2 in iterável2:
        # expressão
```

Exemplo: criar uma lista com todos os pares (x, y) onde x e y variam de 1 a 3:

```python
# Usando loops for aninhados
pares = []
for x in range(1, 4):
    for y in range(1, 4):
        pares.append((x, y))
print(pares)  # [(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]

# Usando compreensão de lista aninhada
pares = [(x, y) for x in range(1, 4) for y in range(1, 4)]
print(pares)  # [(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]
```

Também podemos adicionar condicionais:

```python
pares = [(x, y) for x in range(1, 4) for y in range(1, 4) if x != y]
print(pares)  # [(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
```

Compreensão de Dicionários

A compreensão de dicionários permite criar dicionários de forma concisa:

```
{chave: valor for item in iterável}
```

Exemplo: criar um dicionário que mapeia números para seus quadrados:

```python
# Usando um loop for tradicional
quadrados = {}
for i in range(5):
    quadrados[i] = i ** 2
print(quadrados)  # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

# Usando compreensão de dicionário
```

```python
quadrados = {i: i ** 2 for i in range(5)}
print(quadrados)  # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Podemos adicionar condicionais:

```python
# Apenas números pares
quadrados_pares = {i: i ** 2 for i in range(10) if i % 2 == 0}
print(quadrados_pares)  # {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}
```

Compreensão de Conjuntos

A compreensão de conjuntos é semelhante à compreensão de listas, mas cria um conjunto (elementos únicos):

```
{expressão for item in iterável}
```

Exemplo: criar um conjunto com os quadrados dos números de 0 a 9:

```python
# Usando um loop for tradicional
quadrados = set()
for i in range(10):
    quadrados.add(i ** 2)
print(quadrados)  # {0, 1, 4, 9, 16, 25, 36, 49, 64, 81}

# Usando compreensão de conjunto
quadrados = {i ** 2 for i in range(10)}
print(quadrados)  # {0, 1, 4, 9, 16, 25, 36, 49, 64, 81}
```

Expressões Geradoras

As expressões geradoras são semelhantes às compreensões de listas, mas não criam a lista inteira na memória de uma vez. Em vez disso, elas geram os elementos sob demanda, o que é mais eficiente para grandes conjuntos de dados:

```
(expressão for item in iterável)
```

Exemplo:

```python
# Compreensão de lista (cria toda a lista na memória)
quadrados_lista = [i ** 2 for i in range(10)]

# Expressão geradora (gera elementos sob demanda)
quadrados_gerador = (i ** 2 for i in range(10))

print(quadrados_lista)      # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
print(quadrados_gerador)    # <generator object <genexpr> at 0x...>

# Para ver os elementos do gerador, precisamos iterar sobre ele
for q in quadrados_gerador:
    print(q, end=" ")  # 0 1 4 9 16 25 36 49 64 81
```

As expressões geradoras são especialmente úteis quando trabalhamos com grandes conjuntos de dados ou quando não precisamos de todos os elementos de uma vez.

Casos de Uso Comuns

As compreensões são particularmente úteis para:

1. **Transformação de dados**: Aplicar uma função a cada elemento de uma sequência.

```
# Converter strings para maiúsculas
palavras = ["python", "é", "incrível"]
maiusculas = [palavra.upper() for palavra in palavras]
print(maiusculas)  # ['PYTHON', 'É', 'INCRÍVEL']
```

2. **Filtragem de dados**: Selecionar elementos que atendem a determinados critérios.

```
# Filtrar números divisíveis por 3
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9]
divisiveis_por_3 = [num for num in numeros if num % 3 == 0]
print(divisiveis_por_3)  # [3, 6, 9]
```

3. **Extração de dados**: Extrair informações específicas de estruturas complexas.

```
# Extrair nomes de uma lista de dicionários
pessoas = [
    {"nome": "Ana", "idade": 25},
    {"nome": "Bruno", "idade": 30},
    {"nome": "Carla", "idade": 35}
]
nomes = [pessoa["nome"] for pessoa in pessoas]
print(nomes)  # ['Ana', 'Bruno', 'Carla']
```

4. **Achatamento de listas aninhadas**: Transformar uma lista de listas em uma única lista.

```
# Achatar uma lista de listas
matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
achatada = [num for linha in matriz for num in linha]
print(achatada)  # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Boas Práticas

1. **Legibilidade**: Use compreensões quando elas tornam o código mais legível. Para operações complexas, loops tradicionais podem ser mais claros.
2. **Evite Compreensões Muito Complexas**: Se uma compreensão se tornar muito complexa, considere dividi-la em partes menores ou usar loops tradicionais.
3. **Considere o Desempenho**: Para grandes conjuntos de dados, expressões geradoras podem ser mais eficientes do que compreensões de listas.
4. **Evite Efeitos Colaterais**: Compreensões devem ser usadas principalmente para criar novas coleções, não para efeitos colaterais como imprimir valores.

As compreensões são uma característica poderosa do Python que pode tornar seu código mais conciso e elegante. Dominar as compreensões de listas, dicionários e conjuntos é uma habilidade valiosa para qualquer programador Python.

No próximo tópico, exploraremos a manipulação de arquivos em Python, que é essencial para ler e escrever dados em arquivos externos.

Módulo 2: Python Intermediário

Manipulação de Arquivos

A manipulação de arquivos é uma habilidade essencial para qualquer programador, permitindo que seus programas leiam dados de entrada e salvem resultados de forma permanente. Python oferece funções e métodos simples e poderosos para trabalhar com arquivos.

Abrindo e Fechando Arquivos

A função `open()` é usada para abrir arquivos em Python. Ela retorna um objeto de arquivo que fornece métodos para trabalhar com o arquivo.

```
# Sintaxe básica
file = open(nome_do_arquivo, modo)
```

Os modos mais comuns são: - `'r'`: Leitura (padrão) - `'w'`: Escrita (cria um novo arquivo ou sobrescreve um existente) - `'a'`: Anexar (append) - adiciona conteúdo ao final do arquivo - `'b'`: Modo binário (usado com outros modos, ex: `'rb'` para leitura binária) - `'t'`: Modo texto (padrão, usado com outros modos) - `'+'`: Atualização (leitura e escrita)

Após trabalhar com um arquivo, é importante fechá-lo usando o método `close()`:

```
file = open("exemplo.txt", "r")
# Operações com o arquivo
file.close()
```

Usando o Gerenciador de Contexto (with)

A maneira recomendada de trabalhar com arquivos em Python é usar o gerenciador de contexto `with`, que garante que o arquivo seja fechado corretamente, mesmo se ocorrerem exceções:

```
with open("exemplo.txt", "r") as file:
    # Operações com o arquivo
    conteudo = file.read()
    print(conteudo)
# O arquivo é automaticamente fechado quando saímos do bloco with
```

Lendo Arquivos

Python oferece vários métodos para ler o conteúdo de arquivos:

Lendo o Arquivo Inteiro

```
with open("exemplo.txt", "r") as file:
    conteudo = file.read()  # Lê todo o conteúdo do arquivo como uma string
    print(conteudo)
```

Lendo Linha por Linha

```
with open("exemplo.txt", "r") as file:
    linha = file.readline()  # Lê uma linha
    print(linha)

    # Ou usando um loop para ler todas as linhas
    for linha in file:
        print(linha, end="")  # end="" evita dupla quebra de linha
```

Lendo Todas as Linhas em uma Lista

```
with open("exemplo.txt", "r") as file:
    linhas = file.readlines()  # Retorna uma lista de strings, uma para cada linha
    for linha in linhas:
        print(linha, end="")
```

Escrevendo em Arquivos

Para escrever em arquivos, usamos os modos `'w'` (sobrescreve) ou `'a'` (anexa):

Escrevendo Texto

```
with open("saida.txt", "w") as file:
    file.write("Olá, mundo!\n")  # \n para quebra de linha
    file.write("Esta é a segunda linha.")
```

Escrevendo Múltiplas Linhas

```
linhas = ["Primeira linha\n", "Segunda linha\n", "Terceira linha\n"]

with open("saida.txt", "w") as file:
    file.writelines(linhas)  # writelines não adiciona quebras de linha automaticamente
```

Anexando a um Arquivo Existente

```
with open("saida.txt", "a") as file:
    file.write("\nEsta linha será adicionada ao final do arquivo.")
```

Trabalhando com Caminhos de Arquivo

O módulo `os.path` fornece funções para trabalhar com caminhos de arquivo de forma independente do sistema operacional:

```
import os.path

# Verificar se um arquivo existe
if os.path.exists("arquivo.txt"):
```

```
    print("O arquivo existe")

# Obter o diretório atual
diretorio_atual = os.getcwd()
print(diretorio_atual)

# Juntar partes de um caminho
caminho_completo = os.path.join(diretorio_atual, "dados", "arquivo.txt")
print(caminho_completo)

# Obter o nome do arquivo a partir de um caminho
nome_arquivo = os.path.basename(caminho_completo)
print(nome_arquivo)  # arquivo.txt

# Obter o diretório a partir de um caminho
diretorio = os.path.dirname(caminho_completo)
print(diretorio)

# Dividir um caminho em diretório e nome de arquivo
diretorio, nome_arquivo = os.path.split(caminho_completo)
print(diretorio, nome_arquivo)

# Verificar se um caminho é um arquivo ou diretório
print(os.path.isfile(caminho_completo))
print(os.path.isdir(diretorio))
```

Trabalhando com Diretórios

O módulo os fornece funções para trabalhar com diretórios:

```
import os

# Listar arquivos e diretórios
arquivos = os.listdir(".")  # "." representa o diretório atual
print(arquivos)

# Criar um diretório
os.mkdir("novo_diretorio")

# Criar diretórios recursivamente (incluindo diretórios pais)
os.makedirs("dir1/dir2/dir3")

# Remover um arquivo
os.remove("arquivo_para_remover.txt")

# Remover um diretório vazio
os.rmdir("diretorio_vazio")

# Renomear um arquivo ou diretório
os.rename("antigo.txt", "novo.txt")
```

Trabalhando com Arquivos CSV

CSV (Comma-Separated Values) é um formato comum para armazenar dados tabulares. Python fornece o módulo `csv` para trabalhar com arquivos CSV:

```
import csv

# Lendo um arquivo CSV
with open("dados.csv", "r") as file:
    leitor = csv.reader(file)
    for linha in leitor:
        print(linha)  # linha é uma lista de strings

# Escrevendo em um arquivo CSV
dados = [
    ["Nome", "Idade", "Cidade"],
    ["Ana", "25", "São Paulo"],
    ["Bruno", "30", "Rio de Janeiro"],
    ["Carla", "35", "Belo Horizonte"]
]

with open("pessoas.csv", "w", newline="") as file:
    escritor = csv.writer(file)
    escritor.writerows(dados)  # Escreve todas as linhas de uma vez

# Usando DictReader e DictWriter para trabalhar com dicionários
with open("pessoas.csv", "r") as file:
    leitor = csv.DictReader(file)  # Assume que a primeira linha contém os nomes das colunas
    for linha in leitor:
        print(linha)  # linha é um dicionário

dados_dict = [
    {"Nome": "Ana", "Idade": "25", "Cidade": "São Paulo"},
    {"Nome": "Bruno", "Idade": "30", "Cidade": "Rio de Janeiro"},
    {"Nome": "Carla", "Idade": "35", "Cidade": "Belo Horizonte"}
]

with open("pessoas_dict.csv", "w", newline="") as file:
    campos = ["Nome", "Idade", "Cidade"]
    escritor = csv.DictWriter(file, fieldnames=campos)
    escritor.writeheader()  # Escreve a linha de cabeçalho
    escritor.writerows(dados_dict)  # Escreve todas as linhas de uma vez
```

Trabalhando com Arquivos JSON

JSON (JavaScript Object Notation) é um formato leve de intercâmbio de dados. Python fornece o módulo `json` para trabalhar com arquivos JSON:

```
import json

# Convertendo um dicionário Python para JSON
dados = {
    "nome": "Ana",
    "idade": 25,
    "cidade": "São Paulo",
```

```
    "interesses": ["programação", "música", "viagens"],
    "ativo": True
}

# Escrevendo JSON em um arquivo
with open("dados.json", "w") as file:
    json.dump(dados, file, indent=4)  # indent para formatação bonita

# Convertendo JSON para um dicionário Python
with open("dados.json", "r") as file:
    dados_carregados = json.load(file)
    print(dados_carregados)
    print(dados_carregados["nome"])  # Ana
    print(dados_carregados["interesses"][0])  # programação
```

Trabalhando com Arquivos Binários

Para trabalhar com arquivos binários, usamos os modos `'rb'` (leitura binária) e `'wb'` (escrita binária):

```
# Lendo um arquivo binário
with open("imagem.jpg", "rb") as file:
    dados = file.read()
    print(len(dados), "bytes")

# Escrevendo em um arquivo binário
with open("copia.jpg", "wb") as file:
    file.write(dados)
```

Usando o Módulo pathlib

O módulo `pathlib` introduzido no Python 3.4 oferece uma abordagem orientada a objetos para trabalhar com caminhos de arquivo:

```
from pathlib import Path

# Criando objetos Path
caminho = Path("dados/arquivo.txt")
diretorio = Path("dados")

# Verificando se um arquivo ou diretório existe
print(caminho.exists())
print(diretorio.exists())

# Criando diretórios
diretorio.mkdir(exist_ok=True)  # não gera erro se o diretório já existir
Path("dir1/dir2/dir3").mkdir(parents=True, exist_ok=True)  # cria diretórios pais

# Listando arquivos em um diretório
for arquivo in Path(".").glob("*.py"):  # lista todos os arquivos .py no diretório atual
    print(arquivo)

# Lendo e escrevendo arquivos
texto = caminho.read_text()  # lê todo o conteúdo como texto
caminho.write_text("Novo conteúdo")  # escreve texto no arquivo
```

```python
# Obtendo informações sobre o caminho
print(caminho.name)  # arquivo.txt
print(caminho.stem)  # arquivo
print(caminho.suffix)  # .txt
print(caminho.parent)  # dados
```

Boas Práticas

1. **Use o Gerenciador de Contexto**: Sempre use `with` ao trabalhar com arquivos para garantir que eles sejam fechados corretamente.
2. **Trate Exceções**: Ao trabalhar com arquivos, esteja preparado para lidar com exceções como `FileNotFoundError`, `PermissionError`, etc.

```python
try:
    with open("arquivo_inexistente.txt", "r") as file:
        conteudo = file.read()
except FileNotFoundError:
    print("O arquivo não foi encontrado.")
except PermissionError:
    print("Sem permissão para acessar o arquivo.")
except Exception as e:
    print(f"Ocorreu um erro: {e}")
```

3. **Use Caminhos Absolutos ou Relativos Corretamente**: Entenda a diferença entre caminhos absolutos e relativos e use-os adequadamente.
4. **Considere a Codificação de Caracteres**: Ao trabalhar com arquivos de texto, especifique a codificação correta para evitar problemas com caracteres especiais.

```python
with open("arquivo.txt", "r", encoding="utf-8") as file:
    conteudo = file.read()
```

5. **Feche os Arquivos**: Se não estiver usando o gerenciador de contexto `with`, lembre-se de fechar os arquivos explicitamente com `file.close()`.
6. **Use o Módulo Apropriado**: Para formatos específicos como CSV, JSON, XML, etc., use os módulos específicos em vez de tentar analisar o texto manualmente.

A manipulação de arquivos é uma habilidade fundamental que permite que seus programas interajam com o sistema de arquivos, leiam dados de entrada e salvem resultados. Dominar essas técnicas ampliará significativamente as capacidades dos seus programas Python.

No próximo tópico, exploraremos módulos e pacotes em Python, que permitem organizar e reutilizar código de forma eficiente.

Módulo 2: Python Intermediário

Módulos e Pacotes

Os módulos e pacotes são fundamentais para organizar, reutilizar e compartilhar código em Python. Eles permitem dividir programas grandes em componentes menores e mais gerenciáveis, facilitando a manutenção e o desenvolvimento colaborativo.

Módulos

Um módulo em Python é simplesmente um arquivo Python (`.py`) que contém definições de funções, classes e variáveis que podem ser importadas e usadas em outros programas Python.

Criando um Módulo

Vamos criar um módulo simples chamado `matematica.py`:

```python
# matematica.py

# Variáveis
pi = 3.14159

# Funções
def soma(a, b):
    """Retorna a soma de dois números."""
    return a + b

def subtracao(a, b):
    """Retorna a diferença entre dois números."""
    return a - b

def multiplicacao(a, b):
    """Retorna o produto de dois números."""
    return a * b

def divisao(a, b):
    """Retorna o quociente de dois números."""
    if b == 0:
        raise ValueError("Divisão por zero não é permitida")
    return a / b

# Classe
class Calculadora:
    """Uma classe simples de calculadora."""
```

```
    def __init__(self):
        self.resultado = 0

    def adicionar(self, valor):
        self.resultado += valor
        return self.resultado

    def subtrair(self, valor):
        self.resultado -= valor
        return self.resultado

    def resetar(self):
        self.resultado = 0
        return self.resultado
```

Importando Módulos

Existem várias maneiras de importar módulos em Python:

1. **Importar o módulo inteiro**:

```
import matematica

# Usar funções e variáveis do módulo
print(matematica.pi)  # 3.14159
print(matematica.soma(5, 3))  # 8

# Criar uma instância da classe
calc = matematica.Calculadora()
print(calc.adicionar(10))  # 10
```

2. **Importar itens específicos do módulo**:

```
from matematica import soma, pi

# Usar diretamente, sem o prefixo do módulo
print(pi)  # 3.14159
print(soma(5, 3))  # 8
```

3. **Importar todos os itens do módulo**:

```
from matematica import *

# Usar diretamente, sem o prefixo do módulo
print(pi)  # 3.14159
print(soma(5, 3))  # 8
```

Nota: Importar tudo com * geralmente não é recomendado, pois pode causar conflitos de nomes e tornar o código menos legível.

4. **Importar com um alias**:

```
import matematica as mat

print(mat.pi)  # 3.14159
print(mat.soma(5, 3))  # 8
```

Módulos Integrados

Python vem com uma biblioteca padrão rica em módulos integrados. Alguns exemplos comuns:

```
# Módulo math para funções matemáticas
import math
print(math.sqrt(16))  # 4.0
print(math.sin(math.pi/2))  # 1.0

# Módulo random para geração de números aleatórios
import random
print(random.randint(1, 10))  # Número aleatório entre 1 e 10
print(random.choice(["maçã", "banana", "laranja"]))  # Escolhe um item aleatório

# Módulo datetime para manipulação de datas e horas
import datetime
agora = datetime.datetime.now()
print(agora)  # Data e hora atuais
print(agora.year)  # Ano atual
```

Localização de Módulos

Quando você importa um módulo, Python procura por ele em várias localizações, na seguinte ordem:

1. O diretório atual
2. Os diretórios listados na variável de ambiente `PYTHONPATH`
3. Os diretórios de instalação padrão do Python

Você pode verificar todos os diretórios onde Python procura por módulos usando:

```
import sys
print(sys.path)
```

Pacotes

Um pacote é uma forma de organizar módulos relacionados em uma estrutura de diretórios hierárquica. Essencialmente, um pacote é um diretório que contém um arquivo especial chamado `__init__.py` e pode conter outros módulos ou subpacotes.

Criando um Pacote

Vamos criar um pacote simples chamado `matematica_avancada`:

```
matematica_avancada/
    __init__.py
    algebra.py
    geometria.py
    estatistica/
        __init__.py
        descritiva.py
        inferencial.py
```

O arquivo `__init__.py` pode estar vazio ou conter código de inicialização para o pacote. Ele é necessário para que Python reconheça o diretório como um pacote.

Exemplo de conteúdo para os arquivos:

```
# matematica_avancada/__init__.py
print("Pacote matematica_avancada importado")
```

```
# matematica_avancada/algebra.py
def resolver_equacao_linear(a, b):
    """Resolve a equação ax + b = 0."""
    if a == 0:
        raise ValueError("Coeficiente 'a' não pode ser zero")
    return -b / a

# matematica_avancada/geometria.py
import math

def area_circulo(raio):
    """Calcula a área de um círculo."""
    return math.pi * raio ** 2

def area_retangulo(largura, altura):
    """Calcula a área de um retângulo."""
    return largura * altura

# matematica_avancada/estatistica/__init__.py
print("Subpacote estatistica importado")

# matematica_avancada/estatistica/descritiva.py
def media(valores):
    """Calcula a média aritmética de uma lista de valores."""
    return sum(valores) / len(valores)

def mediana(valores):
    """Calcula a mediana de uma lista de valores."""
    valores_ordenados = sorted(valores)
    n = len(valores)
    if n % 2 == 0:
        return (valores_ordenados[n//2 - 1] + valores_ordenados[n//2]) / 2
    else:
        return valores_ordenados[n//2]

# matematica_avancada/estatistica/inferencial.py
import math

def erro_padrao(desvio_padrao, tamanho_amostra):
    """Calcula o erro padrão da média."""
    return desvio_padrao / math.sqrt(tamanho_amostra)
```

Importando de Pacotes

Existem várias maneiras de importar de pacotes:

1. **Importar um módulo específico do pacote**:

```
import matematica_avancada.geometria

area = matematica_avancada.geometria.area_circulo(5)
print(area)  # 78.53981633974483
```

2. **Importar funções específicas de um módulo**:

```
from matematica_avancada.geometria import area_circulo, area_retangulo

print(area_circulo(5))  # 78.53981633974483
print(area_retangulo(4, 6))  # 24
```

3. **Importar um subpacote**:

```
import matematica_avancada.estatistica

# Agora você precisa importar os módulos do subpacote
from matematica_avancada.estatistica import descritiva

print(descritiva.media([1, 2, 3, 4, 5]))  # 3.0
```

4. **Importar com alias**:

```
import matematica_avancada.geometria as geo

print(geo.area_circulo(5))  # 78.53981633974483
```

Módulo `if __name__ == "__main__"`

Uma prática comum em Python é incluir código que será executado apenas quando o módulo for executado diretamente, não quando for importado. Isso é feito usando a verificação `if __name__ == "__main__"`:

```
# exemplo.py

def funcao_util():
    """Uma função útil que pode ser importada."""
    return "Esta função faz algo útil"

# Este código será executado apenas se o arquivo for executado diretamente
if __name__ == "__main__":
    print("Este módulo está sendo executado diretamente")
    print(funcao_util())
```

Se você importar este módulo, o código dentro do bloco `if` não será executado:

```
import exemplo
print(exemplo.funcao_util())  # "Esta função faz algo útil"
# O código dentro do bloco if não é executado
```

Mas se você executar o arquivo diretamente (`python exemplo.py`), o código dentro do bloco `if` será executado.

Recarregando Módulos

Se você modificar um módulo após importá-lo, as alterações não serão refletidas automaticamente. Você pode recarregar um módulo usando a função `reload()` do módulo `importlib`:

```
import matematica
print(matematica.soma(5, 3))  # 8

# Suponha que você modificou o arquivo matematica.py
import importlib
importlib.reload(matematica)
```

Instalando Pacotes Externos

Python tem um vasto ecossistema de pacotes de terceiros que podem ser instalados usando o gerenciador de pacotes `pip`:

```
# Instalar um pacote
pip install nome_do_pacote

# Instalar uma versão específica
pip install nome_do_pacote==1.2.3

# Atualizar um pacote
pip upgrade nome_do_pacote

# Desinstalar um pacote
pip uninstall nome_do_pacote

# Listar pacotes instalados
pip list
```

Alguns pacotes populares incluem: - `numpy`: Para computação numérica - `pandas`: Para análise de dados - `matplotlib`: Para visualização de dados - `requests`: Para fazer requisições HTTP - `flask` e `django`: Para desenvolvimento web - `scikit-learn`: Para aprendizado de máquina

Ambientes Virtuais

Os ambientes virtuais são uma ferramenta essencial para gerenciar dependências de projetos Python. Eles permitem criar ambientes isolados onde você pode instalar pacotes específicos para um projeto, sem afetar outros projetos.

Criando e Usando Ambientes Virtuais

```
# Criar um ambiente virtual
python -m venv meu_ambiente

# Ativar o ambiente virtual
# No Windows
meu_ambiente\Scripts\activate
# No macOS/Linux
source meu_ambiente/bin/activate

# Desativar o ambiente virtual
deactivate
```

Gerenciando Dependências

Você pode salvar as dependências do seu projeto em um arquivo `requirements.txt`:

```
# Gerar requirements.txt
pip freeze > requirements.txt

# Instalar dependências de um requirements.txt
pip install -r requirements.txt
```

Boas Práticas

1. **Organização de Código**: Use módulos e pacotes para organizar seu código de forma lógica e modular.
2. **Importações Específicas**: Prefira importar apenas o que você precisa (`from modulo import item`) em vez de importar tudo (`from modulo import *`).
3. **Docstrings**: Documente seus módulos, funções e classes com docstrings para facilitar o uso por outros desenvolvedores.
4. **Ambientes Virtuais**: Use ambientes virtuais para isolar as dependências de diferentes projetos.
5. **Estrutura de Pacotes**: Mantenha uma estrutura de pacotes clara e intuitiva, agrupando funcionalidades relacionadas.
6. **Evite Importações Circulares**: Tenha cuidado com importações circulares, onde o módulo A importa o módulo B, que por sua vez importa o módulo A.
7. **Nomes Significativos**: Use nomes significativos para seus módulos e pacotes, que reflitam seu conteúdo e propósito.

Os módulos e pacotes são ferramentas poderosas para organizar e reutilizar código em Python. Dominar seu uso é essencial para desenvolver aplicações Python escaláveis e bem estruturadas.

No próximo tópico, exploraremos o tratamento de exceções em Python, que permite lidar com erros e situações inesperadas de forma elegante e robusta.

Módulo 2: Python Intermediário

Tratamento de Exceções

O tratamento de exceções é uma parte fundamental da programação em Python, permitindo que os programas lidem com erros e situações inesperadas de forma elegante e controlada. Em vez de falhar abruptamente quando ocorre um erro, um programa com tratamento de exceções adequado pode identificar o problema, informar o usuário e, possivelmente, tomar medidas corretivas.

O que são Exceções?

Exceções são eventos que ocorrem durante a execução de um programa e interrompem o fluxo normal de instruções. Em Python, exceções são objetos que representam erros.

Quando uma exceção ocorre, dizemos que ela foi "lançada" ou "levantada" (raised). Se a exceção não for tratada, o programa será encerrado e uma mensagem de erro será exibida.

Exceções Comuns em Python

Python tem vários tipos de exceções integradas. Algumas das mais comuns são:

- `SyntaxError`: Erro de sintaxe no código
- `NameError`: Tentativa de usar uma variável ou função não definida
- `TypeError`: Operação aplicada a um objeto de tipo inadequado
- `ValueError`: Operação com valor inadequado
- `IndexError`: Tentativa de acessar um índice inválido em uma sequência
- `KeyError`: Tentativa de acessar uma chave inexistente em um dicionário
- `FileNotFoundError`: Tentativa de abrir um arquivo inexistente
- `ZeroDivisionError`: Tentativa de dividir por zero
- `ImportError`: Falha ao importar um módulo
- `AttributeError`: Tentativa de acessar um atributo inexistente

Estrutura Básica de Tratamento de Exceções

A estrutura básica para tratamento de exceções em Python usa os blocos `try`, `except`, `else` e `finally`:

```
try:
    # Código que pode gerar uma exceção
    resultado = 10 / 0
except ZeroDivisionError:
    # Código executado se ocorrer uma ZeroDivisionError
    print("Erro: Divisão por zero!")
else:
    # Código executado se nenhuma exceção ocorrer no bloco try
    print(f"O resultado é {resultado}")
finally:
```

```
    # Código executado sempre, independentemente de ocorrer exceção ou não
    print("Operação finalizada")
```

Capturando Múltiplas Exceções

Você pode capturar múltiplas exceções de diferentes maneiras:

Múltiplos blocos except

```
try:
    numero = int(input("Digite um número: "))
    resultado = 10 / numero
    print(f"O resultado é {resultado}")
except ValueError:
    print("Erro: Entrada inválida. Por favor, digite um número.")
except ZeroDivisionError:
    print("Erro: Divisão por zero não é permitida.")
```

Capturando múltiplas exceções em um único bloco except

```
try:
    numero = int(input("Digite um número: "))
    resultado = 10 / numero
    print(f"O resultado é {resultado}")
except (ValueError, ZeroDivisionError):
    print("Erro: Entrada inválida ou divisão por zero.")
```

Capturando Todas as Exceções

Você pode capturar todas as exceções usando `except` sem especificar um tipo de exceção, mas isso geralmente não é recomendado, pois pode mascarar erros inesperados:

```
try:
    # Algum código
    pass
except:  # Não recomendado
    print("Ocorreu um erro")
```

Uma abordagem melhor é capturar `Exception`, que é a classe base para a maioria das exceções:

```
try:
    # Algum código
    pass
except Exception as e:
    print(f"Ocorreu um erro: {e}")
```

Obtendo Informações sobre a Exceção

Você pode obter informações detalhadas sobre a exceção usando a cláusula `as`:

```
try:
    with open("arquivo_inexistente.txt", "r") as arquivo:
        conteudo = arquivo.read()
except FileNotFoundError as e:
```

```
    print(f"Erro: {e}")
    print(f"Tipo de erro: {type(e).__name__}")
```

Lançando Exceções

Você pode lançar exceções explicitamente usando a instrução `raise`:

```
def dividir(a, b):
    if b == 0:
        raise ValueError("Divisão por zero não é permitida")
    return a / b

try:
    resultado = dividir(10, 0)
except ValueError as e:
    print(f"Erro: {e}")
```

Criando Exceções Personalizadas

Você pode criar suas próprias exceções personalizadas herdando de `Exception` ou de uma de suas subclasses:

```
class ValorNegativoError(Exception):
    """Exceção lançada quando um valor negativo é fornecido onde não é permitido."""
    pass

def calcular_raiz_quadrada(numero):
    if numero < 0:
        raise ValorNegativoError("Não é possível calcular a raiz quadrada de um número negativo")
    return numero ** 0.5

try:
    resultado = calcular_raiz_quadrada(-5)
except ValorNegativoError as e:
    print(f"Erro: {e}")
```

Relançando Exceções

Às vezes, você pode querer capturar uma exceção, fazer algo com ela (como registrar o erro) e depois relançá-la para ser tratada em outro lugar:

```
try:
    # Algum código que pode gerar uma exceção
    resultado = 10 / 0
except ZeroDivisionError:
    print("Registrando erro de divisão por zero")
    # Relança a exceção
    raise
```

Usando else e finally

O bloco `else` é executado se nenhuma exceção for lançada no bloco `try`. O bloco `finally` é sempre executado, independentemente de ocorrer uma exceção ou não:

```
try:
    arquivo = open("dados.txt", "r")
```

```
    conteudo = arquivo.read()
except FileNotFoundError:
    print("O arquivo não foi encontrado")
else:
    print(f"O arquivo tem {len(conteudo)} caracteres")
    arquivo.close()
finally:
    print("Operação de leitura finalizada")
```

O bloco `finally` é especialmente útil para garantir que recursos sejam liberados, como fechar arquivos ou conexões de banco de dados, mesmo se ocorrerem exceções.

Exceções em Contexto (with)

O gerenciador de contexto `with` é uma forma elegante de garantir que recursos sejam liberados corretamente, mesmo em caso de exceções:

```
try:
    with open("dados.txt", "r") as arquivo:
        conteudo = arquivo.read()
        # Se ocorrer uma exceção aqui, o arquivo ainda será fechado
except FileNotFoundError:
    print("O arquivo não foi encontrado")
```

Encadeamento de Exceções

A partir do Python 3, você pode encadear exceções usando a sintaxe `raise ... from ...`:

```
try:
    numero = int("abc")
except ValueError as e:
    raise RuntimeError("Falha ao converter para inteiro") from e
```

Isso preserva a exceção original como a "causa" da nova exceção, facilitando a depuração.

Assertions

As asserções são uma forma de verificar se uma condição é verdadeira e lançar uma exceção se não for:

```
def dividir(a, b):
    assert b != 0, "Divisor não pode ser zero"
    return a / b

try:
    resultado = dividir(10, 0)
except AssertionError as e:
    print(f"Erro: {e}")
```

As asserções são principalmente usadas para depuração e testes, não para tratamento de erros em produção.

Boas Práticas no Tratamento de Exceções

1. **Seja Específico**: Capture apenas as exceções que você espera e sabe como tratar. Evite capturar `Exception` ou usar `except` sem especificar um tipo de exceção, a menos que seja realmente necessário.
2. **Mantenha os Blocos try Pequenos**: Coloque apenas o código que pode gerar exceções dentro do bloco `try`. Isso torna mais claro qual parte do código está sendo protegida.

3. **Use finally para Limpeza**: Use o bloco `finally` para garantir que recursos sejam liberados, independentemente de ocorrerem exceções.
4. **Prefira Gerenciadores de Contexto**: Use `with` sempre que possível para recursos que precisam ser fechados ou liberados.
5. **Documente Exceções**: Documente as exceções que sua função pode lançar, para que os usuários da função saibam o que esperar.
6. **Crie Exceções Personalizadas Significativas**: Ao criar exceções personalizadas, dê a elas nomes significativos e forneça mensagens de erro úteis.
7. **Não Ignore Exceções**: Evite blocos `except` vazios ou que apenas passam (`pass`). Se você capturar uma exceção, faça algo útil com ela.
8. **Registre Exceções**: Em aplicações de produção, registre exceções com detalhes suficientes para depuração, incluindo rastreamentos de pilha.

Exemplo Completo

Vamos ver um exemplo completo que incorpora várias das técnicas discutidas:

```
import logging

# Configurar logging
logging.basicConfig(
    filename='app.log',
    level=logging.ERROR,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)

class ValorInvalidoError(Exception):
    """Exceção lançada para erros de valor inválido."""
    pass

def ler_numero_do_arquivo(nome_arquivo):
    """
    Lê um número de um arquivo.

    Args:
        nome_arquivo (str): O nome do arquivo a ser lido.

    Returns:
        float: O número lido do arquivo.

    Raises:
        FileNotFoundError: Se o arquivo não existir.
        ValorInvalidoError: Se o conteúdo do arquivo não for um número válido.
    """
    try:
        with open(nome_arquivo, 'r') as arquivo:
            conteudo = arquivo.read().strip()

        try:
            numero = float(conteudo)
            if numero < 0:
                raise ValorInvalidoError("Número negativo não é permitido")
```

```
            return numero
        except ValueError:
            raise ValorInvalidoError(f"'{conteudo}' não é um número válido")
    except FileNotFoundError:
        logging.error(f"Arquivo não encontrado: {nome_arquivo}")
        raise  # Relança a exceção

def calcular_raiz_quadrada(numero):
    """
    Calcula a raiz quadrada de um número.

    Args:
        numero (float): O número para calcular a raiz quadrada.

    Returns:
        float: A raiz quadrada do número.

    Raises:
        ValorInvalidoError: Se o número for negativo.
    """
    if numero < 0:
        raise ValorInvalidoError("Não é possível calcular a raiz quadrada de um número negativo")
    return numero ** 0.5

def main():
    """Função principal do programa."""
    nome_arquivo = input("Digite o nome do arquivo: ")

    try:
        numero = ler_numero_do_arquivo(nome_arquivo)
        resultado = calcular_raiz_quadrada(numero)
        print(f"A raiz quadrada de {numero} é {resultado:.2f}")
    except FileNotFoundError:
        print(f"Erro: O arquivo '{nome_arquivo}' não foi encontrado.")
    except ValorInvalidoError as e:
        print(f"Erro de valor: {e}")
    except Exception as e:
        print(f"Ocorreu um erro inesperado: {e}")
        logging.exception("Erro inesperado")
    else:
        print("Operação concluída com sucesso!")
    finally:
        print("Programa finalizado")

if __name__ == "__main__":
    main()
```

O tratamento de exceções é uma habilidade essencial para escrever código Python robusto e confiável. Ao dominar essas técnicas, você poderá criar programas que lidam graciosamente com erros e situações inesperadas, proporcionando uma melhor experiência para os usuários e facilitando a manutenção e depuração.

No próximo tópico, exploraremos a programação orientada a objetos em Python, um paradigma poderoso que permite organizar código em torno de objetos que combinam dados e comportamento.

Módulo 2: Python Intermediário

Programação Orientada a Objetos

A Programação Orientada a Objetos (POO) é um paradigma de programação que organiza o código em torno de objetos, que são instâncias de classes. Este paradigma permite modelar entidades do mundo real de forma mais natural e intuitiva, facilitando o desenvolvimento de sistemas complexos. Python é uma linguagem multiparadigma que suporta plenamente a programação orientada a objetos.

Conceitos Fundamentais da POO

Classes e Objetos

Uma **classe** é um modelo ou blueprint que define as características (atributos) e comportamentos (métodos) que os objetos dessa classe terão. Um **objeto** é uma instância de uma classe.

```
# Definindo uma classe
class Carro:
    # Método inicializador (construtor)
    def __init__(self, marca, modelo, ano):
        # Atributos de instância
        self.marca = marca
        self.modelo = modelo
        self.ano = ano
        self.velocidade = 0

    # Métodos
    def acelerar(self, incremento):
        self.velocidade += incremento
        print(f"O carro acelerou para {self.velocidade} km/h")

    def frear(self, decremento):
        if self.velocidade >= decremento:
            self.velocidade -= decremento
        else:
            self.velocidade = 0
        print(f"O carro freou para {self.velocidade} km/h")

    def buzinar(self):
        print("Bi bi!")

# Criando objetos (instâncias da classe)
meu_carro = Carro("Toyota", "Corolla", 2020)
seu_carro = Carro("Honda", "Civic", 2019)
```

```python
# Acessando atributos
print(f"Meu carro é um {meu_carro.marca} {meu_carro.modelo} de {meu_carro.ano}")

# Chamando métodos
meu_carro.acelerar(30)
meu_carro.buzinar()
meu_carro.frear(10)
```

Atributos e Métodos

Atributos são variáveis que pertencem a uma classe ou objeto. Em Python, existem diferentes tipos de atributos:

1. **Atributos de instância**: Pertencem a cada objeto individualmente e são definidos no método `__init__`.
2. **Atributos de classe**: Compartilhados por todas as instâncias da classe.

Métodos são funções que pertencem a uma classe. Existem diferentes tipos de métodos:

1. **Métodos de instância**: Operam em uma instância específica da classe.
2. **Métodos de classe**: Operam na classe como um todo, não em instâncias específicas.
3. **Métodos estáticos**: Não operam nem na classe nem em instâncias, mas estão logicamente relacionados à classe.

```python
class Pessoa:
    # Atributo de classe
    especie = "Homo sapiens"

    # Método inicializador
    def __init__(self, nome, idade):
        # Atributos de instância
        self.nome = nome
        self.idade = idade

    # Método de instância
    def apresentar(self):
        return f"Olá, meu nome é {self.nome} e tenho {self.idade} anos."

    # Método de classe
    @classmethod
    def criar_de_ano_nascimento(cls, nome, ano_nascimento):
        idade = 2025 - ano_nascimento  # Considerando o ano atual como 2025
        return cls(nome, idade)

    # Método estático
    @staticmethod
    def e_adulto(idade):
        return idade >= 18

# Usando atributos e métodos
pessoa1 = Pessoa("Ana", 30)
print(pessoa1.apresentar())  # Método de instância
print(Pessoa.especie)  # Atributo de classe

# Usando método de classe
pessoa2 = Pessoa.criar_de_ano_nascimento("Carlos", 1995)
```

```
print(pessoa2.apresentar())

# Usando método estático
print(Pessoa.e_adulto(16))  # False
print(Pessoa.e_adulto(21))  # True
```

Os Quatro Pilares da POO

1. Encapsulamento

O **encapsulamento** é o princípio de esconder os detalhes internos de uma classe e expor apenas o necessário. Em Python, o encapsulamento é implementado por convenção, usando prefixos de underscore:

- `_atributo`: Indica que o atributo é "protegido" e não deve ser acessado diretamente fora da classe (convenção).
- `__atributo`: Indica que o atributo é "privado" e seu nome é modificado pelo Python para evitar acesso direto (name mangling).

```
class ContaBancaria:
    def __init__(self, titular, saldo_inicial):
        self.titular = titular  # Público
        self._saldo = saldo_inicial  # Protegido
        self.__senha = "1234"  # Privado

    def depositar(self, valor):
        if valor > 0:
            self._saldo += valor
            return True
        return False

    def sacar(self, valor):
        if 0 < valor <= self._saldo:
            self._saldo -= valor
            return True
        return False

    def consultar_saldo(self):
        return self._saldo

    def _validar_senha(self, senha):
        return senha == self.__senha

# Usando a classe
conta = ContaBancaria("João", 1000)
print(conta.titular)  # Acesso a atributo público
print(conta.consultar_saldo())  # Acesso ao saldo através de método

# Acesso direto a atributo protegido (possível, mas não recomendado)
print(conta._saldo)

# Tentativa de acesso a atributo privado (não funciona diretamente)
# print(conta.__senha)  # Isso geraria um AttributeError

# O Python renomeia atributos privados para _NomeClasse__atributo
print(conta._ContaBancaria__senha)  # Isso funciona, mas não é recomendado
```

2. Herança

A **herança** permite que uma classe (subclasse) herde atributos e métodos de outra classe (superclasse). Isso promove a reutilização de código e estabelece uma relação "é um" entre as classes.

```
# Classe base (superclasse)
class Animal:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    def fazer_som(self):
        print("Som genérico de animal")

    def apresentar(self):
        print(f"Olá, sou um animal chamado {self.nome} e tenho {self.idade} anos.")

# Subclasse
class Cachorro(Animal):
    def __init__(self, nome, idade, raca):
        # Chamando o inicializador da superclasse
        super().__init__(nome, idade)
        self.raca = raca

    # Sobrescrevendo um método da superclasse
    def fazer_som(self):
        print("Au au!")

    # Método específico da subclasse
    def abanar_rabo(self):
        print(f"{self.nome} está abanando o rabo!")

# Subclasse
class Gato(Animal):
    def fazer_som(self):
        print("Miau!")

    def ronronar(self):
        print(f"{self.nome} está ronronando...")

# Usando as classes
animal = Animal("Genérico", 5)
animal.apresentar()
animal.fazer_som()

rex = Cachorro("Rex", 3, "Labrador")
rex.apresentar()  # Método herdado
rex.fazer_som()  # Método sobrescrito
rex.abanar_rabo()  # Método específico

felix = Gato("Felix", 2)
felix.apresentar()  # Método herdado
felix.fazer_som()  # Método sobrescrito
felix.ronronar()  # Método específico
```

Python também suporta herança múltipla, onde uma classe pode herdar de várias superclasses:

```
class A:
    def metodo(self):
        print("Método da classe A")

class B:
    def metodo(self):
        print("Método da classe B")

    def outro_metodo(self):
        print("Outro método da classe B")

# Herança múltipla
class C(A, B):
    pass

# Usando a classe
c = C()
c.metodo()  # Chama o método da classe A (primeira na ordem de herança)
c.outro_metodo()  # Chama o método da classe B
```

3. Polimorfismo

O **polimorfismo** permite que objetos de diferentes classes sejam tratados como objetos de uma classe comum. Em Python, o polimorfismo é implementado naturalmente devido à tipagem dinâmica.

```
def fazer_animal_falar(animal):
    animal.fazer_som()

# Usando o polimorfismo
rex = Cachorro("Rex", 3, "Labrador")
felix = Gato("Felix", 2)

fazer_animal_falar(rex)  # Au au!
fazer_animal_falar(felix)  # Miau!
```

4. Abstração

A **abstração** envolve simplificar sistemas complexos ocultando detalhes desnecessários e expondo apenas o essencial. Em Python, podemos usar classes abstratas para definir interfaces que as subclasses devem implementar.

```
from abc import ABC, abstractmethod

# Classe abstrata
class FormaGeometrica(ABC):
    @abstractmethod
    def calcular_area(self):
        pass

    @abstractmethod
    def calcular_perimetro(self):
        pass

# Subclasses concretas
```

```python
class Retangulo(FormaGeometrica):
    def __init__(self, largura, altura):
        self.largura = largura
        self.altura = altura

    def calcular_area(self):
        return self.largura * self.altura

    def calcular_perimetro(self):
        return 2 * (self.largura + self.altura)

class Circulo(FormaGeometrica):
    def __init__(self, raio):
        self.raio = raio

    def calcular_area(self):
        return 3.14159 * self.raio ** 2

    def calcular_perimetro(self):
        return 2 * 3.14159 * self.raio

# Não podemos instanciar uma classe abstrata
# forma = FormaGeometrica()  # Isso geraria um TypeError

# Usando as subclasses
retangulo = Retangulo(5, 3)
print(f"Área do retângulo: {retangulo.calcular_area()}")
print(f"Perímetro do retângulo: {retangulo.calcular_perimetro()}")

circulo = Circulo(4)
print(f"Área do círculo: {circulo.calcular_area()}")
print(f"Perímetro do círculo: {circulo.calcular_perimetro()}")
```

Métodos Especiais (Dunder Methods)

Python possui métodos especiais, também conhecidos como "dunder methods" (double underscore methods), que permitem definir comportamentos específicos para operações integradas da linguagem.

```python
class Ponto:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # Representação para desenvolvedores
    def __repr__(self):
        return f"Ponto({self.x}, {self.y})"

    # Representação para usuários
    def __str__(self):
        return f"({self.x}, {self.y})"

    # Sobrecarga do operador +
    def __add__(self, outro):
        return Ponto(self.x + outro.x, self.y + outro.y)
```

```
    # Sobrecarga do operador ==
    def __eq__(self, outro):
        return self.x == outro.x and self.y == outro.y

    # Método chamado quando o objeto é usado em len()
    def __len__(self):
        return int((self.x ** 2 + self.y ** 2) ** 0.5)

    # Método chamado quando o objeto é usado como uma função
    def __call__(self, z):
        return self.x + self.y + z

# Usando os métodos especiais
p1 = Ponto(3, 4)
p2 = Ponto(1, 2)

print(repr(p1))  # Ponto(3, 4)
print(str(p1))   # (3, 4)
print(p1 + p2)   # (4, 6)
print(p1 == p2)  # False
print(len(p1))   # 5 (distância da origem)
print(p1(5))     # 12 (3 + 4 + 5)
```

Propriedades

As propriedades permitem definir métodos que se comportam como atributos, proporcionando controle sobre o acesso e modificação de atributos.

```
class Temperatura:
    def __init__(self, celsius=0):
        self._celsius = celsius

    # Getter
    @property
    def celsius(self):
        return self._celsius

    # Setter
    @celsius.setter
    def celsius(self, valor):
        if valor < -273.15:
            raise ValueError("Temperatura abaixo do zero absoluto!")
        self._celsius = valor

    # Propriedade calculada
    @property
    def fahrenheit(self):
        return self._celsius * 9/5 + 32

    @fahrenheit.setter
    def fahrenheit(self, valor):
        self.celsius = (valor - 32) * 5/9
```

```python
# Usando propriedades
temp = Temperatura(25)
print(temp.celsius)     # 25
print(temp.fahrenheit)  # 77.0

temp.celsius = 30
print(temp.fahrenheit)  # 86.0

temp.fahrenheit = 68
print(temp.celsius)     # 20.0

# Validação em ação
try:
    temp.celsius = -300
except ValueError as e:
    print(e)  # Temperatura abaixo do zero absoluto!
```

Herança vs. Composição

Embora a herança seja poderosa, às vezes a composição (incluir instâncias de outras classes como atributos) é uma abordagem melhor. Isso segue o princípio "prefira composição sobre herança".

```python
# Abordagem com herança
class Veiculo:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

    def mover(self):
        print("Veículo se movendo")

class Carro(Veiculo):
    def __init__(self, marca, modelo, motor):
        super().__init__(marca, modelo)
        self.motor = motor

    def mover(self):
        print(f"Carro {self.marca} {self.modelo} se movendo com motor {self.motor}")

# Abordagem com composição
class Motor:
    def __init__(self, tipo, potencia):
        self.tipo = tipo
        self.potencia = potencia

    def ligar(self):
        print(f"Motor {self.tipo} ligado")

    def desligar(self):
        print(f"Motor {self.tipo} desligado")

class CarroComposicao:
    def __init__(self, marca, modelo, tipo_motor, potencia_motor):
        self.marca = marca
```

```
        self.modelo = modelo
        self.motor = Motor(tipo_motor, potencia_motor)

    def mover(self):
        self.motor.ligar()
        print(f"Carro {self.marca} {self.modelo} se movendo")

    def parar(self):
        print(f"Carro {self.marca} {self.modelo} parado")
        self.motor.desligar()

# Usando composição
meu_carro = CarroComposicao("Toyota", "Corolla", "V6", 200)
meu_carro.mover()
meu_carro.parar()
```

Classes Internas (Nested Classes)

Python permite definir classes dentro de outras classes, o que pode ser útil para encapsular funcionalidades relacionadas.

```
class Exterior:
    def __init__(self, valor):
        self.valor = valor
        self.interior = self.Interior(valor * 2)

    def mostrar(self):
        print(f"Exterior: {self.valor}")
        self.interior.mostrar()

    class Interior:
        def __init__(self, valor):
            self.valor = valor

        def mostrar(self):
            print(f"Interior: {self.valor}")

# Usando classes aninhadas
obj = Exterior(10)
obj.mostrar()

# Também podemos acessar a classe interna diretamente
obj_interno = Exterior.Interior(5)
obj_interno.mostrar()
```

Metaclasses

Metaclasses são "classes de classes" que definem como as classes são criadas. Elas são um tópico avançado, mas podem ser poderosas para casos específicos.

```
# Definindo uma metaclasse
class Meta(type):
    def __new__(cls, name, bases, attrs):
        # Adiciona um prefixo a todos os métodos não especiais
```

```
        for key, value in list(attrs.items()):
            if not key.startswith('__') and callable(value):
                attrs[f"prefixo_{key}"] = value
                del attrs[key]

        return super().__new__(cls, name, bases, attrs)

# Usando a metaclasse
class MinhaClasse(metaclass=Meta):
    def metodo(self):
        return "Método original"

# A metaclasse transformou 'metodo' em 'prefixo_metodo'
obj = MinhaClasse()
# print(obj.metodo())  # Isso geraria um AttributeError
print(obj.prefixo_metodo())  # "Método original"
```

Boas Práticas em POO com Python

1. **Siga as Convenções de Nomenclatura**:
   - Classes: CamelCase (ex: `MinhaClasse`)
   - Métodos e atributos: snake_case (ex: `meu_metodo`, `meu_atributo`)
   - Constantes: MAIÚSCULAS_COM_UNDERSCORE (ex: `VALOR_MAXIMO`)
2. **Use Encapsulamento Apropriadamente**:
   - Use `_atributo` para indicar atributos protegidos
   - Use `__atributo` para atributos privados quando realmente necessário
   - Forneça métodos getter/setter ou propriedades para acesso controlado
3. **Prefira Composição sobre Herança**:
   - Use herança quando existe uma relação "é um"
   - Use composição quando existe uma relação "tem um"
4. **Mantenha as Classes Coesas**:
   - Cada classe deve ter uma única responsabilidade
   - Evite classes que fazem muitas coisas diferentes
5. **Use Métodos Estáticos e de Classe Apropriadamente**:
   - Métodos estáticos para funções relacionadas à classe que não usam o estado da classe
   - Métodos de classe para métodos que operam na classe como um todo
6. **Documente suas Classes**:
   - Use docstrings para documentar classes, métodos e atributos
   - Explique o propósito, parâmetros e valores de retorno
7. **Teste suas Classes**:
   - Escreva testes unitários para verificar o comportamento das classes
   - Teste casos normais e casos de borda

A programação orientada a objetos é um paradigma poderoso que permite modelar sistemas complexos de forma mais natural e intuitiva. Python oferece suporte completo à POO, com recursos como herança, polimorfismo, encapsulamento e abstração. Dominar esses conceitos permitirá que você escreva código mais organizado, reutilizável e fácil de manter.

No próximo tópico, apresentaremos exercícios práticos para consolidar os conhecimentos adquiridos no módulo de Python Intermediário.

Módulo 2: Python Intermediário

Exercícios Práticos

Nesta seção, apresentamos uma série de exercícios práticos para consolidar os conhecimentos adquiridos no Módulo 2. A prática é fundamental para o aprendizado de programação, especialmente para conceitos intermediários como estruturas de dados, manipulação de arquivos e programação orientada a objetos.

Exercício 1: Estruturas de Dados

Crie um programa que gerencie uma lista de contatos. Cada contato deve ter nome, telefone e email. O programa deve permitir adicionar, remover, buscar e listar contatos.

```
# Solução
def criar_contato(nome, telefone, email):
    """Cria um dicionário representando um contato."""
    return {
        "nome": nome,
        "telefone": telefone,
        "email": email
    }

def adicionar_contato(lista_contatos, contato):
    """Adiciona um contato à lista de contatos."""
    lista_contatos.append(contato)
    print(f"Contato {contato['nome']} adicionado com sucesso!")

def remover_contato(lista_contatos, nome):
    """Remove um contato da lista pelo nome."""
    for i, contato in enumerate(lista_contatos):
        if contato["nome"].lower() == nome.lower():
            del lista_contatos[i]
            print(f"Contato {nome} removido com sucesso!")
            return True
    print(f"Contato {nome} não encontrado.")
    return False

def buscar_contato(lista_contatos, termo):
    """Busca contatos que contenham o termo no nome, telefone ou email."""
    resultados = []
    for contato in lista_contatos:
        if (termo.lower() in contato["nome"].lower() or
            termo in contato["telefone"] or
            termo.lower() in contato["email"].lower()):
```

```
            resultados.append(contato)
    return resultados

def listar_contatos(lista_contatos):
    """Lista todos os contatos."""
    if not lista_contatos:
        print("Lista de contatos vazia.")
        return

    print("\n=== Lista de Contatos ===")
    for i, contato in enumerate(lista_contatos, 1):
        print(f"{i}. Nome: {contato['nome']}")
        print(f"   Telefone: {contato['telefone']}")
        print(f"   Email: {contato['email']}")
        print("-" * 25)

def menu():
    """Exibe o menu de opções."""
    print("\n=== Gerenciador de Contatos ===")
    print("1. Adicionar contato")
    print("2. Remover contato")
    print("3. Buscar contato")
    print("4. Listar todos os contatos")
    print("5. Sair")
    return input("Escolha uma opção: ")

def main():
    """Função principal do programa."""
    contatos = []

    # Pré-populando com alguns contatos para teste
    contatos.append(criar_contato("Ana Silva", "11 98765-4321", "ana@email.com"))
    contatos.append(criar_contato("Bruno Santos", "21 91234-5678", "bruno@email.com"))
    contatos.append(criar_contato("Carla Oliveira", "31 99876-5432", "carla@email.com"))

    while True:
        opcao = menu()

        if opcao == "1":
            nome = input("Nome: ")
            telefone = input("Telefone: ")
            email = input("Email: ")
            novo_contato = criar_contato(nome, telefone, email)
            adicionar_contato(contatos, novo_contato)

        elif opcao == "2":
            nome = input("Nome do contato a remover: ")
            remover_contato(contatos, nome)

        elif opcao == "3":
            termo = input("Termo de busca: ")
            resultados = buscar_contato(contatos, termo)
            if resultados:
```

```
                print(f"\nEncontrados {len(resultados)} contatos:")
                for contato in resultados:
                    print(f"Nome: {contato['nome']}, Telefone: {contato['telefone']}, Email: {contato['
            else:
                print("Nenhum contato encontrado.")

        elif opcao == "4":
            listar_contatos(contatos)

        elif opcao == "5":
            print("Saindo do programa. Até logo!")
            break

        else:
            print("Opção inválida. Tente novamente.")

if __name__ == "__main__":
    main()
```

Exercício 2: Compreensão de Listas

Crie um programa que use compreensão de listas para gerar: 1. Uma lista com os quadrados dos números de 1 a 20 2. Uma lista apenas com os números pares de 1 a 50 3. Uma lista com os números divisíveis por 3 e 5 entre 1 e 100 4. Um dicionário que mapeia cada número de 1 a 10 ao seu cubo

```
# Solução
def main():
    # 1. Lista com os quadrados dos números de 1 a 20
    quadrados = [n**2 for n in range(1, 21)]
    print("Quadrados de 1 a 20:")
    print(quadrados)

    # 2. Lista apenas com os números pares de 1 a 50
    pares = [n for n in range(1, 51) if n % 2 == 0]
    print("\nNúmeros pares de 1 a 50:")
    print(pares)

    # 3. Lista com os números divisíveis por 3 e 5 entre 1 e 100
    divisiveis_3_5 = [n for n in range(1, 101) if n % 3 == 0 and n % 5 == 0]
    print("\nNúmeros divisíveis por 3 e 5 entre 1 e 100:")
    print(divisiveis_3_5)

    # 4. Dicionário que mapeia cada número de 1 a 10 ao seu cubo
    cubos = {n: n**3 for n in range(1, 11)}
    print("\nDicionário de cubos de 1 a 10:")
    for numero, cubo in cubos.items():
        print(f"{numero}³ = {cubo}")

if __name__ == "__main__":
    main()
```

Exercício 3: Manipulação de Arquivos

Crie um programa que leia um arquivo CSV contendo dados de alunos (nome, idade, nota) e realize as seguintes operações: 1. Calcule a média das notas 2. Encontre o aluno com a maior nota 3. Crie um novo arquivo com apenas os alunos aprovados (nota >= 7)

```
# Solução
import csv

def ler_arquivo_csv(nome_arquivo):
    """Lê um arquivo CSV e retorna uma lista de dicionários."""
    alunos = []
    try:
        with open(nome_arquivo, 'r', encoding='utf-8') as arquivo:
            leitor = csv.DictReader(arquivo)
            for linha in leitor:
                # Convertendo idade e nota para números
                linha['idade'] = int(linha['idade'])
                linha['nota'] = float(linha['nota'])
                alunos.append(linha)
        return alunos
    except FileNotFoundError:
        print(f"Erro: O arquivo {nome_arquivo} não foi encontrado.")
        return []
    except Exception as e:
        print(f"Erro ao ler o arquivo: {e}")
        return []

def calcular_media_notas(alunos):
    """Calcula a média das notas dos alunos."""
    if not alunos:
        return 0
    soma_notas = sum(aluno['nota'] for aluno in alunos)
    return soma_notas / len(alunos)

def encontrar_aluno_maior_nota(alunos):
    """Encontra o aluno com a maior nota."""
    if not alunos:
        return None
    return max(alunos, key=lambda aluno: aluno['nota'])

def criar_arquivo_aprovados(alunos, nome_arquivo, nota_minima=7.0):
    """Cria um novo arquivo com apenas os alunos aprovados."""
    aprovados = [aluno for aluno in alunos if aluno['nota'] >= nota_minima]

    try:
        with open(nome_arquivo, 'w', newline='', encoding='utf-8') as arquivo:
            campos = ['nome', 'idade', 'nota']
            escritor = csv.DictWriter(arquivo, fieldnames=campos)
            escritor.writeheader()
            escritor.writerows(aprovados)
        return len(aprovados)
    except Exception as e:
        print(f"Erro ao criar o arquivo: {e}")
        return 0
```

```python
def main():
    # Criando um arquivo CSV de exemplo
    with open('alunos.csv', 'w', newline='', encoding='utf-8') as arquivo:
        escritor = csv.writer(arquivo)
        escritor.writerow(['nome', 'idade', 'nota'])
        escritor.writerow(['Ana Silva', 18, 9.5])
        escritor.writerow(['Bruno Santos', 19, 7.8])
        escritor.writerow(['Carla Oliveira', 20, 6.5])
        escritor.writerow(['Daniel Lima', 18, 8.2])
        escritor.writerow(['Elena Martins', 19, 5.4])

    # Lendo o arquivo
    alunos = ler_arquivo_csv('alunos.csv')

    if alunos:
        # Calculando a média das notas
        media = calcular_media_notas(alunos)
        print(f"Média das notas: {media:.2f}")

        # Encontrando o aluno com a maior nota
        melhor_aluno = encontrar_aluno_maior_nota(alunos)
        print(f"Aluno com a maior nota: {melhor_aluno['nome']} ({melhor_aluno['nota']})")

        # Criando um novo arquivo com apenas os alunos aprovados
        qtd_aprovados = criar_arquivo_aprovados(alunos, 'aprovados.csv')
        print(f"Arquivo 'aprovados.csv' criado com {qtd_aprovados} alunos aprovados.")

        # Mostrando o conteúdo do novo arquivo
        print("\nConteúdo do arquivo 'aprovados.csv':")
        with open('aprovados.csv', 'r', encoding='utf-8') as arquivo:
            print(arquivo.read())

if __name__ == "__main__":
    main()
```

Exercício 4: Módulos e Pacotes

Crie um pacote chamado `utilidades` com os seguintes módulos: 1. `matematica.py`: Funções para operações matemáticas básicas 2. `texto.py`: Funções para manipulação de texto 3. `arquivo.py`: Funções para manipulação de arquivos

Em seguida, crie um programa principal que importe e use funções de cada um desses módulos.

```python
# Estrutura de diretórios:
# utilidades/
#   __init__.py
#   matematica.py
#   texto.py
#   arquivo.py
# programa_principal.py

# utilidades/__init__.py
"""Pacote de utilidades diversas."""
print("Pacote 'utilidades' importado.")
```

```python
# utilidades/matematica.py
"""Módulo com funções matemáticas básicas."""

def soma(a, b):
    """Retorna a soma de dois números."""
    return a + b

def subtracao(a, b):
    """Retorna a diferença entre dois números."""
    return a - b

def multiplicacao(a, b):
    """Retorna o produto de dois números."""
    return a * b

def divisao(a, b):
    """Retorna o quociente de dois números."""
    if b == 0:
        raise ValueError("Divisão por zero não é permitida.")
    return a / b

def potencia(base, expoente):
    """Retorna a base elevada ao expoente."""
    return base ** expoente

# utilidades/texto.py
"""Módulo com funções para manipulação de texto."""

def contar_palavras(texto):
    """Conta o número de palavras em um texto."""
    palavras = texto.split()
    return len(palavras)

def inverter(texto):
    """Inverte um texto."""
    return texto[::-1]

def capitalizar_palavras(texto):
    """Capitaliza a primeira letra de cada palavra."""
    return ' '.join(palavra.capitalize() for palavra in texto.split())

def remover_espacos_extras(texto):
    """Remove espaços extras de um texto."""
    import re
    return re.sub(r'\s+', ' ', texto).strip()

# utilidades/arquivo.py
"""Módulo com funções para manipulação de arquivos."""

def ler_arquivo(nome_arquivo):
    """Lê o conteúdo de um arquivo."""
    try:
        with open(nome_arquivo, 'r', encoding='utf-8') as arquivo:
```

```
            return arquivo.read()
    except FileNotFoundError:
        print(f"Erro: O arquivo {nome_arquivo} não foi encontrado.")
        return ""
    except Exception as e:
        print(f"Erro ao ler o arquivo: {e}")
        return ""

def escrever_arquivo(nome_arquivo, conteudo):
    """Escreve conteúdo em um arquivo."""
    try:
        with open(nome_arquivo, 'w', encoding='utf-8') as arquivo:
            arquivo.write(conteudo)
        return True
    except Exception as e:
        print(f"Erro ao escrever no arquivo: {e}")
        return False

def adicionar_ao_arquivo(nome_arquivo, conteudo):
    """Adiciona conteúdo ao final de um arquivo."""
    try:
        with open(nome_arquivo, 'a', encoding='utf-8') as arquivo:
            arquivo.write(conteudo)
        return True
    except Exception as e:
        print(f"Erro ao adicionar ao arquivo: {e}")
        return False

# programa_principal.py
"""Programa principal que usa o pacote utilidades."""

from utilidades import matematica, texto, arquivo

def main():
    # Usando funções do módulo matematica
    print("=== Funções Matemáticas ===")
    a, b = 10, 3
    print(f"{a} + {b} = {matematica.soma(a, b)}")
    print(f"{a} - {b} = {matematica.subtracao(a, b)}")
    print(f"{a} * {b} = {matematica.multiplicacao(a, b)}")
    print(f"{a} / {b} = {matematica.divisao(a, b):.2f}")
    print(f"{a} ^ {b} = {matematica.potencia(a, b)}")

    # Usando funções do módulo texto
    print("\n=== Funções de Texto ===")
    frase = "   python é uma linguagem de programação incrível   "
    print(f"Texto original: '{frase}'")
    print(f"Número de palavras: {texto.contar_palavras(frase)}")
    print(f"Texto invertido: '{texto.inverter(frase)}'")
    print(f"Texto capitalizado: '{texto.capitalizar_palavras(frase)}'")
    print(f"Sem espaços extras: '{texto.remover_espacos_extras(frase)}'")

    # Usando funções do módulo arquivo
```

```
    print("\n=== Funções de Arquivo ===")
    nome_arquivo = "exemplo.txt"

    # Escrevendo no arquivo
    conteudo = "Este é um arquivo de exemplo.\nCriado para demonstrar o módulo de manipulação de arquiv
    if arquivo.escrever_arquivo(nome_arquivo, conteudo):
        print(f"Conteúdo escrito em '{nome_arquivo}'")

    # Adicionando ao arquivo
    if arquivo.adicionar_ao_arquivo(nome_arquivo, "\nEsta linha foi adicionada posteriormente."):
        print(f"Conteúdo adicionado a '{nome_arquivo}'")

    # Lendo o arquivo
    conteudo_lido = arquivo.ler_arquivo(nome_arquivo)
    if conteudo_lido:
        print(f"\nConteúdo do arquivo '{nome_arquivo}':")
        print(conteudo_lido)

if __name__ == "__main__":
    main()
```

Exercício 5: Tratamento de Exceções

Crie um programa que simule um sistema bancário simples com as seguintes funcionalidades: 1. Criar conta 2. Depositar 3. Sacar 4. Consultar saldo

O programa deve tratar exceções como saldo insuficiente, valor inválido, etc.

```
# Solução
class SaldoInsuficienteError(Exception):
    """Exceção lançada quando o saldo é insuficiente para uma operação."""
    pass

class ValorInvalidoError(Exception):
    """Exceção lançada quando um valor inválido é fornecido."""
    pass

class ContaNaoEncontradaError(Exception):
    """Exceção lançada quando uma conta não é encontrada."""
    pass

class ContaBancaria:
    def __init__(self, numero, titular, saldo_inicial=0):
        self.numero = numero
        self.titular = titular
        self._saldo = 0  # Inicializa com zero

        # Usa o método depositar para validar o saldo inicial
        if saldo_inicial > 0:
            self.depositar(saldo_inicial)

    def depositar(self, valor):
        """Deposita um valor na conta."""
        if valor <= 0:
```

```
            raise ValorInvalidoError("O valor do depósito deve ser positivo.")
        self._saldo += valor
        return self._saldo

    def sacar(self, valor):
        """Saca um valor da conta."""
        if valor <= 0:
            raise ValorInvalidoError("O valor do saque deve ser positivo.")
        if valor > self._saldo:
            raise SaldoInsuficienteError(f"Saldo insuficiente. Saldo atual: R${self._saldo:.2f}")
        self._saldo -= valor
        return self._saldo

    def consultar_saldo(self):
        """Retorna o saldo atual da conta."""
        return self._saldo

    def __str__(self):
        return f"Conta {self.numero} - Titular: {self.titular} - Saldo: R${self._saldo:.2f}"

class SistemaBancario:
    def __init__(self):
        self.contas = {}

    def criar_conta(self, numero, titular, saldo_inicial=0):
        """Cria uma nova conta bancária."""
        if numero in self.contas:
            raise ValueError(f"Já existe uma conta com o número {numero}.")

        try:
            conta = ContaBancaria(numero, titular, saldo_inicial)
            self.contas[numero] = conta
            return conta
        except ValorInvalidoError as e:
            raise ValorInvalidoError(f"Erro ao criar conta: {e}")

    def buscar_conta(self, numero):
        """Busca uma conta pelo número."""
        if numero not in self.contas:
            raise ContaNaoEncontradaError(f"Conta {numero} não encontrada.")
        return self.contas[numero]

    def depositar(self, numero_conta, valor):
        """Deposita um valor em uma conta."""
        conta = self.buscar_conta(numero_conta)
        return conta.depositar(valor)

    def sacar(self, numero_conta, valor):
        """Saca um valor de uma conta."""
        conta = self.buscar_conta(numero_conta)
        return conta.sacar(valor)

    def consultar_saldo(self, numero_conta):
```

```
        """Consulta o saldo de uma conta."""
        conta = self.buscar_conta(numero_conta)
        return conta.consultar_saldo()

    def listar_contas(self):
        """Lista todas as contas cadastradas."""
        if not self.contas:
            print("Não há contas cadastradas.")
            return

        print("\n=== Contas Cadastradas ===")
        for conta in self.contas.values():
            print(conta)

def menu():
    """Exibe o menu de opções."""
    print("\n=== Sistema Bancário ===")
    print("1. Criar conta")
    print("2. Depositar")
    print("3. Sacar")
    print("4. Consultar saldo")
    print("5. Listar contas")
    print("6. Sair")
    return input("Escolha uma opção: ")

def main():
    """Função principal do programa."""
    sistema = SistemaBancario()

    # Pré-populando com algumas contas para teste
    try:
        sistema.criar_conta("001", "Ana Silva", 1000)
        sistema.criar_conta("002", "Bruno Santos", 500)
        print("Contas de teste criadas com sucesso!")
    except Exception as e:
        print(f"Erro ao criar contas de teste: {e}")

    while True:
        try:
            opcao = menu()

            if opcao == "1":
                numero = input("Número da conta: ")
                titular = input("Nome do titular: ")

                try:
                    saldo_inicial = float(input("Saldo inicial (opcional, pressione Enter para 0): ") o
                    conta = sistema.criar_conta(numero, titular, saldo_inicial)
                    print(f"Conta criada com sucesso: {conta}")
                except ValueError:
                    print("Erro: O saldo inicial deve ser um número.")

            elif opcao == "2":
```

```
                numero = input("Número da conta: ")
                try:
                    valor = float(input("Valor do depósito: "))
                    novo_saldo = sistema.depositar(numero, valor)
                    print(f"Depósito realizado com sucesso. Novo saldo: R${novo_saldo:.2f}")
                except ValueError:
                    print("Erro: O valor deve ser um número.")

            elif opcao == "3":
                numero = input("Número da conta: ")
                try:
                    valor = float(input("Valor do saque: "))
                    novo_saldo = sistema.sacar(numero, valor)
                    print(f"Saque realizado com sucesso. Novo saldo: R${novo_saldo:.2f}")
                except ValueError:
                    print("Erro: O valor deve ser um número.")

            elif opcao == "4":
                numero = input("Número da conta: ")
                saldo = sistema.consultar_saldo(numero)
                print(f"Saldo atual: R${saldo:.2f}")

            elif opcao == "5":
                sistema.listar_contas()

            elif opcao == "6":
                print("Saindo do sistema bancário. Até logo!")
                break

            else:
                print("Opção inválida. Tente novamente.")

        except ContaNaoEncontradaError as e:
            print(f"Erro: {e}")
        except SaldoInsuficienteError as e:
            print(f"Erro: {e}")
        except ValorInvalidoError as e:
            print(f"Erro: {e}")
        except Exception as e:
            print(f"Erro inesperado: {e}")

if __name__ == "__main__":
    main()
```

Exercício 6: Programação Orientada a Objetos

Crie um sistema de gerenciamento de biblioteca usando POO. O sistema deve ter classes para Livro, Autor, Usuário e Biblioteca, com métodos para emprestar e devolver livros, buscar livros por título ou autor, etc.

```
# Solução
from datetime import datetime, timedelta

class Autor:
    def __init__(self, nome, nacionalidade):
```

```
        self.nome = nome
        self.nacionalidade = nacionalidade
        self.livros = []

    def adicionar_livro(self, livro):
        """Adiciona um livro à lista de livros do autor."""
        self.livros.append(livro)

    def __str__(self):
        return f"{self.nome} ({self.nacionalidade})"

class Livro:
    def __init__(self, titulo, autor, ano_publicacao, codigo):
        self.titulo = titulo
        self.autor = autor
        self.ano_publicacao = ano_publicacao
        self.codigo = codigo
        self.disponivel = True
        self.usuario_atual = None
        self.data_devolucao = None

        # Adiciona este livro à lista de livros do autor
        autor.adicionar_livro(self)

    def emprestar(self, usuario, dias=14):
        """Empresta o livro para um usuário."""
        if not self.disponivel:
            raise ValueError(f"O livro '{self.titulo}' não está disponível.")

        self.disponivel = False
        self.usuario_atual = usuario
        self.data_devolucao = datetime.now() + timedelta(days=dias)
        usuario.adicionar_livro_emprestado(self)
        return self.data_devolucao

    def devolver(self):
        """Devolve o livro à biblioteca."""
        if self.disponivel:
            raise ValueError(f"O livro '{self.titulo}' já está disponível.")

        usuario = self.usuario_atual
        self.disponivel = True
        self.usuario_atual = None
        self.data_devolucao = None
        usuario.remover_livro_emprestado(self)
        return True

    def __str__(self):
        status = "Disponível" if self.disponivel else f"Emprestado para {self.usuario_atual.nome} até {
        return f"{self.titulo} ({self.ano_publicacao}) - {self.autor.nome} - {status}"

class Usuario:
    def __init__(self, nome, email, codigo):
```

```
        self.nome = nome
        self.email = email
        self.codigo = codigo
        self.livros_emprestados = []

    def adicionar_livro_emprestado(self, livro):
        """Adiciona um livro à lista de livros emprestados do usuário."""
        self.livros_emprestados.append(livro)

    def remover_livro_emprestado(self, livro):
        """Remove um livro da lista de livros emprestados do usuário."""
        if livro in self.livros_emprestados:
            self.livros_emprestados.remove(livro)

    def listar_livros_emprestados(self):
        """Lista todos os livros emprestados pelo usuário."""
        if not self.livros_emprestados:
            return "Nenhum livro emprestado."

        resultado = f"Livros emprestados por {self.nome}:\n"
        for i, livro in enumerate(self.livros_emprestados, 1):
            resultado += f"{i}. {livro.titulo} - Devolução: {livro.data_devolucao.strftime('%d/%m/%Y')}"
        return resultado

    def __str__(self):
        return f"{self.nome} ({self.email}) - {len(self.livros_emprestados)} livros emprestados"

class Biblioteca:
    def __init__(self, nome):
        self.nome = nome
        self.livros = {}
        self.autores = {}
        self.usuarios = {}

    def adicionar_autor(self, autor):
        """Adiciona um autor à biblioteca."""
        self.autores[autor.nome] = autor
        return autor

    def adicionar_livro(self, livro):
        """Adiciona um livro à biblioteca."""
        self.livros[livro.codigo] = livro
        return livro

    def adicionar_usuario(self, usuario):
        """Adiciona um usuário à biblioteca."""
        self.usuarios[usuario.codigo] = usuario
        return usuario

    def buscar_livro_por_codigo(self, codigo):
        """Busca um livro pelo código."""
        return self.livros.get(codigo)
```

```
    def buscar_livros_por_titulo(self, titulo):
        """Busca livros pelo título (parcial)."""
        return [livro for livro in self.livros.values() if titulo.lower() in livro.titulo.lower()]

    def buscar_livros_por_autor(self, nome_autor):
        """Busca livros pelo nome do autor (parcial)."""
        return [livro for livro in self.livros.values() if nome_autor.lower() in livro.autor.nome.lower

    def buscar_usuario_por_codigo(self, codigo):
        """Busca um usuário pelo código."""
        return self.usuarios.get(codigo)

    def emprestar_livro(self, codigo_livro, codigo_usuario, dias=14):
        """Empresta um livro para um usuário."""
        livro = self.buscar_livro_por_codigo(codigo_livro)
        if not livro:
            raise ValueError(f"Livro com código {codigo_livro} não encontrado.")

        usuario = self.buscar_usuario_por_codigo(codigo_usuario)
        if not usuario:
            raise ValueError(f"Usuário com código {codigo_usuario} não encontrado.")

        return livro.emprestar(usuario, dias)

    def devolver_livro(self, codigo_livro):
        """Devolve um livro à biblioteca."""
        livro = self.buscar_livro_por_codigo(codigo_livro)
        if not livro:
            raise ValueError(f"Livro com código {codigo_livro} não encontrado.")

        return livro.devolver()

    def listar_livros_disponiveis(self):
        """Lista todos os livros disponíveis."""
        livros_disponiveis = [livro for livro in self.livros.values() if livro.disponivel]
        return livros_disponiveis

    def listar_livros_emprestados(self):
        """Lista todos os livros emprestados."""
        livros_emprestados = [livro for livro in self.livros.values() if not livro.disponivel]
        return livros_emprestados

    def __str__(self):
        return f"Biblioteca {self.nome} - {len(self.livros)} livros, {len(self.autores)} autores, {len(s

def main():
    # Criando a biblioteca
    biblioteca = Biblioteca("Biblioteca Municipal")

    # Criando autores
    autor1 = Autor("J.K. Rowling", "Britânica")
    autor2 = Autor("George Orwell", "Britânico")
    autor3 = Autor("Machado de Assis", "Brasileiro")
```

```
biblioteca.adicionar_autor(autor1)
biblioteca.adicionar_autor(autor2)
biblioteca.adicionar_autor(autor3)

# Criando livros
livro1 = Livro("Harry Potter e a Pedra Filosofal", autor1, 1997, "HP001")
livro2 = Livro("Harry Potter e a Câmara Secreta", autor1, 1998, "HP002")
livro3 = Livro("1984", autor2, 1949, "GO001")
livro4 = Livro("A Revolução dos Bichos", autor2, 1945, "GO002")
livro5 = Livro("Dom Casmurro", autor3, 1899, "MA001")
livro6 = Livro("Memórias Póstumas de Brás Cubas", autor3, 1881, "MA002")

biblioteca.adicionar_livro(livro1)
biblioteca.adicionar_livro(livro2)
biblioteca.adicionar_livro(livro3)
biblioteca.adicionar_livro(livro4)
biblioteca.adicionar_livro(livro5)
biblioteca.adicionar_livro(livro6)

# Criando usuários
usuario1 = Usuario("Ana Silva", "ana@email.com", "U001")
usuario2 = Usuario("Bruno Santos", "bruno@email.com", "U002")

biblioteca.adicionar_usuario(usuario1)
biblioteca.adicionar_usuario(usuario2)

# Testando funcionalidades
print(biblioteca)

print("\n=== Livros disponíveis ===")
for livro in biblioteca.listar_livros_disponiveis():
    print(livro)

# Emprestando livros
try:
    data_devolucao = biblioteca.emprestar_livro("HP001", "U001")
    print(f"\nLivro 'Harry Potter e a Pedra Filosofal' emprestado para Ana Silva. Data de devolução

    data_devolucao = biblioteca.emprestar_livro("GO001", "U001")
    print(f"Livro '1984' emprestado para Ana Silva. Data de devolução: {data_devolucao.strftime('%d

    data_devolucao = biblioteca.emprestar_livro("MA001", "U002")
    print(f"Livro 'Dom Casmurro' emprestado para Bruno Santos. Data de devolução: {data_devolucao.s
except ValueError as e:
    print(f"Erro: {e}")

# Listando livros emprestados por usuário
print("\n" + usuario1.listar_livros_emprestados())
print(usuario2.listar_livros_emprestados())

# Devolvendo um livro
try:
    biblioteca.devolver_livro("HP001")
```

```
        print("\nLivro 'Harry Potter e a Pedra Filosofal' devolvido com sucesso.")
    except ValueError as e:
        print(f"Erro: {e}")

    # Buscando livros
    print("\n=== Busca por título 'Harry' ===")
    for livro in biblioteca.buscar_livros_por_titulo("Harry"):
        print(livro)

    print("\n=== Busca por autor 'Machado' ===")
    for livro in biblioteca.buscar_livros_por_autor("Machado"):
        print(livro)

    # Listando livros disponíveis e emprestados após as operações
    print("\n=== Livros disponíveis ===")
    for livro in biblioteca.listar_livros_disponiveis():
        print(livro)

    print("\n=== Livros emprestados ===")
    for livro in biblioteca.listar_livros_emprestados():
        print(livro)

if __name__ == "__main__":
    main()
```

Exercício 7: Análise de Dados com Pandas

Crie um programa que leia um arquivo CSV contendo dados de vendas (produto, quantidade, preço) e realize uma análise básica, como total de vendas por produto, produto mais vendido, etc.

```
# Solução
import pandas as pd
import matplotlib.pyplot as plt
from datetime import datetime

def criar_arquivo_exemplo():
    """Cria um arquivo CSV de exemplo com dados de vendas."""
    dados = [
        ["2023-01-15", "Notebook", 2, 3500.00],
        ["2023-01-16", "Smartphone", 5, 1200.00],
        ["2023-01-17", "Tablet", 3, 800.00],
        ["2023-01-18", "Notebook", 1, 3500.00],
        ["2023-01-19", "Smartphone", 4, 1200.00],
        ["2023-01-20", "Monitor", 3, 950.00],
        ["2023-01-21", "Tablet", 2, 800.00],
        ["2023-01-22", "Teclado", 10, 120.00],
        ["2023-01-23", "Mouse", 15, 80.00],
        ["2023-01-24", "Notebook", 3, 3500.00],
        ["2023-01-25", "Smartphone", 6, 1200.00],
        ["2023-01-26", "Monitor", 4, 950.00],
        ["2023-01-27", "Teclado", 8, 120.00],
        ["2023-01-28", "Mouse", 12, 80.00],
        ["2023-01-29", "Tablet", 5, 800.00],
        ["2023-01-30", "Notebook", 2, 3500.00]
```

```
    ]

    df = pd.DataFrame(dados, columns=["Data", "Produto", "Quantidade", "Preco"])
    df.to_csv("vendas.csv", index=False)
    print("Arquivo 'vendas.csv' criado com sucesso!")
    return df

def analisar_vendas(df):
    """Realiza uma análise básica dos dados de vendas."""
    # Calculando o valor total de cada venda
    df["Valor_Total"] = df["Quantidade"] * df["Preco"]

    # Convertendo a coluna de data para o tipo datetime
    df["Data"] = pd.to_datetime(df["Data"])

    # Análise 1: Total de vendas por produto
    vendas_por_produto = df.groupby("Produto").agg({
        "Quantidade": "sum",
        "Valor_Total": "sum"
    }).sort_values("Valor_Total", ascending=False)

    # Análise 2: Produto mais vendido (em quantidade)
    produto_mais_vendido = vendas_por_produto["Quantidade"].idxmax()
    qtd_mais_vendido = vendas_por_produto.loc[produto_mais_vendido, "Quantidade"]

    # Análise 3: Produto com maior faturamento
    produto_maior_faturamento = vendas_por_produto["Valor_Total"].idxmax()
    valor_maior_faturamento = vendas_por_produto.loc[produto_maior_faturamento, "Valor_Total"]

    # Análise 4: Vendas por dia
    vendas_por_dia = df.groupby("Data").agg({
        "Quantidade": "sum",
        "Valor_Total": "sum"
    })

    # Análise 5: Dia com maior valor de vendas
    dia_maior_venda = vendas_por_dia["Valor_Total"].idxmax()
    valor_maior_dia = vendas_por_dia.loc[dia_maior_venda, "Valor_Total"]

    # Exibindo os resultados
    print("\n=== Análise de Vendas ===")
    print("\nTotal de vendas por produto:")
    print(vendas_por_produto)

    print(f"\nProduto mais vendido: {produto_mais_vendido} ({qtd_mais_vendido} unidades)")
    print(f"Produto com maior faturamento: {produto_maior_faturamento} (R${valor_maior_faturamento:.2f}
    print(f"Dia com maior valor de vendas: {dia_maior_venda.strftime('%d/%m/%Y')} (R${valor_maior_dia:.

    print(f"\nTotal geral de vendas: {df['Quantidade'].sum()} unidades")
    print(f"Faturamento total: R${df['Valor_Total'].sum():.2f}")

    # Criando gráficos
    plt.figure(figsize=(12, 6))
```

```
    # Gráfico 1: Quantidade vendida por produto
    plt.subplot(1, 2, 1)
    vendas_por_produto["Quantidade"].plot(kind="bar", color="skyblue")
    plt.title("Quantidade Vendida por Produto")
    plt.xlabel("Produto")
    plt.ylabel("Quantidade")
    plt.xticks(rotation=45)

    # Gráfico 2: Faturamento por produto
    plt.subplot(1, 2, 2)
    vendas_por_produto["Valor_Total"].plot(kind="bar", color="salmon")
    plt.title("Faturamento por Produto")
    plt.xlabel("Produto")
    plt.ylabel("Valor Total (R$)")
    plt.xticks(rotation=45)

    plt.tight_layout()
    plt.savefig("analise_vendas.png")
    print("\nGráficos salvos em 'analise_vendas.png'")

    return vendas_por_produto

def main():
    try:
        # Tentando ler o arquivo existente
        df = pd.read_csv("vendas.csv")
        print("Arquivo 'vendas.csv' carregado com sucesso!")
    except FileNotFoundError:
        # Se o arquivo não existir, cria um arquivo de exemplo
        df = criar_arquivo_exemplo()

    # Realizando a análise
    vendas_por_produto = analisar_vendas(df)

    # Exportando os resultados para um arquivo Excel
    vendas_por_produto.to_excel("resumo_vendas.xlsx")
    print("Resumo de vendas exportado para 'resumo_vendas.xlsx'")

if __name__ == "__main__":
    main()
```

Exercício 8: Web Scraping

Crie um programa que extraia informações de uma página web, como títulos de notícias de um portal de notícias ou preços de produtos de um e-commerce.

```
# Solução
import requests
from bs4 import BeautifulSoup
import pandas as pd
from datetime import datetime

def extrair_noticias_g1():
```

```
    """Extrai os títulos e links das notícias mais recentes do G1."""
    url = "https://g1.globo.com/"

    try:
        # Fazendo a requisição HTTP
        headers = {
            "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Ge
        }
        response = requests.get(url, headers=headers)
        response.raise_for_status()  # Levanta uma exceção para códigos de erro HTTP

        # Parseando o HTML
        soup = BeautifulSoup(response.text, "html.parser")

        # Encontrando os elementos das notícias
        # Nota: Os seletores podem mudar se o site for atualizado
        noticias = []

        # Buscando os elementos de notícia
        elementos_noticias = soup.select(".feed-post-body")

        for elemento in elementos_noticias[:10]:  # Limitando a 10 notícias
            titulo_elemento = elemento.select_one(".feed-post-body-title")
            link_elemento = elemento.select_one("a")

            if titulo_elemento and link_elemento:
                titulo = titulo_elemento.text.strip()
                link = link_elemento["href"]
                noticias.append({"titulo": titulo, "link": link})

        return noticias

    except requests.exceptions.RequestException as e:
        print(f"Erro ao fazer a requisição: {e}")
        return []
    except Exception as e:
        print(f"Erro inesperado: {e}")
        return []

def extrair_produtos_mercadolivre(produto, limite=10):
    """Extrai informações de produtos do Mercado Livre."""
    url = f"https://lista.mercadolivre.com.br/{produto.replace(' ', '-')}"

    try:
        # Fazendo a requisição HTTP
        headers = {
            "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Ge
        }
        response = requests.get(url, headers=headers)
        response.raise_for_status()

        # Parseando o HTML
        soup = BeautifulSoup(response.text, "html.parser")
```

```
        # Encontrando os elementos dos produtos
        produtos = []

        # Buscando os elementos de produto
        elementos_produtos = soup.select(".ui-search-result")

        for elemento in elementos_produtos[:limite]:
            try:
                titulo_elemento = elemento.select_one(".ui-search-item__title")
                preco_elemento = elemento.select_one(".price-tag-amount")
                link_elemento = elemento.select_one(".ui-search-link")

                if titulo_elemento and preco_elemento and link_elemento:
                    titulo = titulo_elemento.text.strip()
                    preco_texto = preco_elemento.text.strip()
                    link = link_elemento["href"]

                    # Extraindo apenas o valor numérico do preço
                    preco = preco_texto.replace("R$", "").replace(".", "").replace(",", ".").strip()
                    try:
                        preco = float(preco)
                    except ValueError:
                        preco = None

                    produtos.append({
                        "titulo": titulo,
                        "preco": preco,
                        "link": link
                    })
            except Exception as e:
                print(f"Erro ao processar um produto: {e}")
                continue

        return produtos

    except requests.exceptions.RequestException as e:
        print(f"Erro ao fazer a requisição: {e}")
        return []
    except Exception as e:
        print(f"Erro inesperado: {e}")
        return []

def salvar_resultados(dados, nome_arquivo):
    """Salva os resultados em um arquivo CSV."""
    df = pd.DataFrame(dados)
    df.to_csv(nome_arquivo, index=False, encoding="utf-8-sig")
    print(f"Resultados salvos em '{nome_arquivo}'")

def main():
    print("=== Web Scraping ===")
    print("1. Extrair notícias do G1")
    print("2. Extrair produtos do Mercado Livre")
    opcao = input("Escolha uma opção: ")
```

```
    if opcao == "1":
        print("\nExtraindo notícias do G1...")
        noticias = extrair_noticias_g1()

        if noticias:
            print(f"\nForam encontradas {len(noticias)} notícias:")
            for i, noticia in enumerate(noticias, 1):
                print(f"{i}. {noticia['titulo']}")
                print(f"   Link: {noticia['link']}")
                print()

            # Salvando os resultados
            data_hora = datetime.now().strftime("%Y%m%d_%H%M%S")
            nome_arquivo = f"noticias_g1_{data_hora}.csv"
            salvar_resultados(noticias, nome_arquivo)
        else:
            print("Nenhuma notícia encontrada.")

    elif opcao == "2":
        produto = input("\nDigite o nome do produto a pesquisar: ")
        limite = int(input("Quantidade de produtos a extrair: ") or "10")

        print(f"\nExtraindo produtos '{produto}' do Mercado Livre...")
        produtos = extrair_produtos_mercadolivre(produto, limite)

        if produtos:
            print(f"\nForam encontrados {len(produtos)} produtos:")
            for i, produto in enumerate(produtos, 1):
                preco = f"R$ {produto['preco']:.2f}" if produto['preco'] else "Preço não disponível"
                print(f"{i}. {produto['titulo']}")
                print(f"   Preço: {preco}")
                print(f"   Link: {produto['link']}")
                print()

            # Salvando os resultados
            data_hora = datetime.now().strftime("%Y%m%d_%H%M%S")
            nome_arquivo = f"produtos_ml_{data_hora}.csv"
            salvar_resultados(produtos, nome_arquivo)
        else:
            print("Nenhum produto encontrado.")

    else:
        print("Opção inválida.")

if __name__ == "__main__":
    main()
```

Exercício 9: API REST com Flask

Crie uma API REST simples usando Flask para gerenciar uma lista de tarefas (to-do list).

```
# Solução
from flask import Flask, request, jsonify
```

```
import uuid
from datetime import datetime

app = Flask(__name__)

# Banco de dados em memória
tarefas = []

# Rotas da API
@app.route('/tarefas', methods=['GET'])
def listar_tarefas():
    """Retorna todas as tarefas."""
    return jsonify(tarefas)

@app.route('/tarefas/<string:tarefa_id>', methods=['GET'])
def obter_tarefa(tarefa_id):
    """Retorna uma tarefa específica pelo ID."""
    tarefa = next((t for t in tarefas if t['id'] == tarefa_id), None)
    if tarefa:
        return jsonify(tarefa)
    return jsonify({"erro": "Tarefa não encontrada"}), 404

@app.route('/tarefas', methods=['POST'])
def criar_tarefa():
    """Cria uma nova tarefa."""
    dados = request.json

    if not dados or 'titulo' not in dados:
        return jsonify({"erro": "Dados inválidos. O título é obrigatório."}), 400

    nova_tarefa = {
        'id': str(uuid.uuid4()),
        'titulo': dados['titulo'],
        'descricao': dados.get('descricao', ''),
        'concluida': False,
        'criada_em': datetime.now().isoformat(),
        'atualizada_em': datetime.now().isoformat()
    }

    tarefas.append(nova_tarefa)
    return jsonify(nova_tarefa), 201

@app.route('/tarefas/<string:tarefa_id>', methods=['PUT'])
def atualizar_tarefa(tarefa_id):
    """Atualiza uma tarefa existente."""
    dados = request.json

    if not dados:
        return jsonify({"erro": "Dados inválidos."}), 400

    tarefa = next((t for t in tarefas if t['id'] == tarefa_id), None)
    if not tarefa:
        return jsonify({"erro": "Tarefa não encontrada"}), 404
```

```
    # Atualiza os campos
    if 'titulo' in dados:
        tarefa['titulo'] = dados['titulo']
    if 'descricao' in dados:
        tarefa['descricao'] = dados['descricao']
    if 'concluida' in dados:
        tarefa['concluida'] = dados['concluida']

    tarefa['atualizada_em'] = datetime.now().isoformat()

    return jsonify(tarefa)

@app.route('/tarefas/<string:tarefa_id>', methods=['DELETE'])
def excluir_tarefa(tarefa_id):
    """Exclui uma tarefa."""
    global tarefas
    tarefa = next((t for t in tarefas if t['id'] == tarefa_id), None)

    if not tarefa:
        return jsonify({"erro": "Tarefa não encontrada"}), 404

    tarefas = [t for t in tarefas if t['id'] != tarefa_id]
    return jsonify({"mensagem": "Tarefa excluída com sucesso"})

@app.route('/tarefas/concluidas', methods=['GET'])
def listar_tarefas_concluidas():
    """Retorna todas as tarefas concluídas."""
    concluidas = [t for t in tarefas if t['concluida']]
    return jsonify(concluidas)

@app.route('/tarefas/pendentes', methods=['GET'])
def listar_tarefas_pendentes():
    """Retorna todas as tarefas pendentes."""
    pendentes = [t for t in tarefas if not t['concluida']]
    return jsonify(pendentes)

# Rota para a página inicial
@app.route('/')
def pagina_inicial():
    """Retorna informações sobre a API."""
    return jsonify({
        "api": "API de Tarefas",
        "versao": "1.0",
        "rotas": [
            {"metodo": "GET", "url": "/tarefas", "descricao": "Lista todas as tarefas"},
            {"metodo": "GET", "url": "/tarefas/<id>", "descricao": "Obtém uma tarefa específica"},
            {"metodo": "POST", "url": "/tarefas", "descricao": "Cria uma nova tarefa"},
            {"metodo": "PUT", "url": "/tarefas/<id>", "descricao": "Atualiza uma tarefa existente"},
            {"metodo": "DELETE", "url": "/tarefas/<id>", "descricao": "Exclui uma tarefa"},
            {"metodo": "GET", "url": "/tarefas/concluidas", "descricao": "Lista tarefas concluídas"},
            {"metodo": "GET", "url": "/tarefas/pendentes", "descricao": "Lista tarefas pendentes"}
        ]
    })
```

```python
# Código para executar a aplicação
if __name__ == '__main__':
    # Pré-populando com algumas tarefas para teste
    tarefas.append({
        'id': str(uuid.uuid4()),
        'titulo': 'Estudar Python',
        'descricao': 'Completar o módulo intermediário do curso',
        'concluida': False,
        'criada_em': datetime.now().isoformat(),
        'atualizada_em': datetime.now().isoformat()
    })

    tarefas.append({
        'id': str(uuid.uuid4()),
        'titulo': 'Fazer exercícios',
        'descricao': 'Resolver os exercícios práticos do módulo',
        'concluida': True,
        'criada_em': datetime.now().isoformat(),
        'atualizada_em': datetime.now().isoformat()
    })

    print("Iniciando a API de Tarefas...")
    print("Acesse http://localhost:5000/ para ver a documentação da API")
    app.run(debug=True)
```

Exercício 10: Jogo da Forca

Crie um jogo da forca em Python, onde o jogador tenta adivinhar uma palavra oculta, letra por letra.

```python
# Solução
import random
import os

class JogoDaForca:
    def __init__(self):
        self.palavras = [
            "python", "programacao", "computador", "desenvolvimento",
            "algoritmo", "variavel", "funcao", "classe", "objeto",
            "dicionario", "lista", "tupla", "conjunto", "arquivo",
            "excecao", "modulo", "pacote", "biblioteca", "framework"
        ]
        self.palavra_secreta = ""
        self.palavra_exibida = []
        self.letras_tentadas = []
        self.tentativas_restantes = 6
        self.mensagem = ""
        self.jogo_terminado = False

    def iniciar_jogo(self):
        """Inicializa um novo jogo."""
        self.palavra_secreta = random.choice(self.palavras)
        self.palavra_exibida = ["_" for _ in self.palavra_secreta]
        self.letras_tentadas = []
```

```
        self.tentativas_restantes = 6
        self.mensagem = "Bem-vindo ao Jogo da Forca! Tente adivinhar a palavra."
        self.jogo_terminado = False

    def tentar_letra(self, letra):
        """Processa uma tentativa de letra."""
        # Verifica se o jogo já terminou
        if self.jogo_terminado:
            self.mensagem = "O jogo já terminou. Inicie um novo jogo."
            return

        # Verifica se a entrada é válida
        if not letra.isalpha() or len(letra) != 1:
            self.mensagem = "Por favor, digite apenas uma letra."
            return

        letra = letra.lower()

        # Verifica se a letra já foi tentada
        if letra in self.letras_tentadas:
            self.mensagem = f"Você já tentou a letra '{letra}'."
            return

        # Adiciona a letra às tentativas
        self.letras_tentadas.append(letra)

        # Verifica se a letra está na palavra
        if letra in self.palavra_secreta:
            # Atualiza a palavra exibida
            for i, char in enumerate(self.palavra_secreta):
                if char == letra:
                    self.palavra_exibida[i] = letra

            self.mensagem = f"Boa! A letra '{letra}' está na palavra."

            # Verifica se o jogador ganhou
            if "_" not in self.palavra_exibida:
                self.mensagem = f"Parabéns! Você ganhou! A palavra era '{self.palavra_secreta}'."
                self.jogo_terminado = True
        else:
            # Reduz o número de tentativas
            self.tentativas_restantes -= 1
            self.mensagem = f"Ops! A letra '{letra}' não está na palavra."

            # Verifica se o jogador perdeu
            if self.tentativas_restantes == 0:
                self.mensagem = f"Game Over! Você perdeu. A palavra era '{self.palavra_secreta}'."
                self.jogo_terminado = True

    def desenhar_forca(self):
        """Retorna o desenho da forca baseado no número de tentativas restantes."""
        estagios = [
            # Estágio 6 (inicial)
```

```
        """
          +---+
          |   |
              |
              |
              |
              |
        =========
        """,
        # Estágio 5
        """
          +---+
          |   |
          O   |
              |
              |
              |
        =========
        """,
        # Estágio 4
        """
          +---+
          |   |
          O   |
          |   |
              |
              |
        =========
        """,
        # Estágio 3
        """
          +---+
          |   |
          O   |
         /|   |
              |
              |
        =========
        """,
        # Estágio 2
        """
          +---+
          |   |
          O   |
         /|\\  |
              |
              |
        =========
        """,
        # Estágio 1
        """
          +---+
          |   |
```

```
              O   |
             /|\\  |
             /    |
                  |
            =========
            """,
            # Estágio 0 (final)
            """
              +---+
              |   |
              O   |
             /|\\  |
             / \\  |
                  |
            =========
            """
        ]
        return estagios[6 - self.tentativas_restantes]

    def exibir_status(self):
        """Exibe o status atual do jogo."""
        os.system('cls' if os.name == 'nt' else 'clear')  # Limpa a tela

        print("=== JOGO DA FORCA ===")
        print(self.desenhar_forca())
        print(f"Palavra: {' '.join(self.palavra_exibida)}")
        print(f"Letras tentadas: {', '.join(sorted(self.letras_tentadas))}")
        print(f"Tentativas restantes: {self.tentativas_restantes}")
        print(f"\nMensagem: {self.mensagem}")

    def jogar(self):
        """Inicia o loop principal do jogo."""
        self.iniciar_jogo()

        while True:
            self.exibir_status()

            if self.jogo_terminado:
                jogar_novamente = input("\nDeseja jogar novamente? (s/n): ").lower()
                if jogar_novamente == 's':
                    self.iniciar_jogo()
                else:
                    print("Obrigado por jogar! Até a próxima.")
                    break
            else:
                letra = input("\nDigite uma letra: ")
                self.tentar_letra(letra)

def main():
    jogo = JogoDaForca()
    jogo.jogar()

if __name__ == "__main__":
```

```
    main()
```

Estes exercícios foram projetados para reforçar os conceitos intermediários de Python que aprendemos neste módulo. Recomendo que você tente resolver cada exercício por conta própria antes de olhar a solução. A prática constante é a chave para se tornar um programador proficiente.

No próximo módulo, exploraremos tópicos avançados de Python, incluindo decoradores, geradores, expressões regulares, programação funcional, concorrência e paralelismo.

Módulo 3: Python Avançado

Módulo 3: Python Avançado

Decoradores

Os decoradores são uma poderosa ferramenta em Python que permite modificar o comportamento de funções e classes de forma elegante e não intrusiva. Eles são amplamente utilizados em frameworks como Flask, Django e muitas outras bibliotecas Python. Neste tópico, exploraremos o conceito de decoradores, como eles funcionam e como criar seus próprios decoradores.

O que são Decoradores?

Um decorador é uma função que recebe outra função como argumento, adiciona alguma funcionalidade e retorna uma nova função, sem modificar a função original. Em essência, os decoradores "envolvem" uma função, modificando seu comportamento.

A sintaxe para aplicar um decorador a uma função é usar o símbolo `@` seguido do nome do decorador, colocado acima da definição da função:

```
@decorador
def funcao():
    pass
```

Isso é equivalente a:

```
def funcao():
    pass
funcao = decorador(funcao)
```

Funções como Objetos de Primeira Classe

Para entender os decoradores, é importante lembrar que em Python, funções são objetos de primeira classe, o que significa que elas podem ser:

1. Atribuídas a variáveis
2. Passadas como argumentos para outras funções
3. Retornadas por outras funções
4. Armazenadas em estruturas de dados

Exemplo:

```
def saudacao(nome):
    return f"Olá, {nome}!"

# Atribuindo a função a uma variável
dizer_ola = saudacao
```

```
# Usando a variável como uma função
print(dizer_ola("Maria"))  # Olá, Maria!
```

Funções Aninhadas

Os decoradores frequentemente usam funções aninhadas (funções definidas dentro de outras funções):

```
def funcao_externa(x):
    def funcao_interna(y):
        return x + y
    return funcao_interna

# Criando uma função que adiciona 10 a um número
adicionar_10 = funcao_externa(10)
print(adicionar_10(5))  # 15
```

Criando Decoradores Simples

Vamos criar um decorador simples que mede o tempo de execução de uma função:

```
import time

def medir_tempo(funcao):
    def wrapper(*args, **kwargs):
        inicio = time.time()
        resultado = funcao(*args, **kwargs)
        fim = time.time()
        print(f"A função {funcao.__name__} levou {fim - inicio:.4f} segundos para executar.")
        return resultado
    return wrapper

@medir_tempo
def operacao_lenta():
    print("Iniciando operação...")
    time.sleep(2)  # Simula uma operação que leva 2 segundos
    print("Operação concluída.")

operacao_lenta()
```

Neste exemplo: 1. `medir_tempo` é o decorador que recebe uma função como argumento. 2. `wrapper` é uma função interna que envolve a função original, adicionando a funcionalidade de medir o tempo. 3. `*args` e `**kwargs` permitem que o wrapper aceite qualquer número de argumentos posicionais e nomeados. 4. O decorador é aplicado à função `operacao_lenta` usando a sintaxe `@medir_tempo`.

Decoradores com Argumentos

Podemos criar decoradores que aceitam argumentos, o que requer um nível adicional de aninhamento:

```
def repetir(n):
    def decorador(funcao):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                resultado = funcao(*args, **kwargs)
            return resultado
        return wrapper
```

```
    return decorador

@repetir(3)
def dizer_oi(nome):
    print(f"Oi, {nome}!")
    return nome

dizer_oi("João")  # Imprime "Oi, João!" três vezes
```

Neste exemplo: 1. `repetir` é uma função que retorna o decorador real. 2. `decorador` é o decorador que recebe a função a ser decorada. 3. `wrapper` é a função que envolve a função original. 4. O decorador é aplicado com um argumento: `@repetir(3)`.

Preservando Metadados da Função

Quando decoramos uma função, perdemos alguns de seus metadados, como o nome (`__name__`), a docstring (`__doc__`) e a assinatura. O módulo `functools` fornece o decorador `wraps` para preservar esses metadados:

```
import functools

def meu_decorador(funcao):
    @functools.wraps(funcao)  # Preserva os metadados da função original
    def wrapper(*args, **kwargs):
        print("Antes da função")
        resultado = funcao(*args, **kwargs)
        print("Depois da função")
        return resultado
    return wrapper

@meu_decorador
def saudacao(nome):
    """Retorna uma saudação para o nome fornecido."""
    return f"Olá, {nome}!"

print(saudacao.__name__)  # saudacao (sem @functools.wraps seria 'wrapper')
print(saudacao.__doc__)   # Retorna uma saudação para o nome fornecido.
```

Decoradores de Classe

Além de decorar funções, podemos decorar classes inteiras:

```
def adicionar_metodo(cls):
    def novo_metodo(self):
        return "Este é um novo método!"
    cls.novo_metodo = novo_metodo
    return cls

@adicionar_metodo
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor

    def metodo_original(self):
        return f"Valor: {self.valor}"
```

```
obj = MinhaClasse(42)
print(obj.metodo_original())  # Valor: 42
print(obj.novo_metodo())      # Este é um novo método!
```

Classes como Decoradores

Também podemos usar classes como decoradores, implementando o método `__call__`:

```
class ContarChamadas:
    def __init__(self, funcao):
        self.funcao = funcao
        self.contagem = 0
        functools.update_wrapper(self, funcao)  # Preserva metadados

    def __call__(self, *args, **kwargs):
        self.contagem += 1
        print(f"A função {self.funcao.__name__} foi chamada {self.contagem} vezes.")
        return self.funcao(*args, **kwargs)

@ContarChamadas
def dizer_ola():
    return "Olá!"

print(dizer_ola())  # A função dizer_ola foi chamada 1 vezes. Olá!
print(dizer_ola())  # A função dizer_ola foi chamada 2 vezes. Olá!
```

Decoradores Encadeados

Podemos aplicar múltiplos decoradores a uma única função. Eles são aplicados de baixo para cima:

```
def decorador1(funcao):
    def wrapper(*args, **kwargs):
        print("Decorador 1 - Antes")
        resultado = funcao(*args, **kwargs)
        print("Decorador 1 - Depois")
        return resultado
    return wrapper

def decorador2(funcao):
    def wrapper(*args, **kwargs):
        print("Decorador 2 - Antes")
        resultado = funcao(*args, **kwargs)
        print("Decorador 2 - Depois")
        return resultado
    return wrapper

@decorador1
@decorador2
def minha_funcao():
    print("Função principal")

minha_funcao()
```

Saída:

```
Decorador 1 - Antes
Decorador 2 - Antes
Função principal
Decorador 2 - Depois
Decorador 1 - Depois
```

Casos de Uso Comuns para Decoradores

1. **Logging**: Registrar informações sobre chamadas de função.

```python
def log(funcao):
    @functools.wraps(funcao)
    def wrapper(*args, **kwargs):
        print(f"Chamando {funcao.__name__} com args={args}, kwargs={kwargs}")
        resultado = funcao(*args, **kwargs)
        print(f"{funcao.__name__} retornou {resultado}")
        return resultado
    return wrapper

@log
def soma(a, b):
    return a + b

soma(3, 5)
```

2. **Verificação de Permissões**: Verificar se um usuário tem permissão para executar uma função.

```python
def requer_permissao(permissao):
    def decorador(funcao):
        @functools.wraps(funcao)
        def wrapper(usuario, *args, **kwargs):
            if permissao in usuario.permissoes:
                return funcao(usuario, *args, **kwargs)
            else:
                raise PermissionError(f"Usuário não tem permissão '{permissao}'")
        return wrapper
    return decorador

class Usuario:
    def __init__(self, nome, permissoes):
        self.nome = nome
        self.permissoes = permissoes

@requer_permissao("admin")
def funcao_administrativa(usuario):
    return f"{usuario.nome} executou função administrativa"

usuario1 = Usuario("Alice", ["admin", "user"])
usuario2 = Usuario("Bob", ["user"])

print(funcao_administrativa(usuario1))  # Alice executou função administrativa
# print(funcao_administrativa(usuario2))  # Levanta PermissionError
```

3. **Memoização (Cache)**: Armazenar resultados de chamadas de função para evitar recálculos.

```
def memoize(funcao):
    cache = {}
    @functools.wraps(funcao)
    def wrapper(*args):
        if args in cache:
            return cache[args]
        resultado = funcao(*args)
        cache[args] = resultado
        return resultado
    return wrapper

@memoize
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(35))  # Muito mais rápido com memoização
```

4. **Validação de Entrada**: Verificar se os argumentos passados para uma função são válidos.

```
def validar_positivo(funcao):
    @functools.wraps(funcao)
    def wrapper(x, *args, **kwargs):
        if x <= 0:
            raise ValueError("O argumento deve ser positivo")
        return funcao(x, *args, **kwargs)
    return wrapper

@validar_positivo
def calcular_raiz_quadrada(x):
    return x ** 0.5

print(calcular_raiz_quadrada(16))  # 4.0
# print(calcular_raiz_quadrada(-4))  # Levanta ValueError
```

5. **Sincronização**: Garantir que apenas um thread execute uma função por vez.

```
import threading

def sincronizado(funcao):
    lock = threading.Lock()
    @functools.wraps(funcao)
    def wrapper(*args, **kwargs):
        with lock:
            return funcao(*args, **kwargs)
    return wrapper

@sincronizado
def incrementar_contador(contador):
    contador["valor"] += 1
    return contador["valor"]
```

Decoradores Integrados do Python

Python possui alguns decoradores integrados:

1. **@property**: Transforma um método em um atributo de leitura.

```
class Temperatura:
    def __init__(self, celsius=0):
        self._celsius = celsius

    @property
    def celsius(self):
        return self._celsius

    @celsius.setter
    def celsius(self, valor):
        if valor < -273.15:
            raise ValueError("Temperatura abaixo do zero absoluto!")
        self._celsius = valor

    @property
    def fahrenheit(self):
        return self._celsius * 9/5 + 32

    @fahrenheit.setter
    def fahrenheit(self, valor):
        self.celsius = (valor - 32) * 5/9

temp = Temperatura(25)
print(temp.celsius)     # 25
print(temp.fahrenheit)  # 77.0
temp.celsius = 30
print(temp.fahrenheit)  # 86.0
```

2. **@classmethod**: Define um método de classe que opera na classe em vez de instâncias.

```
class Data:
    def __init__(self, dia, mes, ano):
        self.dia = dia
        self.mes = mes
        self.ano = ano

    @classmethod
    def de_string(cls, data_str):
        dia, mes, ano = map(int, data_str.split('/'))
        return cls(dia, mes, ano)

    def __str__(self):
        return f"{self.dia:02d}/{self.mes:02d}/{self.ano}"

data = Data.de_string("15/04/2023")
print(data)  # 15/04/2023
```

3. **@staticmethod**: Define um método estático que não opera nem na classe nem em instâncias.

```
class MathUtils:
    @staticmethod
```

```
    def e_par(numero):
        return numero % 2 == 0

    @staticmethod
    def e_primo(numero):
        if numero <= 1:
            return False
        for i in range(2, int(numero**0.5) + 1):
            if numero % i == 0:
                return False
        return True

print(MathUtils.e_par(4))    # True
print(MathUtils.e_primo(7))  # True
```

Boas Práticas com Decoradores

1. **Use `functools.wraps`**: Sempre use `@functools.wraps` para preservar os metadados da função original.
2. **Mantenha os Decoradores Simples**: Decoradores devem fazer uma coisa e fazê-la bem.
3. **Documente seus Decoradores**: Explique claramente o que seu decorador faz e como usá-lo.
4. **Considere o Desempenho**: Decoradores adicionam uma camada de indireção, o que pode afetar o desempenho em código crítico.
5. **Teste seus Decoradores**: Escreva testes unitários para garantir que seus decoradores funcionem corretamente.

Conclusão

Os decoradores são uma ferramenta poderosa em Python que permite adicionar funcionalidades a funções e classes de forma limpa e reutilizável. Eles são amplamente utilizados em frameworks e bibliotecas, e dominar seu uso é essencial para escrever código Python avançado e elegante.

No próximo tópico, exploraremos geradores e iteradores, que são outra característica poderosa do Python para trabalhar com sequências de dados de forma eficiente.

Módulo 3: Python Avançado

Geradores e Iteradores

Geradores e iteradores são recursos poderosos em Python que permitem trabalhar com sequências de dados de forma eficiente, especialmente quando lidamos com grandes conjuntos de dados. Eles são fundamentais para escrever código que utiliza memória de forma eficiente e segue o paradigma de programação lazy (preguiçosa), onde os valores são calculados apenas quando necessário.

Iteradores

O que são Iteradores?

Um iterador é um objeto que implementa o protocolo de iteração, que consiste em dois métodos:

1. `__iter__()`: Retorna o próprio objeto iterador.
2. `__next__()`: Retorna o próximo item da sequência. Quando não há mais itens, levanta a exceção `StopIteration`.

Em Python, muitos objetos são iteráveis por padrão, como listas, tuplas, dicionários, conjuntos e strings. Quando usamos um loop `for` para percorrer um objeto iterável, o Python automaticamente cria um iterador a partir desse objeto.

Criando um Iterador Personalizado

Vamos criar um iterador simples que gera uma sequência de números:

```
class Contador:
    def __init__(self, inicio, fim):
        self.inicio = inicio
        self.fim = fim
        self.valor = inicio - 1  # Começamos um antes para que o primeiro next() retorne o valor inicia

    def __iter__(self):
        return self

    def __next__(self):
        self.valor += 1
        if self.valor > self.fim:
            raise StopIteration
        return self.valor

# Usando o iterador
contador = Contador(1, 5)
for numero in contador:
    print(numero)  # Imprime 1, 2, 3, 4, 5
```

Iterando Manualmente

Podemos usar a função `next()` para obter o próximo item de um iterador manualmente:

```
contador = Contador(1, 3)
iterador = iter(contador)  # Obtém o iterador chamando __iter__()

print(next(iterador))  # 1
print(next(iterador))  # 2
print(next(iterador))  # 3
# print(next(iterador))  # Levanta StopIteration
```

Iteradores Infinitos

Podemos criar iteradores que geram sequências infinitas, mas devemos ter cuidado para não tentar consumir todos os elementos:

```
class ContadorInfinito:
    def __init__(self, inicio=0):
        self.valor = inicio

    def __iter__(self):
        return self

    def __next__(self):
        valor_atual = self.valor
        self.valor += 1
        return valor_atual

# Usando o iterador infinito com limitação
contador = ContadorInfinito()
for numero in contador:
    print(numero)
    if numero >= 10:
        break  # Importante: interrompe o loop para evitar um loop infinito
```

Geradores

O que são Geradores?

Geradores são uma forma simplificada de criar iteradores. Em vez de implementar uma classe com os métodos `__iter__()` e `__next__()`, podemos definir uma função que usa a palavra-chave `yield` para retornar valores um de cada vez.

Quando uma função com `yield` é chamada, ela retorna um objeto gerador, que é um tipo especial de iterador. A execução da função é pausada no `yield` e retomada na próxima chamada a `next()`.

Criando um Gerador Simples

```
def contador(inicio, fim):
    valor = inicio
    while valor <= fim:
        yield valor
        valor += 1

# Usando o gerador
```

```python
for numero in contador(1, 5):
    print(numero)  # Imprime 1, 2, 3, 4, 5
```

Este gerador é equivalente ao iterador `Contador` que definimos anteriormente, mas com muito menos código.

Geradores como Iteradores

Os geradores são iteradores, então podemos usar a função `next()` com eles:

```python
gerador = contador(1, 3)
print(next(gerador))  # 1
print(next(gerador))  # 2
print(next(gerador))  # 3
# print(next(gerador))  # Levanta StopIteration
```

Expressões Geradoras

Assim como temos compreensões de lista, Python também suporta expressões geradoras, que são uma forma concisa de criar geradores:

```python
# Compreensão de lista (cria toda a lista na memória)
quadrados_lista = [x**2 for x in range(1, 6)]
print(quadrados_lista)  # [1, 4, 9, 16, 25]

# Expressão geradora (gera valores sob demanda)
quadrados_gerador = (x**2 for x in range(1, 6))
print(quadrados_gerador)  # <generator object <genexpr> at 0x...>

for quadrado in quadrados_gerador:
    print(quadrado)  # Imprime 1, 4, 9, 16, 25
```

A principal diferença é que a compreensão de lista cria toda a lista na memória de uma vez, enquanto a expressão geradora cria um gerador que produz os valores sob demanda.

Geradores Infinitos

Assim como os iteradores, podemos criar geradores que produzem sequências infinitas:

```python
def contador_infinito(inicio=0):
    valor = inicio
    while True:
        yield valor
        valor += 1

# Usando o gerador infinito com limitação
for numero in contador_infinito():
    print(numero)
    if numero >= 10:
        break  # Importante: interrompe o loop para evitar um loop infinito
```

Enviando Valores para Geradores

Os geradores não apenas produzem valores, mas também podem receber valores de volta usando o método `send()`:

```python
def eco():
    valor = yield
    while True:
        valor = yield f"Eco: {valor}"

gerador = eco()
next(gerador)  # Inicializa o gerador até o primeiro yield
print(gerador.send("Olá"))  # Eco: Olá
print(gerador.send("Mundo"))  # Eco: Mundo
```

Geradores com Múltiplos Yields

Um gerador pode ter múltiplos `yield`s, e a execução será pausada em cada um deles:

```python
def multiplos_yields():
    yield "Primeiro"
    yield "Segundo"
    yield "Terceiro"

for valor in multiplos_yields():
    print(valor)  # Imprime Primeiro, Segundo, Terceiro
```

Delegando para Subgeradores (yield from)

A partir do Python 3.3, podemos usar `yield from` para delegar para um subgerador:

```python
def subgerador():
    yield 1
    yield 2
    yield 3

def gerador_principal():
    yield "A"
    yield from subgerador()  # Delega para o subgerador
    yield "B"

for valor in gerador_principal():
    print(valor)  # Imprime A, 1, 2, 3, B
```

Vantagens dos Geradores

Eficiência de Memória

Uma das principais vantagens dos geradores é a eficiência de memória. Eles geram valores sob demanda, em vez de criar toda a sequência na memória:

```python
# Função que retorna uma lista (usa muita memória para grandes valores de n)
def numeros_lista(n):
    resultado = []
    for i in range(n):
        resultado.append(i)
    return resultado

# Gerador (usa pouca memória independentemente do valor de n)
def numeros_gerador(n):
```

```
    for i in range(n):
        yield i

# Para n pequeno, ambos funcionam bem
for numero in numeros_lista(10):
    print(numero, end=" ")  # 0 1 2 3 4 5 6 7 8 9

print()

for numero in numeros_gerador(10):
    print(numero, end=" ")  # 0 1 2 3 4 5 6 7 8 9

# Para n grande, o gerador é muito mais eficiente
# numeros_lista(10**8)  # Pode consumir muita memória
numeros_gerador(10**8)  # Usa pouca memória
```

Avaliação Preguiçosa (Lazy Evaluation)

Os geradores implementam avaliação preguiçosa, calculando valores apenas quando necessário:

```
def numeros_primos():
    """Gerador infinito de números primos."""
    primos = []
    n = 2
    while True:
        if all(n % p != 0 for p in primos):
            primos.append(n)
            yield n
        n += 1

# Obtendo os primeiros 10 números primos
gerador = numeros_primos()
for _ in range(10):
    print(next(gerador), end=" ")  # 2 3 5 7 11 13 17 19 23 29
```

Composição de Geradores

Podemos compor geradores para criar pipelines de processamento de dados:

```
def numeros(inicio, fim):
    """Gera números de inicio a fim."""
    for i in range(inicio, fim + 1):
        yield i

def quadrados(numeros):
    """Gera o quadrado de cada número."""
    for n in numeros:
        yield n ** 2

def pares(numeros):
    """Filtra apenas os números pares."""
    for n in numeros:
        if n % 2 == 0:
            yield n
```

```python
# Criando um pipeline: gera números de 1 a 10, calcula seus quadrados e filtra os pares
pipeline = pares(quadrados(numeros(1, 10)))
for numero in pipeline:
    print(numero, end=" ")  # 4 16 36 64 100
```

Casos de Uso Comuns

Processamento de Arquivos Grandes

Os geradores são ideais para processar arquivos grandes linha por linha, sem carregar todo o arquivo na memória:

```python
def ler_arquivo_grande(nome_arquivo):
    with open(nome_arquivo, 'r') as arquivo:
        for linha in arquivo:
            yield linha.strip()

# Processando um arquivo grande linha por linha
for linha in ler_arquivo_grande("arquivo_grande.txt"):
    # Processa cada linha
    pass
```

Pipelines de Processamento de Dados

Geradores são excelentes para criar pipelines de processamento de dados:

```python
def ler_csv(nome_arquivo):
    with open(nome_arquivo, 'r') as arquivo:
        for linha in arquivo:
            yield linha.strip().split(',')

def filtrar_dados(linhas, coluna, valor):
    for linha in linhas:
        if linha[coluna] == valor:
            yield linha

def calcular_media(linhas, coluna):
    total = 0
    count = 0
    for linha in linhas:
        total += float(linha[coluna])
        count += 1
    return total / count if count > 0 else 0

# Exemplo de uso
linhas = ler_csv("dados.csv")
dados_filtrados = filtrar_dados(linhas, 0, "categoria_a")
media = calcular_media(dados_filtrados, 1)
print(f"Média: {media}")
```

Sequências Infinitas

Geradores são perfeitos para representar sequências infinitas:

```python
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Obtendo os primeiros 10 números de Fibonacci
fib = fibonacci()
for _ in range(10):
    print(next(fib), end=" ")  # 0 1 1 2 3 5 8 13 21 34
```

Itertools: Biblioteca de Ferramentas para Iteradores

O módulo `itertools` da biblioteca padrão do Python fornece uma coleção de ferramentas para trabalhar com iteradores:

```python
import itertools

# count: contador infinito
for i in itertools.islice(itertools.count(10, 2), 5):
    print(i, end=" ")  # 10 12 14 16 18

print()

# cycle: cicla infinitamente por um iterável
ciclo = itertools.cycle(["A", "B", "C"])
for _ in range(7):
    print(next(ciclo), end=" ")  # A B C A B C A

print()

# repeat: repete um elemento
for i in itertools.repeat("X", 5):
    print(i, end=" ")  # X X X X X

print()

# chain: concatena iteráveis
for i in itertools.chain([1, 2], [3, 4], [5, 6]):
    print(i, end=" ")  # 1 2 3 4 5 6

print()

# zip_longest: combina elementos de iteráveis
for i in itertools.zip_longest([1, 2], [3, 4, 5], fillvalue=0):
    print(i, end=" ")  # (1, 3) (2, 4) (0, 5)

print()

# product: produto cartesiano
for i in itertools.product("AB", "12"):
    print("".join(i), end=" ")  # A1 A2 B1 B2

print()
```

```python
# permutations: todas as permutações
for i in itertools.permutations("ABC", 2):
    print("".join(i), end=" ")  # AB AC BA BC CA CB

print()

# combinations: todas as combinações
for i in itertools.combinations("ABC", 2):
    print("".join(i), end=" ")  # AB AC BC
```

Boas Práticas

1. **Use Geradores para Grandes Conjuntos de Dados**: Prefira geradores a listas quando trabalhar com grandes conjuntos de dados.
2. **Composição de Geradores**: Use a composição de geradores para criar pipelines de processamento de dados.
3. **Evite Consumir Geradores Infinitos**: Sempre use limitações (como `break` ou `itertools.islice`) ao trabalhar com geradores infinitos.
4. **Documente seus Geradores**: Explique claramente o que seu gerador produz e quaisquer efeitos colaterais.
5. **Use `yield from` para Delegação**: Use `yield from` para delegar para subgeradores, em vez de iterar manualmente sobre eles.

Conclusão

Geradores e iteradores são ferramentas poderosas em Python que permitem trabalhar com sequências de dados de forma eficiente. Eles são especialmente úteis para processar grandes conjuntos de dados, criar pipelines de processamento e representar sequências infinitas. Dominar o uso de geradores e iteradores é essencial para escrever código Python eficiente e elegante.

No próximo tópico, exploraremos expressões regulares, que são uma ferramenta poderosa para trabalhar com padrões em strings.

Módulo 3: Python Avançado

Expressões Regulares

As expressões regulares (regex ou regexp) são sequências de caracteres que formam um padrão de busca, principalmente utilizadas para operações de busca e substituição em strings. Em Python, o módulo `re` fornece suporte completo para expressões regulares, permitindo que você realize operações complexas de manipulação de texto de forma eficiente.

Introdução às Expressões Regulares

As expressões regulares são uma linguagem concisa e flexível para identificar strings de texto, como caracteres específicos, palavras ou padrões de caracteres. Elas são amplamente utilizadas em validação de formulários, extração de informações de textos, substituição de texto, análise de logs e muito mais.

O Módulo `re` em Python

Para usar expressões regulares em Python, primeiro precisamos importar o módulo `re`:

```
import re
```

O módulo `re` fornece várias funções para trabalhar com expressões regulares:

- `re.search()`: Procura um padrão em uma string
- `re.match()`: Verifica se o padrão está no início da string
- `re.findall()`: Encontra todas as ocorrências do padrão
- `re.finditer()`: Retorna um iterador com todas as ocorrências
- `re.sub()`: Substitui ocorrências do padrão
- `re.split()`: Divide a string pelo padrão
- `re.compile()`: Compila um padrão para reutilização

Padrões Básicos

Caracteres Literais

O caso mais simples é a correspondência de caracteres literais:

```
import re

texto = "Python é uma linguagem de programação poderosa."
padrao = "Python"
resultado = re.search(padrao, texto)

if resultado:
    print(f"Encontrado: {resultado.group()}")  # Encontrado: Python
    print(f"Posição: {resultado.start()}")     # Posição: 0
```

Metacaracteres

Os metacaracteres são caracteres com significado especial:

- `.` - Corresponde a qualquer caractere, exceto nova linha
- `^` - Corresponde ao início da string
- `$` - Corresponde ao fim da string
- `*` - Corresponde a 0 ou mais repetições
- `+` - Corresponde a 1 ou mais repetições
- `?` - Corresponde a 0 ou 1 repetição
- `{m}` - Corresponde a exatamente m repetições
- `{m,n}` - Corresponde de m a n repetições
- `[]` - Conjunto de caracteres
- `|` - Alternação (ou)
- `()` - Grupo
- `\` - Escape ou sequência especial

Exemplos:

```
# . (ponto) - qualquer caractere
re.search("Py...n", "Python")  # Corresponde a "Python"

# ^ (circunflexo) - início da string
re.search("^Python", "Python é legal")  # Corresponde a "Python"
re.search("^é", "Python é legal")       # Não corresponde

# $ (cifrão) - fim da string
re.search("legal$", "Python é legal")  # Corresponde a "legal"
re.search("Python$", "Python é legal") # Não corresponde

# * (asterisco) - 0 ou mais repetições
re.search("ab*c", "ac")     # Corresponde a "ac"
re.search("ab*c", "abc")    # Corresponde a "abc"
re.search("ab*c", "abbc")   # Corresponde a "abbc"

# + (mais) - 1 ou mais repetições
re.search("ab+c", "ac")     # Não corresponde
re.search("ab+c", "abc")    # Corresponde a "abc"
re.search("ab+c", "abbc")   # Corresponde a "abbc"

# ? (interrogação) - 0 ou 1 repetição
re.search("ab?c", "ac")     # Corresponde a "ac"
re.search("ab?c", "abc")    # Corresponde a "abc"
re.search("ab?c", "abbc")   # Não corresponde

# {m} - exatamente m repetições
re.search("ab{2}c", "abc")   # Não corresponde
re.search("ab{2}c", "abbc")  # Corresponde a "abbc"

# {m,n} - de m a n repetições
re.search("ab{1,3}c", "abc")    # Corresponde a "abc"
re.search("ab{1,3}c", "abbc")   # Corresponde a "abbc"
re.search("ab{1,3}c", "abbbc")  # Corresponde a "abbbc"
re.search("ab{1,3}c", "abbbbc") # Não corresponde

# [] - conjunto de caracteres
```

```
re.search("[aeiou]", "python")  # Corresponde a "o"
re.search("[0-9]", "python2")   # Corresponde a "2"

# | (pipe) - alternação
re.search("python|java", "python é legal")  # Corresponde a "python"
re.search("python|java", "java é legal")    # Corresponde a "java"

# () - grupo
resultado = re.search("(py)?thon", "python")
if resultado:
    print(resultado.group(0))  # python
    print(resultado.group(1))  # py

# \ (barra invertida) - escape
re.search("\.", "python.")  # Corresponde a "."
```

Classes de Caracteres

As classes de caracteres permitem especificar um conjunto de caracteres:

- `[abc]` - Corresponde a qualquer dos caracteres a, b ou c
- `[^abc]` - Corresponde a qualquer caractere exceto a, b ou c
- `[a-z]` - Corresponde a qualquer caractere entre a e z
- `[0-9]` - Corresponde a qualquer dígito
- `[a-zA-Z]` - Corresponde a qualquer letra

Exemplos:

```
# Encontrar todas as vogais
re.findall("[aeiou]", "python")  # ['o']

# Encontrar todos os caracteres que não são vogais
re.findall("[^aeiou]", "python")  # ['p', 'y', 't', 'h', 'n']

# Encontrar todas as letras minúsculas
re.findall("[a-z]", "Python3")  # ['y', 't', 'h', 'o', 'n']

# Encontrar todos os dígitos
re.findall("[0-9]", "Python3")  # ['3']
```

Sequências Especiais

Python fornece várias sequências especiais que representam classes de caracteres comuns:

- `\d` - Corresponde a qualquer dígito (`[0-9]`)
- `\D` - Corresponde a qualquer caractere que não seja dígito (`[^0-9]`)
- `\s` - Corresponde a qualquer espaço em branco (`[ \t\n\r\f\v]`)
- `\S` - Corresponde a qualquer caractere que não seja espaço em branco (`[^ \t\n\r\f\v]`)
- `\w` - Corresponde a qualquer caractere alfanumérico (`[a-zA-Z0-9_]`)
- `\W` - Corresponde a qualquer caractere que não seja alfanumérico (`[^a-zA-Z0-9_]`)
- `\b` - Corresponde a uma borda de palavra
- `\B` - Corresponde a qualquer posição que não seja uma borda de palavra

Exemplos:

```
# \d - dígitos
re.findall(r"\d", "Python 3.9")  # ['3', '9']

# \D - não-dígitos
re.findall(r"\D", "Python 3.9")  # ['P', 'y', 't', 'h', 'o', 'n', ' ', '.']

# \s - espaços em branco
re.findall(r"\s", "Python 3.9")  # [' ']

# \S - não-espaços em branco
re.findall(r"\S", "Python 3.9")  # ['P', 'y', 't', 'h', 'o', 'n', '3', '.', '9']

# \w - caracteres alfanuméricos
re.findall(r"\w", "Python 3.9!")  # ['P', 'y', 't', 'h', 'o', 'n', '3', '9']

# \W - não-alfanuméricos
re.findall(r"\W", "Python 3.9!")  # [' ', '.', '!']

# \b - borda de palavra
re.findall(r"\bPy\b", "Python Py")  # ['Py']

# \B - não-borda de palavra
re.findall(r"\BPy\B", "Python")  # []
```

Nota: É uma boa prática usar strings brutas (`r"..."`) ao definir padrões de expressão regular para evitar problemas com sequências de escape.

Funções Principais do Módulo `re`

re.search()

A função `search()` procura um padrão em toda a string e retorna o primeiro match encontrado:

```
import re

texto = "Python é uma linguagem de programação poderosa."
padrao = "linguagem"
resultado = re.search(padrao, texto)

if resultado:
    print(f"Encontrado: {resultado.group()}")  # Encontrado: linguagem
    print(f"Posição: {resultado.start()}-{resultado.end()}")  # Posição: 13-22
```

re.match()

A função `match()` verifica se o padrão está no início da string:

```
# match no início da string
resultado = re.match("Python", "Python é legal")
if resultado:
    print(f"Match: {resultado.group()}")  # Match: Python

# match falha se não estiver no início
resultado = re.match("é", "Python é legal")
if resultado:
```

```
    print(f"Match: {resultado.group()}")  # Não imprime nada
```

re.findall()

A função `findall()` encontra todas as ocorrências do padrão e retorna uma lista:

```
texto = "Python é uma linguagem de programação. Python é divertido."
padrao = "Python"
ocorrencias = re.findall(padrao, texto)
print(ocorrencias)  # ['Python', 'Python']

# Encontrando todas as palavras
palavras = re.findall(r"\b\w+\b", texto)
print(palavras)  # ['Python', 'é', 'uma', 'linguagem', 'de', 'programação', 'Python', 'é', 'divertido']
```

re.finditer()

A função `finditer()` é semelhante a `findall()`, mas retorna um iterador de objetos match:

```
texto = "Python é uma linguagem de programação. Python é divertido."
padrao = "Python"
ocorrencias = re.finditer(padrao, texto)

for match in ocorrencias:
    print(f"Encontrado '{match.group()}' na posição {match.start()}-{match.end()}")
# Encontrado 'Python' na posição 0-6
# Encontrado 'Python' na posição 36-42
```

re.sub()

A função `sub()` substitui ocorrências do padrão por uma string de substituição:

```
texto = "Python é uma linguagem de programação. Python é divertido."
padrao = "Python"
substituicao = "Java"
novo_texto = re.sub(padrao, substituicao, texto)
print(novo_texto)  # Java é uma linguagem de programação. Java é divertido.

# Substituindo apenas a primeira ocorrência
novo_texto = re.sub(padrao, substituicao, texto, count=1)
print(novo_texto)  # Java é uma linguagem de programação. Python é divertido.
```

re.split()

A função `split()` divide a string pelo padrão:

```
texto = "Python,Java,C++,JavaScript"
padrao = ","
partes = re.split(padrao, texto)
print(partes)  # ['Python', 'Java', 'C++', 'JavaScript']

# Limitando o número de divisões
partes = re.split(padrao, texto, maxsplit=2)
print(partes)  # ['Python', 'Java', 'C++,JavaScript']

# Usando expressão regular mais complexa
```

```
texto = "Python, Java; C++: JavaScript"
padrao = r"[,;:]"
partes = re.split(padrao, texto)
print(partes)  # ['Python', ' Java', ' C++', ' JavaScript']
```

re.compile()

A função `compile()` compila um padrão para reutilização, o que pode melhorar o desempenho quando o mesmo padrão é usado várias vezes:

```
padrao = re.compile(r"\b\w+\b")
texto1 = "Python é uma linguagem de programação."
texto2 = "Java é outra linguagem popular."

palavras1 = padrao.findall(texto1)
palavras2 = padrao.findall(texto2)

print(palavras1)  # ['Python', 'é', 'uma', 'linguagem', 'de', 'programação']
print(palavras2)  # ['Java', 'é', 'outra', 'linguagem', 'popular']
```

Grupos e Captura

Os parênteses `()` em expressões regulares criam grupos de captura, que permitem extrair partes específicas do texto correspondente:

```
texto = "O e-mail do João é joao@exemplo.com e o da Maria é maria@exemplo.com"
padrao = r"(\w+)@(\w+)\.(\w+)"
matches = re.finditer(padrao, texto)

for match in matches:
    print(f"E-mail completo: {match.group(0)}")
    print(f"Nome de usuário: {match.group(1)}")
    print(f"Domínio: {match.group(2)}")
    print(f"TLD: {match.group(3)}")
    print()

# E-mail completo: joao@exemplo.com
# Nome de usuário: joao
# Domínio: exemplo
# TLD: com
#
# E-mail completo: maria@exemplo.com
# Nome de usuário: maria
# Domínio: exemplo
# TLD: com
```

Grupos Nomeados

Você pode nomear grupos de captura usando a sintaxe `(?P<nome>...)`:

```
texto = "O e-mail do João é joao@exemplo.com"
padrao = r"(?P<usuario>\w+)@(?P<dominio>\w+)\.(?P<tld>\w+)"
match = re.search(padrao, texto)

if match:
```

```
    print(f"E-mail completo: {match.group(0)}")
    print(f"Nome de usuário: {match.group('usuario')}")
    print(f"Domínio: {match.group('dominio')}")
    print(f"TLD: {match.group('tld')}")

# E-mail completo: joao@exemplo.com
# Nome de usuário: joao
# Domínio: exemplo
# TLD: com
```

Grupos Não-Capturantes

Se você precisa de um grupo para alternação ou repetição, mas não quer capturar seu conteúdo, use `(?:...)`:

```
texto = "abc123def456"
padrao = r"(?:\d+)"
numeros = re.findall(padrao, texto)
print(numeros)  # ['123', '456']
```

Lookahead e Lookbehind

Lookahead e lookbehind são asserções que verificam se um padrão é seguido ou precedido por outro, sem incluir o segundo padrão no match:

Lookahead Positivo

`(?=...)` verifica se o que vem a seguir corresponde ao padrão, sem incluí-lo no match:

```
# Encontrar palavras seguidas por um número
texto = "Python3 Java8 C++11"
padrao = r"\w+(?=\d+)"
matches = re.findall(padrao, texto)
print(matches)  # ['Python', 'Java', 'C++']
```

Lookahead Negativo

`(?!...)` verifica se o que vem a seguir NÃO corresponde ao padrão:

```
# Encontrar palavras que não são seguidas por um número
texto = "Python3 Java8 C++ JavaScript"
padrao = r"\w+(?!\d)"
matches = re.findall(padrao, texto)
print(matches)  # ['Python', 'Java', 'C', 'JavaScript']
```

Lookbehind Positivo

`(?<=...)` verifica se o que vem antes corresponde ao padrão, sem incluí-lo no match:

```
# Encontrar números precedidos por uma letra
texto = "A1 B2 C3 123"
padrao = r"(?<=[A-Z])\d"
matches = re.findall(padrao, texto)
print(matches)  # ['1', '2', '3']
```

Lookbehind Negativo

`(?<!...)` verifica se o que vem antes NÃO corresponde ao padrão:

```
# Encontrar números que não são precedidos por uma letra
texto = "A1 B2 C3 123"
padrao = r"(?<![A-Z])\d"
matches = re.findall(padrao, texto)
print(matches)  # ['2', '3']
```

Flags

O módulo `re` suporta várias flags que modificam o comportamento das expressões regulares:

- `re.IGNORECASE` ou `re.I`: Ignora maiúsculas/minúsculas
- `re.MULTILINE` ou `re.M`: Faz `^` e `$` corresponderem ao início/fim de cada linha
- `re.DOTALL` ou `re.S`: Faz `.` corresponder a qualquer caractere, incluindo nova linha
- `re.VERBOSE` ou `re.X`: Permite adicionar comentários e espaços em branco no padrão

Exemplos:

```
# re.IGNORECASE
texto = "Python é uma linguagem de programação. PYTHON é divertido."
padrao = "python"
ocorrencias = re.findall(padrao, texto, re.IGNORECASE)
print(ocorrencias)  # ['Python', 'PYTHON']

# re.MULTILINE
texto = "Linha 1\nLinha 2\nLinha 3"
padrao = "^Linha"
ocorrencias = re.findall(padrao, texto, re.MULTILINE)
print(ocorrencias)  # ['Linha', 'Linha', 'Linha']

# re.DOTALL
texto = "Linha 1\nLinha 2"
padrao = "Linha.*Linha"
ocorrencias = re.findall(padrao, texto, re.DOTALL)
print(ocorrencias)  # ['Linha 1\nLinha']

# re.VERBOSE
padrao = re.compile(r"""
    \b          # Borda de palavra
    [A-Z]       # Primeira letra maiúscula
    \w*         # Resto da palavra
    \b          # Borda de palavra
""", re.VERBOSE)
texto = "Python Java C++ JavaScript"
ocorrencias = padrao.findall(texto)
print(ocorrencias)  # ['Python', 'Java', 'JavaScript']
```

Exemplos Práticos

Validação de E-mail

```
import re
```

```python
def validar_email(email):
    padrao = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    return bool(re.match(padrao, email))

emails = ["usuario@exemplo.com", "usuario.nome@empresa-exemplo.com.br", "invalido@", "sem_arroba.com"]

for email in emails:
    if validar_email(email):
        print(f"{email} é válido")
    else:
        print(f"{email} é inválido")

# usuario@exemplo.com é válido
# usuario.nome@empresa-exemplo.com.br é válido
# invalido@ é inválido
# sem_arroba.com é inválido
```

Extração de URLs

```python
import re

texto = """
Visite nosso site em https://www.exemplo.com.br para mais informações.
Também estamos em http://blog.exemplo.com e https://loja.exemplo.net/produtos.
"""

padrao = r'https?://(?:www\.)?(?:[a-zA-Z0-9-]+\.)+[a-zA-Z]{2,}(?:/[a-zA-Z0-9-._~:/?#[\]@!$&\'()*+,;=]*)
urls = re.findall(padrao, texto)

for url in urls:
    print(url)

# https://www.exemplo.com.br
# http://blog.exemplo.com
# https://loja.exemplo.net/produtos
```

Análise de Logs

```python
import re

log = """
192.168.1.1 - - [20/May/2023:10:12:45 +0000] "GET /index.html HTTP/1.1" 200 1234
192.168.1.2 - - [20/May/2023:10:13:50 +0000] "POST /login HTTP/1.1" 302 0
192.168.1.1 - - [20/May/2023:10:14:12 +0000] "GET /dashboard HTTP/1.1" 200 5678
192.168.1.3 - - [20/May/2023:10:15:30 +0000] "GET /nonexistent HTTP/1.1" 404 123
"""

padrao = r'(\d+\.\d+\.\d+\.\d+) .+ \[([^\]]+)\] "(\w+) ([^"]+)" (\d+) (\d+)'
matches = re.finditer(padrao, log)

for match in matches:
    ip = match.group(1)
    data = match.group(2)
```

```
    metodo = match.group(3)
    url = match.group(4)
    status = match.group(5)
    tamanho = match.group(6)

    print(f"IP: {ip}, Data: {data}, Método: {metodo}, URL: {url}, Status: {status}, Tamanho: {tamanho}")

# IP: 192.168.1.1, Data: 20/May/2023:10:12:45 +0000, Método: GET, URL: /index.html HTTP/1.1, Status: 20
# IP: 192.168.1.2, Data: 20/May/2023:10:13:50 +0000, Método: POST, URL: /login HTTP/1.1, Status: 302, T
# IP: 192.168.1.1, Data: 20/May/2023:10:14:12 +0000, Método: GET, URL: /dashboard HTTP/1.1, Status: 200
# IP: 192.168.1.3, Data: 20/May/2023:10:15:30 +0000, Método: GET, URL: /nonexistent HTTP/1.1, Status: 4
```

Substituição com Função

A função `re.sub()` pode receber uma função como parâmetro de substituição:

```
import re

def converter_para_celsius(match):
    fahrenheit = float(match.group(1))
    celsius = (fahrenheit - 32) * 5/9
    return f"{celsius:.1f}°C"

texto = "A temperatura hoje é de 68°F em Nova York e 86°F em Miami."
padrao = r'(\d+(?:\.\d+)?)°F'
resultado = re.sub(padrao, converter_para_celsius, texto)

print(resultado)
# A temperatura hoje é de 20.0°C em Nova York e 30.0°C em Miami.
```

Boas Práticas

1. **Use Strings Brutas**: Sempre use strings brutas (`r"..."`) para padrões de expressão regular para evitar problemas com sequências de escape.
2. **Compile Padrões Reutilizados**: Use `re.compile()` para padrões que serão reutilizados várias vezes.
3. **Seja Específico**: Torne seus padrões o mais específicos possível para evitar correspondências indesejadas.
4. **Teste suas Expressões Regulares**: Use ferramentas online como regex101.com para testar e depurar suas expressões regulares.
5. **Documente Expressões Complexas**: Para expressões regulares complexas, use a flag `re.VERBOSE` e adicione comentários.
6. **Considere Alternativas**: Para tarefas simples, funções de string como `split()`, `startswith()`, `endswith()` podem ser mais legíveis e eficientes.

Conclusão

As expressões regulares são uma ferramenta poderosa para manipulação de texto em Python. Embora possam parecer intimidadoras no início, dominar os conceitos básicos permite resolver problemas complexos de processamento de texto de forma elegante e eficiente.

No próximo tópico, exploraremos a programação funcional em Python, um paradigma que enfatiza a aplicação de funções e evita mudanças de estado e dados mutáveis.

Módulo 3: Python Avançado

Programação Funcional

A programação funcional é um paradigma de programação que trata a computação como a avaliação de funções matemáticas e evita estados mutáveis e dados variáveis. Em Python, embora seja uma linguagem multiparadigma, podemos aplicar muitos conceitos da programação funcional para escrever código mais conciso, legível e menos propenso a erros.

Princípios da Programação Funcional

1. Funções como Cidadãos de Primeira Classe

Em Python, as funções são objetos de primeira classe, o que significa que elas podem ser: - Atribuídas a variáveis - Passadas como argumentos para outras funções - Retornadas por outras funções - Armazenadas em estruturas de dados

```
# Atribuindo uma função a uma variável
def saudacao(nome):
    return f"Olá, {nome}!"

dizer_ola = saudacao
print(dizer_ola("Maria"))  # Olá, Maria!

# Passando uma função como argumento
def aplicar_funcao(f, valor):
    return f(valor)

def dobrar(x):
    return x * 2

print(aplicar_funcao(dobrar, 5))  # 10

# Retornando uma função
def criar_multiplicador(fator):
    def multiplicador(x):
        return x * fator
    return multiplicador

duplicar = criar_multiplicador(2)
triplicar = criar_multiplicador(3)

print(duplicar(5))  # 10
print(triplicar(5))  # 15
```

```python
# Armazenando funções em estruturas de dados
operacoes = {
    'soma': lambda x, y: x + y,
    'subtracao': lambda x, y: x - y,
    'multiplicacao': lambda x, y: x * y,
    'divisao': lambda x, y: x / y if y != 0 else "Erro: divisão por zero"
}

print(operacoes['soma'](5, 3))  # 8
print(operacoes['divisao'](10, 2))  # 5.0
```

2. Funções Puras

Uma função pura é uma função que: - Dado o mesmo input, sempre retorna o mesmo output - Não tem efeitos colaterais (não modifica variáveis externas, não realiza I/O, etc.)

```python
# Função pura
def soma_pura(a, b):
    return a + b

# Função impura (tem efeito colateral)
total = 0
def soma_impura(a, b):
    global total
    total += a + b
    return total
```

As funções puras são mais fáceis de testar, depurar e entender, pois seu comportamento depende apenas de seus inputs.

3. Imutabilidade

Na programação funcional, os dados são imutáveis, ou seja, uma vez criados, não podem ser alterados. Em Python, podemos usar tipos imutáveis como tuplas, strings e frozensets:

```python
# Usando tuplas (imutáveis) em vez de listas (mutáveis)
ponto = (3, 4)
# ponto[0] = 5  # Isso geraria um TypeError

# Criando uma nova tupla em vez de modificar a existente
novo_ponto = (5, ponto[1])
print(novo_ponto)  # (5, 4)

# Usando frozenset (conjunto imutável)
conjunto_imutavel = frozenset([1, 2, 3])
# conjunto_imutavel.add(4)  # Isso geraria um AttributeError
```

4. Recursão em vez de Iteração

Na programação funcional pura, loops são substituídos por recursão:

```python
# Cálculo de fatorial usando recursão
def fatorial(n):
    if n <= 1:
        return 1
    return n * fatorial(n - 1)
```

```
print(fatorial(5))  # 120

# Fibonacci usando recursão
def fibonacci(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(7))  # 13
```

No entanto, Python não é otimizado para recursão profunda (tem um limite de recursão), então para casos complexos, podemos usar técnicas como memoização ou até mesmo iteração quando necessário.

Funções de Ordem Superior

Funções de ordem superior são funções que recebem outras funções como argumentos ou retornam funções como resultado.

map()

A função `map()` aplica uma função a cada item de um iterável e retorna um iterador com os resultados:

```
# Dobrar cada número em uma lista
numeros = [1, 2, 3, 4, 5]
dobrados = map(lambda x: x * 2, numeros)
print(list(dobrados))  # [2, 4, 6, 8, 10]

# Converter strings para maiúsculas
nomes = ["ana", "bruno", "carla"]
maiusculas = map(str.upper, nomes)
print(list(maiusculas))  # ['ANA', 'BRUNO', 'CARLA']

# Map com múltiplos iteráveis
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
soma = map(lambda x, y: x + y, lista1, lista2)
print(list(soma))  # [5, 7, 9]
```

filter()

A função `filter()` filtra elementos de um iterável com base em uma função que retorna um booleano:

```
# Filtrar números pares
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
pares = filter(lambda x: x % 2 == 0, numeros)
print(list(pares))  # [2, 4, 6, 8, 10]

# Filtrar strings não vazias
strings = ["", "a", "", "b", "c", ""]
nao_vazias = filter(bool, strings)  # bool("") é False, bool("a") é True
print(list(nao_vazias))  # ['a', 'b', 'c']

# Filtrar números primos
```

```python
def eh_primo(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

numeros = range(1, 20)
primos = filter(eh_primo, numeros)
print(list(primos))  # [2, 3, 5, 7, 11, 13, 17, 19]
```

reduce()

A função `reduce()` aplica uma função cumulativamente aos itens de um iterável, reduzindo-os a um único valor:

```python
from functools import reduce

# Somar todos os números
numeros = [1, 2, 3, 4, 5]
soma = reduce(lambda x, y: x + y, numeros)
print(soma)  # 15

# Encontrar o maior número
maior = reduce(lambda x, y: x if x > y else y, numeros)
print(maior)  # 5

# Concatenar strings
palavras = ["Python", "é", "incrível"]
frase = reduce(lambda x, y: x + " " + y, palavras)
print(frase)  # Python é incrível

# Reduce com valor inicial
produto = reduce(lambda x, y: x * y, numeros, 2)  # Começa com 2 e multiplica por cada número
print(produto)  # 240 (2 * 1 * 2 * 3 * 4 * 5)
```

Funções Lambda

As funções lambda são funções anônimas de uma única expressão:

```python
# Função lambda básica
dobrar = lambda x: x * 2
print(dobrar(5))  # 10

# Lambda com múltiplos argumentos
soma = lambda x, y: x + y
print(soma(3, 4))  # 7
```

```python
# Lambda com argumentos padrão
saudacao = lambda nome, mensagem="Olá": f"{mensagem}, {nome}!"
print(saudacao("Maria"))  # Olá, Maria!
print(saudacao("João", "Bem-vindo"))  # Bem-vindo, João!

# Lambda em expressões
pessoas = [
    {"nome": "Ana", "idade": 25},
    {"nome": "Bruno", "idade": 30},
    {"nome": "Carla", "idade": 20}
]

# Ordenar por idade
pessoas_ordenadas = sorted(pessoas, key=lambda p: p["idade"])
print([p["nome"] for p in pessoas_ordenadas])  # ['Carla', 'Ana', 'Bruno']

# Filtrar pessoas com mais de 25 anos
adultos = filter(lambda p: p["idade"] > 25, pessoas)
print([p["nome"] for p in adultos])  # ['Bruno']
```

Compreensões

As compreensões são uma forma concisa de criar listas, dicionários e conjuntos:

Compreensão de Lista

```python
# Quadrados dos números de 1 a 10
quadrados = [x**2 for x in range(1, 11)]
print(quadrados)  # [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

# Compreensão com condição
pares = [x for x in range(1, 11) if x % 2 == 0]
print(pares)  # [2, 4, 6, 8, 10]

# Compreensão com múltiplas condições
numeros = [x for x in range(1, 101) if x % 3 == 0 if x % 5 == 0]
print(numeros)  # [15, 30, 45, 60, 75, 90]

# Compreensão com if-else
classificacao = ["par" if x % 2 == 0 else "ímpar" for x in range(1, 6)]
print(classificacao)  # ['ímpar', 'par', 'ímpar', 'par', 'ímpar']

# Compreensão aninhada
matriz = [[i * j for j in range(1, 4)] for i in range(1, 4)]
print(matriz)  # [[1, 2, 3], [2, 4, 6], [3, 6, 9]]
```

Compreensão de Dicionário

```python
# Mapeando números para seus quadrados
quadrados = {x: x**2 for x in range(1, 6)}
print(quadrados)  # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

```python
# Filtrando itens
numeros = {x: x**2 for x in range(1, 11) if x % 2 == 0}
print(numeros)  # {2: 4, 4: 16, 6: 36, 8: 64, 10: 100}

# Invertendo um dicionário
original = {"a": 1, "b": 2, "c": 3}
invertido = {v: k for k, v in original.items()}
print(invertido)  # {1: 'a', 2: 'b', 3: 'c'}
```

Compreensão de Conjunto

```python
# Conjunto de quadrados
quadrados = {x**2 for x in range(1, 6)}
print(quadrados)  # {1, 4, 9, 16, 25}

# Conjunto de letras únicas
letras = {c.lower() for c in "Python Programming"}
print(letras)  # {'p', 'y', 't', 'h', 'o', 'n', 'g', 'r', 'a', 'm', 'i'}
```

Funções Parciais

A função `partial` do módulo `functools` permite criar novas funções com alguns argumentos fixos:

```python
from functools import partial

def potencia(base, expoente):
    return base ** expoente

# Criando uma função para calcular o quadrado
quadrado = partial(potencia, expoente=2)
print(quadrado(5))  # 25

# Criando uma função para calcular o cubo
cubo = partial(potencia, expoente=3)
print(cubo(5))  # 125

# Função parcial com múltiplos argumentos fixos
def saudacao(cumprimento, nome, pontuacao):
    return f"{cumprimento}, {nome}{pontuacao}"

dizer_ola = partial(saudacao, "Olá", pontuacao="!")
print(dizer_ola("Maria"))  # Olá, Maria!

dizer_adeus = partial(saudacao, "Adeus", pontuacao="...")
print(dizer_adeus("João"))  # Adeus, João...
```

Closures

Uma closure é uma função que "lembra" o ambiente em que foi criada, mesmo quando é executada fora desse ambiente:

```python
def criar_contador():
    contador = 0
```

```
    def incrementar():
        nonlocal contador
        contador += 1
        return contador

    return incrementar

contador1 = criar_contador()
contador2 = criar_contador()

print(contador1())  # 1
print(contador1())  # 2
print(contador2())  # 1 (contador2 é independente de contador1)
print(contador1())  # 3
```

Decoradores

Os decoradores são uma aplicação de funções de ordem superior e closures. Eles permitem modificar o comportamento de funções ou métodos:

```
def registrar(funcao):
    def wrapper(*args, **kwargs):
        print(f"Chamando {funcao.__name__} com args={args}, kwargs={kwargs}")
        resultado = funcao(*args, **kwargs)
        print(f"{funcao.__name__} retornou {resultado}")
        return resultado
    return wrapper

@registrar
def soma(a, b):
    return a + b

soma(3, 5)
# Saída:
# Chamando soma com args=(3, 5), kwargs={}
# soma retornou 8
```

Para mais detalhes sobre decoradores, consulte o tópico específico sobre eles neste módulo.

Módulo functools

O módulo `functools` fornece ferramentas para trabalhar com funções de ordem superior:

lru_cache

`lru_cache` é um decorador que implementa memoização, armazenando os resultados de chamadas de função para evitar recálculos:

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

```
# Sem lru_cache, isso seria muito lento
print(fibonacci(35))  # 9227465
```

singledispatch

`singledispatch` permite criar funções genéricas que se comportam diferentemente dependendo do tipo do primeiro argumento:

```
from functools import singledispatch

@singledispatch
def formatar(obj):
    return str(obj)

@formatar.register
def _(obj: int):
    return f"Inteiro: {obj}"

@formatar.register
def _(obj: list):
    return f"Lista com {len(obj)} itens: {', '.join(map(str, obj))}"

@formatar.register
def _(obj: dict):
    return f"Dicionário com {len(obj)} chaves: {', '.join(obj.keys())}"

print(formatar("Olá"))  # Olá
print(formatar(42))     # Inteiro: 42
print(formatar([1, 2, 3]))  # Lista com 3 itens: 1, 2, 3
print(formatar({"a": 1, "b": 2}))  # Dicionário com 2 chaves: a, b
```

Módulo operator

O módulo `operator` fornece funções que correspondem aos operadores do Python, úteis para usar com funções como `map()`, `filter()` e `reduce()`:

```
import operator
from functools import reduce

# Operadores aritméticos
numeros = [1, 2, 3, 4, 5]
soma = reduce(operator.add, numeros)
print(soma)  # 15

produto = reduce(operator.mul, numeros)
print(produto)  # 120

# Operadores de comparação
maiores_que_3 = list(filter(lambda x: operator.gt(x, 3), numeros))
print(maiores_que_3)  # [4, 5]

# Operadores de item/atributo
pessoas = [
    {"nome": "Ana", "idade": 25},
```

```
    {"nome": "Bruno", "idade": 30},
    {"nome": "Carla", "idade": 20}
]

# Extrair idades
idades = list(map(operator.itemgetter("idade"), pessoas))
print(idades)  # [25, 30, 20]

# Ordenar por nome
pessoas_ordenadas = sorted(pessoas, key=operator.itemgetter("nome"))
print([p["nome"] for p in pessoas_ordenadas])  # ['Ana', 'Bruno', 'Carla']

# Operadores lógicos
verdadeiro = operator.and_(True, True)
print(verdadeiro)  # True

falso = operator.or_(False, False)
print(falso)  # False
```

Módulo itertools

O módulo `itertools` fornece ferramentas para criar e trabalhar com iteradores:

```
import itertools

# count: contador infinito
contador = itertools.count(10, 2)  # Começa em 10, incrementa de 2 em 2
for i in itertools.islice(contador, 5):
    print(i, end=" ")  # 10 12 14 16 18
print()

# cycle: cicla infinitamente por um iterável
ciclo = itertools.cycle(["A", "B", "C"])
for i, c in zip(range(7), ciclo):
    print(c, end=" ")  # A B C A B C A
print()

# chain: concatena iteráveis
for i in itertools.chain([1, 2], [3, 4], [5, 6]):
    print(i, end=" ")  # 1 2 3 4 5 6
print()

# combinations: todas as combinações de r elementos
for combo in itertools.combinations("ABC", 2):
    print("".join(combo), end=" ")  # AB AC BC
print()

# permutations: todas as permutações de r elementos
for perm in itertools.permutations("ABC", 2):
    print("".join(perm), end=" ")  # AB AC BA BC CA CB
print()

# product: produto cartesiano
for prod in itertools.product("AB", "12"):
```

```
    print("".join(prod), end=" ")  # A1 A2 B1 B2
print()

# groupby: agrupa elementos consecutivos por uma chave
animais = ["cachorro", "gato", "camelo", "cobra", "girafa"]
for chave, grupo in itertools.groupby(sorted(animais), key=lambda x: x[0]):
    print(f"{chave}: {list(grupo)}")
# c: ['cachorro', 'camelo', 'cobra']
# g: ['gato', 'girafa']
```

Programação Funcional vs. Imperativa

Vamos comparar abordagens funcionais e imperativas para resolver o mesmo problema:

Problema: Calcular a soma dos quadrados dos números pares de 1 a 10.

Abordagem Imperativa:

```
# Abordagem imperativa
def soma_quadrados_pares_imperativa():
    resultado = 0
    for i in range(1, 11):
        if i % 2 == 0:
            resultado += i ** 2
    return resultado

print(soma_quadrados_pares_imperativa())  # 220
```

Abordagem Funcional:

```
# Abordagem funcional com funções de ordem superior
def soma_quadrados_pares_funcional():
    return sum(map(lambda x: x**2, filter(lambda x: x % 2 == 0, range(1, 11))))

print(soma_quadrados_pares_funcional())  # 220

# Abordagem funcional com compreensão de lista
def soma_quadrados_pares_compreensao():
    return sum(x**2 for x in range(1, 11) if x % 2 == 0)

print(soma_quadrados_pares_compreensao())  # 220
```

A abordagem funcional tende a ser mais concisa e expressiva, focando no "o quê" em vez do "como".

Vantagens da Programação Funcional

1. **Código mais conciso e expressivo**: As operações funcionais como `map`, `filter` e compreensões permitem expressar operações complexas de forma concisa.
2. **Menos bugs**: A imutabilidade e a ausência de efeitos colaterais reduzem a chance de bugs relacionados a estados compartilhados.
3. **Mais fácil de testar**: Funções puras são mais fáceis de testar, pois seu comportamento depende apenas de seus inputs.
4. **Paralelização mais fácil**: Código sem efeitos colaterais é mais fácil de paralelizar, pois não há preocupação com estados compartilhados.

5. **Composição de funções**: Funções podem ser facilmente combinadas para criar operações mais complexas.

Desvantagens e Limitações

1. **Curva de aprendizado**: A programação funcional pode ser menos intuitiva para programadores acostumados com o paradigma imperativo.
2. **Desempenho**: Em alguns casos, abordagens funcionais podem ser menos eficientes em termos de desempenho e uso de memória.
3. **Recursão limitada**: Python tem um limite de recursão, o que pode ser um problema para algoritmos recursivos complexos.
4. **Legibilidade**: Código funcional muito denso pode ser difícil de ler para programadores não familiarizados com o paradigma.

Boas Práticas

1. **Equilíbrio**: Use programação funcional onde ela traz benefícios claros, mas não force o paradigma onde abordagens imperativas são mais simples.
2. **Funções Puras**: Prefira funções puras sempre que possível, isolando código com efeitos colaterais.
3. **Nomes Descritivos**: Use nomes descritivos para funções e variáveis, especialmente ao usar funções lambda e composição de funções.
4. **Documentação**: Documente seu código funcional, especialmente para padrões menos comuns.
5. **Evite Excesso de Aninhamento**: Funções aninhadas demais podem tornar o código difícil de entender.

Conclusão

A programação funcional em Python oferece ferramentas poderosas para escrever código mais conciso, expressivo e menos propenso a erros. Embora Python não seja uma linguagem puramente funcional, ele suporta muitos conceitos funcionais que podem ser combinados com outros paradigmas para criar código elegante e eficiente.

No próximo tópico, exploraremos concorrência e paralelismo em Python, que permitem executar múltiplas tarefas simultaneamente para melhorar o desempenho e a responsividade de aplicações.

Módulo 3: Python Avançado

Concorrência e Paralelismo

A concorrência e o paralelismo são conceitos fundamentais na programação moderna, especialmente quando se trata de melhorar o desempenho de aplicações que lidam com operações intensivas em CPU ou I/O. Python oferece várias ferramentas e bibliotecas para implementar concorrência e paralelismo, cada uma com suas próprias vantagens e casos de uso.

Concorrência vs. Paralelismo

Antes de mergulharmos nas implementações, é importante entender a diferença entre concorrência e paralelismo:

- **Concorrência**: Refere-se à capacidade de um programa lidar com múltiplas tarefas ao mesmo tempo, alternando entre elas. As tarefas podem começar, executar e completar em períodos sobrepostos, mas não necessariamente executam ao mesmo tempo.
- **Paralelismo**: Refere-se à execução simultânea de múltiplas tarefas, geralmente em múltiplos processadores ou núcleos. O paralelismo implica em concorrência, mas a concorrência não implica necessariamente em paralelismo.

Uma analogia comum é pensar em um chef de cozinha: - Um chef **concorrente** pode estar preparando vários pratos ao mesmo tempo, alternando entre eles (colocar algo para assar, cortar legumes enquanto espera, etc.). - Vários chefs **paralelos** podem estar trabalhando simultaneamente em diferentes pratos na mesma cozinha.

O Global Interpreter Lock (GIL)

Uma consideração importante ao trabalhar com concorrência e paralelismo em Python é o Global Interpreter Lock (GIL). O GIL é um mecanismo que permite que apenas uma thread execute código Python em um processo por vez, o que pode limitar o paralelismo real em código CPU-bound.

No entanto, o GIL não impede totalmente o paralelismo: - Para operações I/O-bound, o GIL é liberado durante a espera por I/O, permitindo que outras threads executem. - Para operações CPU-bound, podemos usar processos múltiplos (que têm GILs separados) ou extensões C que liberam o GIL.

Threads em Python

As threads são a forma mais básica de concorrência em Python, implementadas através do módulo `threading`.

Criando e Executando Threads

```
import threading
import time

def tarefa(nome, tempo_espera):
```

```
    print(f"Iniciando a tarefa {nome}")
    time.sleep(tempo_espera)  # Simula uma operação que leva tempo
    print(f"Concluindo a tarefa {nome}")

# Criando threads
thread1 = threading.Thread(target=tarefa, args=("A", 2))
thread2 = threading.Thread(target=tarefa, args=("B", 1))

# Iniciando threads
inicio = time.time()
thread1.start()
thread2.start()

# Esperando as threads terminarem
thread1.join()
thread2.join()
fim = time.time()

print(f"Tempo total: {fim - inicio:.2f} segundos")
```

Neste exemplo, as tarefas A e B são executadas concorrentemente. Embora a tarefa A leve 2 segundos e a tarefa B leve 1 segundo, o tempo total é de aproximadamente 2 segundos, não 3, porque elas executam concorrentemente.

Sincronização com Locks

Quando múltiplas threads acessam e modificam os mesmos dados, podem ocorrer condições de corrida. Locks (ou mutexes) são usados para sincronizar o acesso a recursos compartilhados:

```
import threading
import time

contador = 0
lock = threading.Lock()

def incrementar(n):
    global contador
    for _ in range(n):
        with lock:  # Adquire o lock antes de modificar o contador
            contador += 1

# Criando threads
thread1 = threading.Thread(target=incrementar, args=(100000,))
thread2 = threading.Thread(target=incrementar, args=(100000,))

# Iniciando threads
thread1.start()
thread2.start()

# Esperando as threads terminarem
thread1.join()
thread2.join()

print(f"Contador final: {contador}")  # Deve ser 200000
```

Sem o lock, o valor final do contador seria imprevisível devido a condições de corrida.

Outros Mecanismos de Sincronização

Python oferece outros mecanismos de sincronização além de locks:

- **RLock**: Um lock reentrant que pode ser adquirido várias vezes pela mesma thread.
- **Semaphore**: Controla o acesso a um recurso compartilhado por múltiplas threads.
- **Event**: Permite que uma thread sinalize a ocorrência de um evento para outras threads.
- **Condition**: Combina a funcionalidade de um lock e um evento.
- **Barrier**: Bloqueia até que um número específico de threads tenha alcançado o barrier.

Exemplo com Event:

```
import threading
import time

evento = threading.Event()

def esperar_evento():
    print("Thread esperando pelo evento...")
    evento.wait()  # Bloqueia até que o evento seja definido
    print("Thread continuando após o evento")

# Criando e iniciando a thread
thread = threading.Thread(target=esperar_evento)
thread.start()

# Simulando algum processamento
time.sleep(2)
print("Definindo o evento")
evento.set()  # Sinaliza o evento, desbloqueando a thread

# Esperando a thread terminar
thread.join()
```

Thread Pool com concurrent.futures

O módulo `concurrent.futures` fornece uma interface de alto nível para trabalhar com threads e processos:

```
import concurrent.futures
import time

def tarefa(nome, tempo_espera):
    print(f"Iniciando a tarefa {nome}")
    time.sleep(tempo_espera)  # Simula uma operação que leva tempo
    return f"Resultado da tarefa {nome}"

# Usando ThreadPoolExecutor
with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
    # Submete tarefas para execução
    futuro1 = executor.submit(tarefa, "A", 2)
    futuro2 = executor.submit(tarefa, "B", 1)
    futuro3 = executor.submit(tarefa, "C", 3)

    # Obtém os resultados à medida que ficam disponíveis
```

```
    for futuro in concurrent.futures.as_completed([futuro1, futuro2, futuro3]):
        try:
            resultado = futuro.result()
            print(resultado)
        except Exception as e:
            print(f"Tarefa gerou uma exceção: {e}")
```

Também podemos usar `map` para aplicar uma função a múltiplos argumentos em paralelo:

```
import concurrent.futures
import time

def tarefa(params):
    nome, tempo_espera = params
    print(f"Iniciando a tarefa {nome}")
    time.sleep(tempo_espera)  # Simula uma operação que leva tempo
    return f"Resultado da tarefa {nome}"

# Lista de parâmetros para as tarefas
params = [("A", 2), ("B", 1), ("C", 3), ("D", 1.5)]

# Usando ThreadPoolExecutor com map
with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
    resultados = list(executor.map(tarefa, params))

    for resultado in resultados:
        print(resultado)
```

Multiprocessamento em Python

Para operações CPU-bound, o multiprocessamento pode oferecer melhor desempenho do que threading devido ao GIL. O módulo `multiprocessing` fornece uma API semelhante a `threading`, mas usa processos em vez de threads.

Criando e Executando Processos

```
import multiprocessing
import time

def tarefa_cpu_intensiva(n):
    """Função que realiza uma operação intensiva em CPU."""
    resultado = 0
    for i in range(n):
        resultado += i * i
    return resultado

if __name__ == "__main__":
    # Criando processos
    processo1 = multiprocessing.Process(target=tarefa_cpu_intensiva, args=(10000000,))
    processo2 = multiprocessing.Process(target=tarefa_cpu_intensiva, args=(20000000,))

    # Iniciando processos
    inicio = time.time()
    processo1.start()
```

```
    processo2.start()

    # Esperando os processos terminarem
    processo1.join()
    processo2.join()
    fim = time.time()

    print(f"Tempo total com multiprocessamento: {fim - inicio:.2f} segundos")

    # Comparação com execução sequencial
    inicio = time.time()
    tarefa_cpu_intensiva(10000000)
    tarefa_cpu_intensiva(20000000)
    fim = time.time()

    print(f"Tempo total sequencial: {fim - inicio:.2f} segundos")
```

Compartilhando Dados entre Processos

Ao contrário das threads, os processos não compartilham memória por padrão. O módulo `multiprocessing` fornece várias maneiras de compartilhar dados entre processos:

- **Queue**: Uma fila thread-safe e process-safe para comunicação entre processos.
- **Pipe**: Um canal bidirecional para comunicação entre dois processos.
- **Value e Array**: Objetos compartilhados usando memória compartilhada.
- **Manager**: Um servidor de objetos que permite compartilhar objetos Python mais complexos.

Exemplo com Queue:

```
import multiprocessing
import time
import random

def produtor(fila):
    """Produz itens e os coloca na fila."""
    for i in range(5):
        item = random.randint(1, 100)
        fila.put(item)
        print(f"Produtor colocou {item} na fila")
        time.sleep(random.random())

def consumidor(fila):
    """Consome itens da fila."""
    while True:
        try:
            item = fila.get(timeout=3)  # Espera até 3 segundos por um item
            print(f"Consumidor obteve {item} da fila")
            time.sleep(random.random() * 2)
        except multiprocessing.queues.Empty:
            print("Fila vazia, consumidor saindo")
            break

if __name__ == "__main__":
    # Criando uma fila compartilhada
    fila = multiprocessing.Queue()
```

```
    # Criando processos
    proc_produtor = multiprocessing.Process(target=produtor, args=(fila,))
    proc_consumidor = multiprocessing.Process(target=consumidor, args=(fila,))

    # Iniciando processos
    proc_produtor.start()
    proc_consumidor.start()

    # Esperando os processos terminarem
    proc_produtor.join()
    proc_consumidor.join()
```

Exemplo com Value e Array:

```
import multiprocessing
import time

def incrementar_valor(valor_compartilhado, array_compartilhado, lock):
    """Incrementa um valor compartilhado e modifica um array compartilhado."""
    with lock:
        valor_compartilhado.value += 1
        for i in range(len(array_compartilhado)):
            array_compartilhado[i] = array_compartilhado[i] * 2

    print(f"Processo {multiprocessing.current_process().name}:")
    print(f"  Valor: {valor_compartilhado.value}")
    print(f"  Array: {list(array_compartilhado)}")

if __name__ == "__main__":
    # Criando objetos compartilhados
    valor = multiprocessing.Value('i', 0)  # 'i' para inteiro
    array = multiprocessing.Array('i', [1, 2, 3, 4, 5])  # 'i' para inteiro
    lock = multiprocessing.Lock()

    # Criando processos
    processos = [
        multiprocessing.Process(target=incrementar_valor, args=(valor, array, lock))
        for _ in range(3)
    ]

    # Iniciando processos
    for p in processos:
        p.start()

    # Esperando os processos terminarem
    for p in processos:
        p.join()

    print("\nValores finais:")
    print(f"Valor: {valor.value}")
    print(f"Array: {list(array)}")
```

Process Pool com concurrent.futures

Assim como com threads, podemos usar `concurrent.futures` para trabalhar com pools de processos:

```
import concurrent.futures
import time
import math

def calcular_fatorial(n):
    """Calcula o fatorial de n."""
    if n == 0:
        return 1
    return math.factorial(n)

if __name__ == "__main__":
    numeros = list(range(1, 21))  # Calcular fatorial de 1 a 20

    # Usando ProcessPoolExecutor
    inicio = time.time()
    with concurrent.futures.ProcessPoolExecutor(max_workers=4) as executor:
        resultados = list(executor.map(calcular_fatorial, numeros))
    fim = time.time()

    print(f"Tempo com ProcessPoolExecutor: {fim - inicio:.4f} segundos")

    # Comparação com execução sequencial
    inicio = time.time()
    resultados_seq = [calcular_fatorial(n) for n in numeros]
    fim = time.time()

    print(f"Tempo sequencial: {fim - inicio:.4f} segundos")

    # Verificando se os resultados são iguais
    print(f"Resultados iguais: {resultados == resultados_seq}")
```

Asyncio: Programação Assíncrona

O módulo `asyncio` introduzido no Python 3.4 (e significativamente melhorado nas versões subsequentes) fornece uma estrutura para escrever código concorrente usando a sintaxe `async/await`. É particularmente útil para operações I/O-bound.

Conceitos Básicos de Asyncio

- **Coroutines**: Funções que podem ser pausadas e retomadas. Definidas com `async def`.
- **Tasks**: Wrappers em torno de coroutines que são agendadas para execução no event loop.
- **Event Loop**: O núcleo do asyncio, responsável por executar coroutines, lidar com I/O e agendar callbacks.
- **Futures**: Objetos que representam o resultado de uma operação que ainda não foi concluída.

Exemplo Básico de Asyncio

```
import asyncio

async def tarefa_async(nome, tempo_espera):
    """Uma coroutine que simula uma operação assíncrona."""
```

```
    print(f"Iniciando a tarefa {nome}")
    await asyncio.sleep(tempo_espera)  # Pausa a coroutine sem bloquear o event loop
    print(f"Concluindo a tarefa {nome}")
    return f"Resultado da tarefa {nome}"

async def main():
    # Executando coroutines concorrentemente
    resultados = await asyncio.gather(
        tarefa_async("A", 2),
        tarefa_async("B", 1),
        tarefa_async("C", 3)
    )

    print(f"Resultados: {resultados}")

# Executando o event loop
asyncio.run(main())
```

Criando e Executando Tasks

```
import asyncio
import time

async def tarefa_async(nome, tempo_espera):
    print(f"Iniciando a tarefa {nome}")
    await asyncio.sleep(tempo_espera)
    print(f"Concluindo a tarefa {nome}")
    return f"Resultado da tarefa {nome}"

async def main():
    # Criando tasks
    task1 = asyncio.create_task(tarefa_async("A", 2))
    task2 = asyncio.create_task(tarefa_async("B", 1))
    task3 = asyncio.create_task(tarefa_async("C", 3))

    # Esperando todas as tasks completarem
    await asyncio.wait([task1, task2, task3])

    # Obtendo resultados
    print(f"Resultado task1: {task1.result()}")
    print(f"Resultado task2: {task2.result()}")
    print(f"Resultado task3: {task3.result()}")

# Executando o event loop
inicio = time.time()
asyncio.run(main())
fim = time.time()

print(f"Tempo total: {fim - inicio:.2f} segundos")
```

Trabalhando com Timeouts

```
import asyncio

async def operacao_longa():
    print("Iniciando operação longa...")
    await asyncio.sleep(5)
    print("Operação longa concluída!")
    return "Resultado da operação longa"

async def main():
    try:
        # Tenta executar a operação com timeout de 2 segundos
        resultado = await asyncio.wait_for(operacao_longa(), timeout=2)
        print(f"Resultado: {resultado}")
    except asyncio.TimeoutError:
        print("A operação excedeu o timeout!")

asyncio.run(main())
```

Executando Código Bloqueante com Executors

Para executar código bloqueante (como operações de I/O síncronas ou código CPU-bound) sem bloquear o event loop, podemos usar executors:

```
import asyncio
import time
import concurrent.futures

def operacao_bloqueante(tempo):
    """Uma função bloqueante que simula uma operação intensiva em CPU."""
    print(f"Iniciando operação bloqueante por {tempo} segundos")
    time.sleep(tempo)  # Bloqueia a thread
    print(f"Operação bloqueante de {tempo} segundos concluída")
    return f"Resultado após {tempo} segundos"

async def main():
    print("Iniciando main()")

    # Criando um executor de threads
    with concurrent.futures.ThreadPoolExecutor() as pool:
        # Executando operações bloqueantes em threads separadas
        resultados = await asyncio.gather(
            asyncio.to_thread(operacao_bloqueante, 2),
            asyncio.to_thread(operacao_bloqueante, 1),
            asyncio.to_thread(operacao_bloqueante, 3)
        )

        print(f"Resultados: {resultados}")

    print("main() concluído")

asyncio.run(main())
```

Comunicação entre Coroutines com Queues

```
import asyncio
import random

async def produtor(fila):
    """Produz itens e os coloca na fila assíncrona."""
    for i in range(5):
        item = random.randint(1, 100)
        await fila.put(item)
        print(f"Produtor colocou {item} na fila")
        await asyncio.sleep(random.random())

async def consumidor(fila):
    """Consome itens da fila assíncrona."""
    while True:
        try:
            item = await asyncio.wait_for(fila.get(), timeout=3)
            print(f"Consumidor obteve {item} da fila")
            fila.task_done()
            await asyncio.sleep(random.random() * 2)
        except asyncio.TimeoutError:
            print("Timeout, consumidor saindo")
            break

async def main():
    # Criando uma fila assíncrona
    fila = asyncio.Queue()

    # Criando tasks
    prod_task = asyncio.create_task(produtor(fila))
    cons_task = asyncio.create_task(consumidor(fila))

    # Esperando o produtor terminar
    await prod_task

    # Esperando a fila ser esvaziada
    await fila.join()

    # Cancelando o consumidor
    cons_task.cancel()
    try:
        await cons_task
    except asyncio.CancelledError:
        pass

asyncio.run(main())
```

Combinando Diferentes Abordagens

Em aplicações reais, muitas vezes combinamos diferentes abordagens de concorrência e paralelismo:

```
import asyncio
import concurrent.futures
```

```
import time
import random

# Função CPU-bound para executar em processos
def tarefa_cpu(n):
    """Função que realiza uma operação intensiva em CPU."""
    print(f"Iniciando tarefa CPU com n={n}")
    resultado = 0
    for i in range(n):
        resultado += i * i
    print(f"Concluindo tarefa CPU com n={n}")
    return resultado

# Função I/O-bound para executar em threads
def tarefa_io(tempo):
    """Função que simula uma operação I/O-bound."""
    print(f"Iniciando tarefa I/O por {tempo} segundos")
    time.sleep(tempo)  # Simula I/O bloqueante
    print(f"Concluindo tarefa I/O de {tempo} segundos")
    return f"Resultado I/O após {tempo} segundos"

# Coroutine assíncrona
async def tarefa_async(nome, tempo):
    """Coroutine que simula uma operação assíncrona."""
    print(f"Iniciando tarefa assíncrona {nome}")
    await asyncio.sleep(tempo)
    print(f"Concluindo tarefa assíncrona {nome}")
    return f"Resultado assíncrono {nome}"

async def main():
    print("Iniciando main()")

    # Criando executors
    process_pool = concurrent.futures.ProcessPoolExecutor(max_workers=2)
    thread_pool = concurrent.futures.ThreadPoolExecutor(max_workers=3)

    # Tarefas CPU-bound em processos
    cpu_tasks = [
        asyncio.get_event_loop().run_in_executor(process_pool, tarefa_cpu, 10000000),
        asyncio.get_event_loop().run_in_executor(process_pool, tarefa_cpu, 20000000)
    ]

    # Tarefas I/O-bound em threads
    io_tasks = [
        asyncio.get_event_loop().run_in_executor(thread_pool, tarefa_io, 2),
        asyncio.get_event_loop().run_in_executor(thread_pool, tarefa_io, 1),
        asyncio.get_event_loop().run_in_executor(thread_pool, tarefa_io, 3)
    ]

    # Tarefas assíncronas
    async_tasks = [
        tarefa_async("A", 2),
        tarefa_async("B", 1),
```

```
        tarefa_async("C", 3)
    ]

    # Executando todas as tarefas concorrentemente
    resultados = await asyncio.gather(
        *cpu_tasks,
        *io_tasks,
        *async_tasks
    )

    print(f"Número de resultados: {len(resultados)}")

    # Fechando os executors
    process_pool.shutdown()
    thread_pool.shutdown()

    print("main() concluído")

if __name__ == "__main__":
    inicio = time.time()
    asyncio.run(main())
    fim = time.time()

    print(f"Tempo total: {fim - inicio:.2f} segundos")
```

Escolhendo a Abordagem Certa

A escolha entre threading, multiprocessing e asyncio depende do tipo de tarefa:

- **Threading**: Melhor para tarefas I/O-bound onde o GIL não é um gargalo (e.g., requisições de rede, operações de arquivo).
- **Multiprocessing**: Melhor para tarefas CPU-bound que podem se beneficiar de múltiplos cores (e.g., processamento de imagem, cálculos numéricos).
- **Asyncio**: Melhor para tarefas I/O-bound com alta concorrência, especialmente em servidores web e aplicações de rede.

Boas Práticas

1. **Evite Compartilhar Estado**: Minimize o compartilhamento de estado entre threads ou processos para evitar problemas de sincronização.
2. **Use Locks com Cuidado**: Locks podem causar deadlocks se não forem usados corretamente. Use `with` para garantir que locks sejam liberados.
3. **Prefira Estruturas de Alto Nível**: Use `concurrent.futures` ou `asyncio` em vez de trabalhar diretamente com threads ou processos quando possível.
4. **Teste com Cargas Realistas**: O comportamento concorrente pode mudar drasticamente com diferentes cargas de trabalho.
5. **Monitore o Uso de Recursos**: Concorrência e paralelismo podem aumentar o uso de CPU e memória.
6. **Considere o Overhead**: Criar threads ou processos tem um custo. Para tarefas muito pequenas, o overhead pode superar os benefícios.

7. **Documente o Comportamento Concorrente**: Deixe claro como seu código lida com concorrência para facilitar a manutenção.

Conclusão

Python oferece várias ferramentas para implementar concorrência e paralelismo, cada uma com suas próprias vantagens e casos de uso. Entender as diferenças entre threading, multiprocessing e asyncio, bem como as limitações impostas pelo GIL, é essencial para escrever código Python eficiente e escalável.

No próximo tópico, exploraremos testes unitários em Python, uma prática essencial para garantir a qualidade e a confiabilidade do código.

Módulo 3: Python Avançado

Testes Unitários

Os testes unitários são uma prática fundamental na engenharia de software que consiste em testar individualmente componentes ou unidades de código para garantir que funcionem conforme o esperado. Em Python, existem várias ferramentas e frameworks que facilitam a escrita e execução de testes unitários, permitindo que os desenvolvedores criem software mais confiável e de fácil manutenção.

Por que Testar?

Antes de mergulharmos nas ferramentas e técnicas, é importante entender por que os testes são essenciais:

1. **Detecção precoce de bugs**: Identificar problemas antes que cheguem à produção.
2. **Facilitar refatoração**: Garantir que mudanças no código não quebrem funcionalidades existentes.
3. **Documentação viva**: Testes bem escritos documentam como o código deve funcionar.
4. **Design de código melhor**: Escrever código testável geralmente leva a um design mais modular e coeso.
5. **Confiança nas mudanças**: Fazer alterações com a segurança de que os testes detectarão regressões.

O Módulo unittest

O `unittest` é um framework de testes integrado à biblioteca padrão do Python, inspirado no JUnit do Java. Ele fornece uma estrutura para organizar e executar testes.

Conceitos Básicos do unittest

- **TestCase**: Classe base para criar casos de teste.
- **Assertions**: Métodos para verificar condições esperadas.
- **Test Fixtures**: Métodos para configurar e limpar o ambiente de teste.
- **Test Runner**: Componente que executa os testes e apresenta os resultados.

Exemplo Básico com unittest

Vamos criar um exemplo simples para testar uma função que calcula o fatorial de um número:

```python
# arquivo: matematica.py
def fatorial(n):
    """Calcula o fatorial de n."""
    if not isinstance(n, int):
        raise TypeError("n deve ser um inteiro")
    if n < 0:
        raise ValueError("n deve ser não-negativo")
    if n == 0:
        return 1
    return n * fatorial(n - 1)
```

Agora, vamos escrever testes para esta função:

```
# arquivo: test_matematica.py
import unittest
from matematica import fatorial

class TestFatorial(unittest.TestCase):

    def test_fatorial_de_0(self):
        """Testa o fatorial de 0."""
        self.assertEqual(fatorial(0), 1)

    def test_fatorial_de_1(self):
        """Testa o fatorial de 1."""
        self.assertEqual(fatorial(1), 1)

    def test_fatorial_de_5(self):
        """Testa o fatorial de 5."""
        self.assertEqual(fatorial(5), 120)

    def test_fatorial_de_numero_negativo(self):
        """Testa se um ValueError é levantado para números negativos."""
        with self.assertRaises(ValueError):
            fatorial(-1)

    def test_fatorial_de_tipo_invalido(self):
        """Testa se um TypeError é levantado para tipos não inteiros."""
        with self.assertRaises(TypeError):
            fatorial(3.5)

if __name__ == '__main__':
    unittest.main()
```

Para executar os testes:

```
python test_matematica.py
```

Ou usando o módulo `unittest` diretamente:

```
python -m unittest test_matematica.py
```

Métodos de Assertion Comuns

O `unittest` fornece vários métodos de assertion para verificar diferentes condições:

- `assertEqual(a, b)`: Verifica se a == b
- `assertNotEqual(a, b)`: Verifica se a != b
- `assertTrue(x)`: Verifica se bool(x) é True
- `assertFalse(x)`: Verifica se bool(x) é False
- `assertIs(a, b)`: Verifica se a é b (identidade)
- `assertIsNot(a, b)`: Verifica se a não é b (identidade)
- `assertIsNone(x)`: Verifica se x é None
- `assertIsNotNone(x)`: Verifica se x não é None
- `assertIn(a, b)`: Verifica se a está em b
- `assertNotIn(a, b)`: Verifica se a não está em b
- `assertIsInstance(a, b)`: Verifica se a é uma instância de b
- `assertNotIsInstance(a, b)`: Verifica se a não é uma instância de b

- `assertRaises(exc, func, *args, **kwargs)`: Verifica se func(*args, **kwargs) levanta a exceção exc
- `assertAlmostEqual(a, b)`: Verifica se a e b são aproximadamente iguais (útil para números de ponto flutuante)

Test Fixtures

Os test fixtures são métodos que preparam o ambiente para os testes e o limpam após a execução:

- `setUp()`: Executado antes de cada método de teste.
- `tearDown()`: Executado após cada método de teste.
- `setUpClass()`: Executado uma vez antes de todos os testes da classe.
- `tearDownClass()`: Executado uma vez após todos os testes da classe.

Exemplo:

```
import unittest
import tempfile
import os

class TestArquivo(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        """Configuração executada uma vez antes de todos os testes."""
        print("Iniciando testes da classe TestArquivo")

    @classmethod
    def tearDownClass(cls):
        """Limpeza executada uma vez após todos os testes."""
        print("Finalizando testes da classe TestArquivo")

    def setUp(self):
        """Configuração executada antes de cada teste."""
        # Cria um arquivo temporário para os testes
        self.arquivo_temp = tempfile.NamedTemporaryFile(delete=False)
        self.arquivo_temp.write(b"Conteudo de teste")
        self.arquivo_temp.close()

    def tearDown(self):
        """Limpeza executada após cada teste."""
        # Remove o arquivo temporário
        os.unlink(self.arquivo_temp.name)

    def test_arquivo_existe(self):
        """Testa se o arquivo existe."""
        self.assertTrue(os.path.exists(self.arquivo_temp.name))

    def test_arquivo_conteudo(self):
        """Testa o conteúdo do arquivo."""
        with open(self.arquivo_temp.name, 'rb') as f:
            conteudo = f.read()
        self.assertEqual(conteudo, b"Conteudo de teste")

if __name__ == '__main__':
    unittest.main()
```

Organizando Testes em Suites

Podemos organizar testes em suites para executá-los em conjunto:

```
import unittest
from test_matematica import TestFatorial
from test_arquivo import TestArquivo

def suite():
    """Cria uma suite de testes."""
    test_suite = unittest.TestSuite()
    test_suite.addTest(unittest.makeSuite(TestFatorial))
    test_suite.addTest(unittest.makeSuite(TestArquivo))
    return test_suite

if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    runner.run(suite())
```

O Módulo pytest

O `pytest` é um framework de testes alternativo que oferece uma sintaxe mais simples e recursos avançados. Embora não faça parte da biblioteca padrão, é amplamente utilizado na comunidade Python.

Instalação do pytest

```
pip install pytest
```

Exemplo Básico com pytest

Usando o mesmo exemplo de fatorial, os testes com pytest seriam assim:

```
# arquivo: test_matematica_pytest.py
import pytest
from matematica import fatorial

def test_fatorial_de_0():
    """Testa o fatorial de 0."""
    assert fatorial(0) == 1

def test_fatorial_de_1():
    """Testa o fatorial de 1."""
    assert fatorial(1) == 1

def test_fatorial_de_5():
    """Testa o fatorial de 5."""
    assert fatorial(5) == 120

def test_fatorial_de_numero_negativo():
    """Testa se um ValueError é levantado para números negativos."""
    with pytest.raises(ValueError):
        fatorial(-1)

def test_fatorial_de_tipo_invalido():
    """Testa se um TypeError é levantado para tipos não inteiros."""
```

```
    with pytest.raises(TypeError):
        fatorial(3.5)
```

Para executar os testes:

```
pytest test_matematica_pytest.py
```

Fixtures no pytest

O pytest tem um sistema de fixtures mais flexível que o unittest:

```
import pytest
import tempfile
import os

@pytest.fixture
def arquivo_temporario():
    """Fixture que cria um arquivo temporário para os testes."""
    arquivo_temp = tempfile.NamedTemporaryFile(delete=False)
    arquivo_temp.write(b"Conteudo de teste")
    arquivo_temp.close()

    # Retorna o caminho do arquivo para o teste
    yield arquivo_temp.name

    # Código de limpeza executado após o teste
    os.unlink(arquivo_temp.name)

def test_arquivo_existe(arquivo_temporario):
    """Testa se o arquivo existe."""
    assert os.path.exists(arquivo_temporario)

def test_arquivo_conteudo(arquivo_temporario):
    """Testa o conteúdo do arquivo."""
    with open(arquivo_temporario, 'rb') as f:
        conteudo = f.read()
    assert conteudo == b"Conteudo de teste"
```

Parametrização de Testes

O pytest permite parametrizar testes para executá-los com diferentes entradas:

```
import pytest
from matematica import fatorial

@pytest.mark.parametrize("entrada,esperado", [
    (0, 1),
    (1, 1),
    (2, 2),
    (3, 6),
    (4, 24),
    (5, 120)
])
def test_fatorial(entrada, esperado):
    """Testa o fatorial com vários valores."""
    assert fatorial(entrada) == esperado
```

Marcadores (Markers)

Os marcadores permitem categorizar testes e executar subconjuntos específicos:

```
import pytest

@pytest.mark.rapido
def test_funcao_rapida():
    assert 1 + 1 == 2

@pytest.mark.lento
def test_funcao_lenta():
    import time
    time.sleep(1)
    assert 2 + 2 == 4
```

Para executar apenas testes rápidos:

```
pytest -m rapido
```

Cobertura de Código

A cobertura de código mede quanto do seu código é executado pelos testes. Podemos usar o pacote `pytest-cov`:

```
pip install pytest-cov
```

Para executar testes com relatório de cobertura:

```
pytest --cov=matematica test_matematica_pytest.py
```

Para um relatório HTML detalhado:

```
pytest --cov=matematica --cov-report=html test_matematica_pytest.py
```

Mocks e Patches

Mocks são objetos que simulam o comportamento de objetos reais de forma controlada. Eles são úteis para isolar o código que está sendo testado de suas dependências.

Usando unittest.mock

O módulo `unittest.mock` (disponível a partir do Python 3.3) fornece classes para substituir partes do sistema em teste:

```
# arquivo: servico.py
import requests

def obter_dados_usuario(id_usuario):
    """Obtém dados de um usuário de uma API externa."""
    response = requests.get(f"https://api.exemplo.com/usuarios/{id_usuario}")
    if response.status_code == 200:
        return response.json()
    return None
```

Teste usando mock:

```
import unittest
from unittest.mock import patch
from servico import obter_dados_usuario
```

```
class TestServico(unittest.TestCase):

    @patch('servico.requests.get')
    def test_obter_dados_usuario_sucesso(self, mock_get):
        """Testa obter_dados_usuario quando a API retorna sucesso."""
        # Configura o mock
        mock_response = unittest.mock.Mock()
        mock_response.status_code = 200
        mock_response.json.return_value = {"id": 1, "nome": "João"}
        mock_get.return_value = mock_response

        # Chama a função
        resultado = obter_dados_usuario(1)

        # Verifica o resultado
        self.assertEqual(resultado, {"id": 1, "nome": "João"})
        mock_get.assert_called_once_with("https://api.exemplo.com/usuarios/1")

    @patch('servico.requests.get')
    def test_obter_dados_usuario_falha(self, mock_get):
        """Testa obter_dados_usuario quando a API retorna erro."""
        # Configura o mock
        mock_response = unittest.mock.Mock()
        mock_response.status_code = 404
        mock_get.return_value = mock_response

        # Chama a função
        resultado = obter_dados_usuario(999)

        # Verifica o resultado
        self.assertIsNone(resultado)
        mock_get.assert_called_once_with("https://api.exemplo.com/usuarios/999")

if __name__ == '__main__':
    unittest.main()
```

Usando pytest-mock

O pytest-mock é um plugin que fornece uma fixture `mocker` para facilitar o uso de mocks no pytest:

```
pip install pytest-mock
```

```
import pytest
from servico import obter_dados_usuario

def test_obter_dados_usuario_sucesso(mocker):
    """Testa obter_dados_usuario quando a API retorna sucesso."""
    # Configura o mock
    mock_get = mocker.patch('servico.requests.get')
    mock_response = mocker.Mock()
    mock_response.status_code = 200
    mock_response.json.return_value = {"id": 1, "nome": "João"}
    mock_get.return_value = mock_response
```

```
    # Chama a função
    resultado = obter_dados_usuario(1)

    # Verifica o resultado
    assert resultado == {"id": 1, "nome": "João"}
    mock_get.assert_called_once_with("https://api.exemplo.com/usuarios/1")

def test_obter_dados_usuario_falha(mocker):
    """Testa obter_dados_usuario quando a API retorna erro."""
    # Configura o mock
    mock_get = mocker.patch('servico.requests.get')
    mock_response = mocker.Mock()
    mock_response.status_code = 404
    mock_get.return_value = mock_response

    # Chama a função
    resultado = obter_dados_usuario(999)

    # Verifica o resultado
    assert resultado is None
    mock_get.assert_called_once_with("https://api.exemplo.com/usuarios/999")
```

Test-Driven Development (TDD)

O Test-Driven Development é uma metodologia de desenvolvimento que inverte o fluxo tradicional:

1. Escreva um teste que falha para a funcionalidade desejada.
2. Implemente o código mínimo para fazer o teste passar.
3. Refatore o código mantendo os testes passando.

Exemplo de TDD para implementar uma função de validação de e-mail:

```
# Passo 1: Escrever o teste primeiro
import pytest

def test_validar_email_valido():
    """Testa se emails válidos são aceitos."""
    assert validar_email("usuario@exemplo.com") is True
    assert validar_email("usuario.nome@empresa-exemplo.com.br") is True

def test_validar_email_invalido():
    """Testa se emails inválidos são rejeitados."""
    assert validar_email("usuario@") is False
    assert validar_email("@exemplo.com") is False
    assert validar_email("usuario@.com") is False
    assert validar_email("usuario@exemplo") is False
    assert validar_email("usuario@exemplo.") is False
    assert validar_email("") is False
    assert validar_email(None) is False

# Passo 2: Implementar o código mínimo para fazer o teste passar
import re

def validar_email(email):
    """Valida um endereço de e-mail."""
```

```
    if email is None:
        return False

    padrao = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    return bool(re.match(padrao, email))

# Passo 3: Refatorar se necessário
```

Testes de Integração

Enquanto os testes unitários testam componentes isoladamente, os testes de integração verificam como diferentes partes do sistema funcionam juntas.

Exemplo de teste de integração para um sistema de banco de dados:

```
import unittest
import sqlite3
from banco_dados import BancoDados

class TestBancoDadosIntegracao(unittest.TestCase):

    def setUp(self):
        """Configura um banco de dados de teste."""
        self.conexao = sqlite3.connect(':memory:')  # Banco de dados em memória
        self.banco = BancoDados(self.conexao)
        self.banco.criar_tabela_usuarios()

    def tearDown(self):
        """Fecha a conexão com o banco de dados."""
        self.conexao.close()

    def test_inserir_e_buscar_usuario(self):
        """Testa inserir um usuário e depois buscá-lo."""
        # Insere um usuário
        usuario_id = self.banco.inserir_usuario("João", "joao@exemplo.com")

        # Busca o usuário
        usuario = self.banco.buscar_usuario_por_id(usuario_id)

        # Verifica se os dados estão corretos
        self.assertEqual(usuario["nome"], "João")
        self.assertEqual(usuario["email"], "joao@exemplo.com")

    def test_atualizar_usuario(self):
        """Testa atualizar os dados de um usuário."""
        # Insere um usuário
        usuario_id = self.banco.inserir_usuario("Maria", "maria@exemplo.com")

        # Atualiza o usuário
        self.banco.atualizar_usuario(usuario_id, nome="Maria Silva")

        # Busca o usuário atualizado
        usuario = self.banco.buscar_usuario_por_id(usuario_id)
```

```
        # Verifica se os dados foram atualizados
        self.assertEqual(usuario["nome"], "Maria Silva")
        self.assertEqual(usuario["email"], "maria@exemplo.com")

    def test_deletar_usuario(self):
        """Testa deletar um usuário."""
        # Insere um usuário
        usuario_id = self.banco.inserir_usuario("Carlos", "carlos@exemplo.com")

        # Verifica se o usuário existe
        usuario = self.banco.buscar_usuario_por_id(usuario_id)
        self.assertIsNotNone(usuario)

        # Deleta o usuário
        self.banco.deletar_usuario(usuario_id)

        # Verifica se o usuário foi deletado
        usuario = self.banco.buscar_usuario_por_id(usuario_id)
        self.assertIsNone(usuario)

if __name__ == '__main__':
    unittest.main()
```

Boas Práticas em Testes

1. **Testes Independentes**: Cada teste deve ser independente dos outros. Um teste não deve depender do estado deixado por outro teste.
2. **Testes Determinísticos**: Os testes devem produzir o mesmo resultado toda vez que são executados, independentemente da ordem ou do ambiente.
3. **Testes Rápidos**: Os testes devem ser rápidos para incentivar a execução frequente.
4. **Testes Legíveis**: Os nomes dos testes devem descrever claramente o que está sendo testado. Use nomes descritivos como `test_usuario_nao_pode_ter_senha_vazia`.
5. **Cobertura Adequada**: Busque uma boa cobertura de código, mas lembre-se que 100% de cobertura não garante ausência de bugs.
6. **Teste Comportamento, Não Implementação**: Teste o que o código faz, não como ele faz. Isso permite refatorar a implementação sem quebrar os testes.
7. **Evite Lógica Complexa nos Testes**: Os testes devem ser simples e diretos. Se você precisa de lógica complexa em um teste, isso pode ser um sinal de que o código sendo testado é muito complexo.
8. **Use Fixtures e Helpers**: Extraia código comum de configuração para fixtures ou helpers para manter os testes DRY (Don't Repeat Yourself).
9. **Teste Casos de Borda**: Além dos casos normais, teste casos de borda como valores vazios, nulos, negativos, muito grandes, etc.
10. **Teste Exceções**: Verifique se o código lança as exceções esperadas em condições de erro.

Ferramentas Adicionais

doctest

O módulo `doctest` permite escrever testes dentro das docstrings do código:

```python
def soma(a, b):
    """
    Retorna a soma de dois números.

    >>> soma(2, 3)
    5
    >>> soma(-1, 1)
    0
    >>> soma(0, 0)
    0
    """
    return a + b

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

tox

O `tox` é uma ferramenta que automatiza testes em diferentes ambientes Python:

```
# arquivo: tox.ini
[tox]
envlist = py36,py37,py38,py39

[testenv]
deps = pytest
commands = pytest
```

Hypothesis

O `hypothesis` é uma biblioteca para testes baseados em propriedades, que gera automaticamente casos de teste:

```python
from hypothesis import given
from hypothesis import strategies as st

@given(st.integers(), st.integers())
def test_soma_comutativa(a, b):
    """Testa se a + b == b + a para quaisquer inteiros a e b."""
    assert soma(a, b) == soma(b, a)
```

Conclusão

Os testes unitários são uma parte essencial do desenvolvimento de software moderno. Eles ajudam a garantir que o código funcione conforme o esperado, facilitam a refatoração e servem como documentação viva do comportamento esperado do sistema.

Python oferece várias ferramentas para testes, desde o módulo `unittest` integrado até frameworks de terceiros como `pytest`. A escolha da ferramenta depende das necessidades específicas do projeto e das preferências da equipe.

Independentemente da ferramenta escolhida, o importante é adotar uma cultura de testes, onde escrever testes é parte integrante do processo de desenvolvimento, não uma atividade separada ou opcional.

No próximo tópico, apresentaremos exercícios práticos que aplicam os conceitos avançados de Python que

aprendemos neste módulo, incluindo decoradores, geradores, expressões regulares, programação funcional, concorrência e testes unitários.

Módulo 3: Python Avançado

Exercícios Práticos

Nesta seção, apresentamos uma série de exercícios práticos para consolidar os conhecimentos avançados de Python que adquirimos neste módulo. Estes exercícios abordam decoradores, geradores, expressões regulares, programação funcional, concorrência e testes unitários.

Exercício 1: Decoradores

Crie um sistema de registro de tempo de execução de funções usando decoradores.

Requisitos: 1. Implemente um decorador `@registrar_tempo` que mede o tempo de execução de uma função. 2. O decorador deve imprimir o nome da função e o tempo que levou para executar. 3. Implemente um segundo decorador `@registrar_chamadas` que conta quantas vezes uma função foi chamada. 4. Combine os dois decoradores para registrar tanto o tempo quanto o número de chamadas.

Solução:

```
import time
import functools

def registrar_tempo(funcao):
    """Decorador que registra o tempo de execução de uma função."""
    @functools.wraps(funcao)
    def wrapper(*args, **kwargs):
        inicio = time.time()
        resultado = funcao(*args, **kwargs)
        fim = time.time()
        print(f"A função {funcao.__name__} levou {fim - inicio:.4f} segundos para executar.")
        return resultado
    return wrapper

def registrar_chamadas(funcao):
    """Decorador que conta o número de chamadas de uma função."""
    contador = 0

    @functools.wraps(funcao)
    def wrapper(*args, **kwargs):
        nonlocal contador
        contador += 1
        print(f"A função {funcao.__name__} foi chamada {contador} vezes.")
        return funcao(*args, **kwargs)

    return wrapper
```

```python
# Exemplo de uso
@registrar_tempo
@registrar_chamadas
def fibonacci(n):
    """Calcula o n-ésimo número de Fibonacci recursivamente."""
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

# Testando
print(fibonacci(10))  # Deve imprimir o número de chamadas e o tempo para cada chamada recursiva
```

Exercício 2: Geradores e Iteradores

Implemente um gerador de números primos eficiente e use-o para resolver problemas matemáticos.

Requisitos: 1. Crie um gerador `primos()` que gere números primos indefinidamente. 2. Implemente uma função que use o gerador para encontrar a soma dos primeiros N números primos. 3. Implemente uma função que use o gerador para encontrar o primeiro número primo maior que um valor dado.

Solução:

```python
def primos():
    """Gerador que produz números primos indefinidamente."""
    # 2 é o primeiro número primo
    yield 2

    # Lista para armazenar os primos já encontrados
    primos_encontrados = [2]

    # Começar a verificar a partir do 3
    numero = 3

    while True:
        # Verificar se o número é divisível por algum primo já encontrado
        eh_primo = True
        for primo in primos_encontrados:
            # Podemos parar de verificar quando o primo for maior que a raiz quadrada do número
            if primo * primo > numero:
                break

            if numero % primo == 0:
                eh_primo = False
                break

        if eh_primo:
            primos_encontrados.append(numero)
            yield numero

        # Avançar para o próximo número ímpar (os pares não são primos, exceto o 2)
        numero += 2

def soma_primeiros_n_primos(n):
    """Calcula a soma dos primeiros n números primos."""
```

```
    gerador = primos()
    return sum(next(gerador) for _ in range(n))

def primeiro_primo_maior_que(valor):
    """Encontra o primeiro número primo maior que o valor dado."""
    gerador = primos()
    primo = next(gerador)

    while primo <= valor:
        primo = next(gerador)

    return primo

# Testando
print(f"Soma dos primeiros 10 primos: {soma_primeiros_n_primos(10)}")
print(f"Primeiro primo maior que 100: {primeiro_primo_maior_que(100)}")

# Listar os primeiros 20 números primos
gerador = primos()
primeiros_20 = [next(gerador) for _ in range(20)]
print(f"Primeiros 20 primos: {primeiros_20}")
```

Exercício 3: Expressões Regulares

Crie um validador e extrator de informações de textos usando expressões regulares.

Requisitos: 1. Implemente uma função que valide números de telefone em diferentes formatos (ex: (11) 98765-4321, 11 987654321, etc.). 2. Implemente uma função que extraia todos os e-mails de um texto. 3. Implemente uma função que substitua todas as ocorrências de CPF (formato: 123.456.789-00) em um texto por "CPF ocultado".

Solução:

```
import re

def validar_telefone(telefone):
    """
    Valida números de telefone em diferentes formatos.
    Formatos válidos:
    - (11) 98765-4321
    - (11) 987654321
    - 11 98765-4321
    - 11 987654321
    - 11987654321
    """
    padrao = r'^\(?(\d{2})\)?\s?(\d{5})-?(\d{4})$'
    match = re.match(padrao, telefone)

    if match:
        # Formatar o telefone de maneira padronizada
        ddd, parte1, parte2 = match.groups()
        return f"({ddd}) {parte1}-{parte2}"

    return None
```

```python
def extrair_emails(texto):
    """Extrai todos os endereços de e-mail de um texto."""
    padrao = r'[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}'
    return re.findall(padrao, texto)

def ocultar_cpf(texto):
    """Substitui CPFs por 'CPF ocultado' em um texto."""
    padrao = r'\d{3}\.\d{3}\.\d{3}-\d{2}'
    return re.sub(padrao, "CPF ocultado", texto)

# Testando validação de telefone
telefones = [
    "(11) 98765-4321",
    "(11) 987654321",
    "11 98765-4321",
    "11 987654321",
    "11987654321",
    "123 45678",  # Inválido
    "abc12345678"  # Inválido
]

for tel in telefones:
    resultado = validar_telefone(tel)
    if resultado:
        print(f"{tel} é válido: {resultado}")
    else:
        print(f"{tel} é inválido")

# Testando extração de e-mails
texto_emails = """
Contatos:
- João: joao@empresa.com.br
- Maria: maria.silva@gmail.com
- Site: contato@site.com
Visite www.site.com para mais informações.
"""

emails = extrair_emails(texto_emails)
print(f"\nE-mails encontrados: {emails}")

# Testando ocultação de CPF
texto_cpf = """
Dados do cliente:
Nome: João Silva
CPF: 123.456.789-00
Endereço: Rua Exemplo, 123
Outro cliente: Maria Souza, CPF: 987.654.321-00
"""

texto_seguro = ocultar_cpf(texto_cpf)
print(f"\nTexto com CPFs ocultados:\n{texto_seguro}")
```

Exercício 4: Programação Funcional

Implemente um sistema de processamento de dados usando conceitos de programação funcional.

Requisitos: 1. Crie uma lista de dicionários representando pessoas com nome, idade e salário. 2. Use funções de ordem superior (map, filter, reduce) para: - Filtrar pessoas com mais de 30 anos - Aumentar o salário de cada pessoa em 10% - Calcular a média salarial 3. Implemente as mesmas operações usando compreensões de lista.

Solução:

```
from functools import reduce

# Lista de pessoas
pessoas = [
    {"nome": "Ana", "idade": 28, "salario": 3500},
    {"nome": "Bruno", "idade": 35, "salario": 4200},
    {"nome": "Carla", "idade": 42, "salario": 5100},
    {"nome": "Daniel", "idade": 25, "salario": 3000},
    {"nome": "Elena", "idade": 33, "salario": 4800}
]

# Usando funções de ordem superior
print("Usando funções de ordem superior:")

# Filtrar pessoas com mais de 30 anos
pessoas_mais_30 = list(filter(lambda p: p["idade"] > 30, pessoas))
print(f"Pessoas com mais de 30 anos: {len(pessoas_mais_30)}")
for p in pessoas_mais_30:
    print(f"- {p['nome']}, {p['idade']} anos, R${p['salario']}")

# Aumentar o salário em 10%
pessoas_aumento = list(map(
    lambda p: {**p, "salario": p["salario"] * 1.1},
    pessoas
))
print("\nPessoas com aumento de 10%:")
for p in pessoas_aumento:
    print(f"- {p['nome']}, R${p['salario']:.2f}")

# Calcular a média salarial
media_salarial = reduce(
    lambda acc, p: acc + p["salario"],
    pessoas,
    0
) / len(pessoas)
print(f"\nMédia salarial: R${media_salarial:.2f}")

# Usando compreensões de lista
print("\nUsando compreensões de lista:")

# Filtrar pessoas com mais de 30 anos
pessoas_mais_30_comp = [p for p in pessoas if p["idade"] > 30]
print(f"Pessoas com mais de 30 anos: {len(pessoas_mais_30_comp)}")
for p in pessoas_mais_30_comp:
    print(f"- {p['nome']}, {p['idade']} anos, R${p['salario']}")
```

```python
# Aumentar o salário em 10%
pessoas_aumento_comp = [{**p, "salario": p["salario"] * 1.1} for p in pessoas]
print("\nPessoas com aumento de 10%:")
for p in pessoas_aumento_comp:
    print(f"- {p['nome']}, R${p['salario']:.2f}")

# Calcular a média salarial
media_salarial_comp = sum(p["salario"] for p in pessoas) / len(pessoas)
print(f"\nMédia salarial: R${media_salarial_comp:.2f}")
```

Exercício 5: Concorrência e Paralelismo

Implemente um sistema de download concorrente de múltiplas URLs.

Requisitos: 1. Crie uma função que baixe o conteúdo de uma URL e salve em um arquivo. 2. Implemente três versões da função para baixar múltiplas URLs: - Versão sequencial (sem concorrência) - Versão com threads - Versão com asyncio 3. Compare o tempo de execução das três versões.

Solução:

```python
import requests
import time
import threading
import asyncio
import aiohttp
import os

# Lista de URLs para baixar
urls = [
    "https://www.python.org",
    "https://docs.python.org",
    "https://pypi.org",
    "https://www.djangoproject.com",
    "https://flask.palletsprojects.com",
    "https://www.numpy.org",
    "https://pandas.pydata.org",
    "https://matplotlib.org",
    "https://scikit-learn.org",
    "https://www.tensorflow.org"
]

# Criar diretório para salvar os arquivos
os.makedirs("downloads", exist_ok=True)

def baixar_url(url):
    """Baixa o conteúdo de uma URL e salva em um arquivo."""
    try:
        nome_arquivo = url.split("//")[1].split("/")[0] + ".html"
        caminho_arquivo = os.path.join("downloads", nome_arquivo)

        response = requests.get(url, timeout=10)
        with open(caminho_arquivo, "w", encoding="utf-8") as f:
            f.write(response.text)
```

```
        print(f"Baixado: {url} -> {caminho_arquivo}")
        return True
    except Exception as e:
        print(f"Erro ao baixar {url}: {e}")
        return False

# Versão 1: Sequencial
def baixar_sequencial(urls):
    """Baixa URLs sequencialmente."""
    print("\nIniciando download sequencial...")
    inicio = time.time()

    resultados = []
    for url in urls:
        resultados.append(baixar_url(url))

    fim = time.time()
    print(f"Download sequencial concluído em {fim - inicio:.2f} segundos.")
    print(f"Sucesso: {resultados.count(True)}/{len(resultados)}")
    return fim - inicio

# Versão 2: Com Threads
def baixar_com_threads(urls):
    """Baixa URLs usando threads."""
    print("\nIniciando download com threads...")
    inicio = time.time()

    threads = []
    for url in urls:
        thread = threading.Thread(target=baixar_url, args=(url,))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

    fim = time.time()
    print(f"Download com threads concluído em {fim - inicio:.2f} segundos.")
    return fim - inicio

# Versão 3: Com Asyncio
async def baixar_url_async(url, session):
    """Baixa o conteúdo de uma URL de forma assíncrona."""
    try:
        nome_arquivo = url.split("//")[1].split("/")[0] + ".html"
        caminho_arquivo = os.path.join("downloads", nome_arquivo)

        async with session.get(url, timeout=10) as response:
            conteudo = await response.text()
            with open(caminho_arquivo, "w", encoding="utf-8") as f:
                f.write(conteudo)

        print(f"Baixado (async): {url} -> {caminho_arquivo}")
```

```
            return True
        except Exception as e:
            print(f"Erro ao baixar {url} (async): {e}")
            return False

async def baixar_com_asyncio_impl(urls):
    """Implementação do download assíncrono."""
    async with aiohttp.ClientSession() as session:
        tarefas = [baixar_url_async(url, session) for url in urls]
        return await asyncio.gather(*tarefas)

def baixar_com_asyncio(urls):
    """Baixa URLs usando asyncio."""
    print("\nIniciando download com asyncio...")
    inicio = time.time()

    resultados = asyncio.run(baixar_com_asyncio_impl(urls))

    fim = time.time()
    print(f"Download com asyncio concluído em {fim - inicio:.2f} segundos.")
    print(f"Sucesso: {resultados.count(True)}/{len(resultados)}")
    return fim - inicio

# Executar e comparar as três versões
if __name__ == "__main__":
    print("Comparação de métodos de download:")

    tempo_sequencial = baixar_sequencial(urls)
    tempo_threads = baixar_com_threads(urls)
    tempo_asyncio = baixar_com_asyncio(urls)

    print("\nResultados:")
    print(f"Sequencial: {tempo_sequencial:.2f} segundos")
    print(f"Threads: {tempo_threads:.2f} segundos")
    print(f"Asyncio: {tempo_asyncio:.2f} segundos")

    # Calcular a aceleração
    aceleracao_threads = tempo_sequencial / tempo_threads
    aceleracao_asyncio = tempo_sequencial / tempo_asyncio

    print(f"\nAceleração com threads: {aceleracao_threads:.2f}x")
    print(f"Aceleração com asyncio: {aceleracao_asyncio:.2f}x")
```

Exercício 6: Testes Unitários

Implemente um sistema de gerenciamento de tarefas com testes unitários.

Requisitos: 1. Crie uma classe `GerenciadorTarefas` com métodos para adicionar, remover, marcar como concluída e listar tarefas. 2. Implemente testes unitários para cada método usando unittest ou pytest. 3. Use mocks para simular a persistência de dados em um arquivo.

Solução:

```
# gerenciador_tarefas.py
import json
```

```
import os
from datetime import datetime

class Tarefa:
    def __init__(self, id, titulo, descricao="", concluida=False):
        self.id = id
        self.titulo = titulo
        self.descricao = descricao
        self.concluida = concluida
        self.data_criacao = datetime.now().isoformat()
        self.data_conclusao = None

    def marcar_como_concluida(self):
        self.concluida = True
        self.data_conclusao = datetime.now().isoformat()

    def desmarcar_como_concluida(self):
        self.concluida = False
        self.data_conclusao = None

    def to_dict(self):
        return {
            "id": self.id,
            "titulo": self.titulo,
            "descricao": self.descricao,
            "concluida": self.concluida,
            "data_criacao": self.data_criacao,
            "data_conclusao": self.data_conclusao
        }

    @classmethod
    def from_dict(cls, dados):
        tarefa = cls(
            dados["id"],
            dados["titulo"],
            dados.get("descricao", ""),
            dados.get("concluida", False)
        )
        tarefa.data_criacao = dados.get("data_criacao", datetime.now().isoformat())
        tarefa.data_conclusao = dados.get("data_conclusao", None)
        return tarefa

class GerenciadorTarefas:
    def __init__(self, arquivo_dados="tarefas.json"):
        self.arquivo_dados = arquivo_dados
        self.tarefas = []
        self.carregar_tarefas()

    def carregar_tarefas(self):
        """Carrega as tarefas do arquivo."""
        if os.path.exists(self.arquivo_dados):
            try:
                with open(self.arquivo_dados, "r") as f:
```

```
                dados = json.load(f)
                self.tarefas = [Tarefa.from_dict(t) for t in dados]
        except Exception as e:
            print(f"Erro ao carregar tarefas: {e}")
            self.tarefas = []

def salvar_tarefas(self):
    """Salva as tarefas no arquivo."""
    try:
        with open(self.arquivo_dados, "w") as f:
            dados = [t.to_dict() for t in self.tarefas]
            json.dump(dados, f, indent=2)
        return True
    except Exception as e:
        print(f"Erro ao salvar tarefas: {e}")
        return False

def adicionar_tarefa(self, titulo, descricao=""):
    """Adiciona uma nova tarefa."""
    # Gerar um novo ID (maior ID existente + 1, ou 1 se não houver tarefas)
    novo_id = max([t.id for t in self.tarefas], default=0) + 1

    tarefa = Tarefa(novo_id, titulo, descricao)
    self.tarefas.append(tarefa)
    self.salvar_tarefas()
    return tarefa

def remover_tarefa(self, id_tarefa):
    """Remove uma tarefa pelo ID."""
    tarefa = self.obter_tarefa(id_tarefa)
    if tarefa:
        self.tarefas.remove(tarefa)
        self.salvar_tarefas()
        return True
    return False

def marcar_como_concluida(self, id_tarefa):
    """Marca uma tarefa como concluída."""
    tarefa = self.obter_tarefa(id_tarefa)
    if tarefa:
        tarefa.marcar_como_concluida()
        self.salvar_tarefas()
        return True
    return False

def desmarcar_como_concluida(self, id_tarefa):
    """Desmarca uma tarefa como concluída."""
    tarefa = self.obter_tarefa(id_tarefa)
    if tarefa:
        tarefa.desmarcar_como_concluida()
        self.salvar_tarefas()
        return True
    return False
```

```
    def obter_tarefa(self, id_tarefa):
        """Obtém uma tarefa pelo ID."""
        for tarefa in self.tarefas:
            if tarefa.id == id_tarefa:
                return tarefa
        return None

    def listar_tarefas(self, apenas_concluidas=None):
        """Lista todas as tarefas, opcionalmente filtrando por status."""
        if apenas_concluidas is None:
            return self.tarefas

        return [t for t in self.tarefas if t.concluida == apenas_concluidas]
```

```
# test_gerenciador_tarefas.py
import unittest
from unittest.mock import patch, mock_open
import json
import os
from gerenciador_tarefas import GerenciadorTarefas, Tarefa

class TestGerenciadorTarefas(unittest.TestCase):

    def setUp(self):
        # Configuração para cada teste
        # Usar um arquivo temporário para os testes
        self.arquivo_teste = "tarefas_teste.json"

        # Garantir que o arquivo de teste não existe
        if os.path.exists(self.arquivo_teste):
            os.remove(self.arquivo_teste)

        # Criar um gerenciador com o arquivo de teste
        self.gerenciador = GerenciadorTarefas(self.arquivo_teste)

    def tearDown(self):
        # Limpeza após cada teste
        if os.path.exists(self.arquivo_teste):
            os.remove(self.arquivo_teste)

    def test_adicionar_tarefa(self):
        """Testa a adição de uma tarefa."""
        tarefa = self.gerenciador.adicionar_tarefa("Teste", "Descrição de teste")

        # Verificar se a tarefa foi adicionada corretamente
        self.assertEqual(tarefa.titulo, "Teste")
        self.assertEqual(tarefa.descricao, "Descrição de teste")
        self.assertEqual(tarefa.id, 1)
        self.assertFalse(tarefa.concluida)

        # Verificar se a tarefa está na lista
        self.assertEqual(len(self.gerenciador.tarefas), 1)
        self.assertEqual(self.gerenciador.tarefas[0].titulo, "Teste")
```

```
    def test_remover_tarefa(self):
        """Testa a remoção de uma tarefa."""
        # Adicionar uma tarefa para depois remover
        tarefa = self.gerenciador.adicionar_tarefa("Tarefa para remover")

        # Verificar se a tarefa foi adicionada
        self.assertEqual(len(self.gerenciador.tarefas), 1)

        # Remover a tarefa
        resultado = self.gerenciador.remover_tarefa(tarefa.id)

        # Verificar se a remoção foi bem-sucedida
        self.assertTrue(resultado)
        self.assertEqual(len(self.gerenciador.tarefas), 0)

    def test_remover_tarefa_inexistente(self):
        """Testa a remoção de uma tarefa que não existe."""
        resultado = self.gerenciador.remover_tarefa(999)
        self.assertFalse(resultado)

    def test_marcar_como_concluida(self):
        """Testa marcar uma tarefa como concluída."""
        tarefa = self.gerenciador.adicionar_tarefa("Tarefa para concluir")

        # Verificar que a tarefa começa como não concluída
        self.assertFalse(tarefa.concluida)

        # Marcar como concluída
        resultado = self.gerenciador.marcar_como_concluida(tarefa.id)

        # Verificar se a operação foi bem-sucedida
        self.assertTrue(resultado)

        # Verificar se a tarefa foi marcada como concluída
        tarefa_atualizada = self.gerenciador.obter_tarefa(tarefa.id)
        self.assertTrue(tarefa_atualizada.concluida)
        self.assertIsNotNone(tarefa_atualizada.data_conclusao)

    def test_desmarcar_como_concluida(self):
        """Testa desmarcar uma tarefa como concluída."""
        tarefa = self.gerenciador.adicionar_tarefa("Tarefa para desmarcar")

        # Primeiro marcar como concluída
        self.gerenciador.marcar_como_concluida(tarefa.id)

        # Verificar que a tarefa está concluída
        tarefa_concluida = self.gerenciador.obter_tarefa(tarefa.id)
        self.assertTrue(tarefa_concluida.concluida)

        # Desmarcar como concluída
        resultado = self.gerenciador.desmarcar_como_concluida(tarefa.id)

        # Verificar se a operação foi bem-sucedida
```

```
        self.assertTrue(resultado)

        # Verificar se a tarefa foi desmarcada
        tarefa_atualizada = self.gerenciador.obter_tarefa(tarefa.id)
        self.assertFalse(tarefa_atualizada.concluida)
        self.assertIsNone(tarefa_atualizada.data_conclusao)

    def test_listar_tarefas(self):
        """Testa listar todas as tarefas."""
        # Adicionar algumas tarefas
        self.gerenciador.adicionar_tarefa("Tarefa 1")
        self.gerenciador.adicionar_tarefa("Tarefa 2")
        self.gerenciador.adicionar_tarefa("Tarefa 3")

        # Listar todas as tarefas
        tarefas = self.gerenciador.listar_tarefas()

        # Verificar se todas as tarefas foram listadas
        self.assertEqual(len(tarefas), 3)
        self.assertEqual(tarefas[0].titulo, "Tarefa 1")
        self.assertEqual(tarefas[1].titulo, "Tarefa 2")
        self.assertEqual(tarefas[2].titulo, "Tarefa 3")

    def test_listar_tarefas_concluidas(self):
        """Testa listar apenas tarefas concluídas."""
        # Adicionar tarefas
        t1 = self.gerenciador.adicionar_tarefa("Tarefa 1")
        t2 = self.gerenciador.adicionar_tarefa("Tarefa 2")
        t3 = self.gerenciador.adicionar_tarefa("Tarefa 3")

        # Marcar algumas como concluídas
        self.gerenciador.marcar_como_concluida(t1.id)
        self.gerenciador.marcar_como_concluida(t3.id)

        # Listar tarefas concluídas
        concluidas = self.gerenciador.listar_tarefas(apenas_concluidas=True)

        # Verificar se apenas as tarefas concluídas foram listadas
        self.assertEqual(len(concluidas), 2)
        self.assertEqual(concluidas[0].titulo, "Tarefa 1")
        self.assertEqual(concluidas[1].titulo, "Tarefa 3")

    def test_listar_tarefas_pendentes(self):
        """Testa listar apenas tarefas pendentes."""
        # Adicionar tarefas
        t1 = self.gerenciador.adicionar_tarefa("Tarefa 1")
        t2 = self.gerenciador.adicionar_tarefa("Tarefa 2")
        t3 = self.gerenciador.adicionar_tarefa("Tarefa 3")

        # Marcar algumas como concluídas
        self.gerenciador.marcar_como_concluida(t1.id)
        self.gerenciador.marcar_como_concluida(t3.id)
```

```
        # Listar tarefas pendentes
        pendentes = self.gerenciador.listar_tarefas(apenas_concluidas=False)

        # Verificar se apenas as tarefas pendentes foram listadas
        self.assertEqual(len(pendentes), 1)
        self.assertEqual(pendentes[0].titulo, "Tarefa 2")

    @patch('gerenciador_tarefas.open', new_callable=mock_open, read_data='[]')
    @patch('gerenciador_tarefas.os.path.exists', return_value=True)
    def test_carregar_tarefas(self, mock_exists, mock_file):
        """Testa carregar tarefas do arquivo."""
        gerenciador = GerenciadorTarefas("arquivo_mock.json")
        self.assertEqual(len(gerenciador.tarefas), 0)

    @patch('gerenciador_tarefas.open', new_callable=mock_open)
    @patch('gerenciador_tarefas.json.dump')
    def test_salvar_tarefas(self, mock_json_dump, mock_file):
        """Testa salvar tarefas no arquivo."""
        # Adicionar uma tarefa
        self.gerenciador.adicionar_tarefa("Tarefa de teste")

        # Verificar se o método json.dump foi chamado
        mock_json_dump.assert_called_once()

        # Verificar se o arquivo foi aberto para escrita
        mock_file.assert_called_with(self.arquivo_teste, "w")

if __name__ == '__main__':
    unittest.main()
```

Exercício 7: Projeto Integrado

Crie um sistema de análise de dados de vendas que utilize todos os conceitos avançados aprendidos.

Requisitos: 1. Crie um gerador que leia dados de vendas de um arquivo CSV linha por linha. 2. Use expressões regulares para validar e limpar os dados. 3. Implemente funções de ordem superior para analisar os dados (filtrar, mapear, reduzir). 4. Use decoradores para medir o tempo de execução e registrar operações. 5. Implemente processamento paralelo para analisar grandes conjuntos de dados. 6. Escreva testes unitários para as principais funcionalidades.

Solução:

```
# analisador_vendas.py
import csv
import re
import time
import functools
import concurrent.futures
from datetime import datetime
import os

# Decoradores
def registrar_tempo(funcao):
    """Decorador que registra o tempo de execução de uma função."""
    @functools.wraps(funcao)
```

```
    def wrapper(*args, **kwargs):
        inicio = time.time()
        resultado = funcao(*args, **kwargs)
        fim = time.time()
        print(f"A função {funcao.__name__} levou {fim - inicio:.4f} segundos para executar.")
        return resultado
    return wrapper

def registrar_operacao(funcao):
    """Decorador que registra a execução de uma operação."""
    @functools.wraps(funcao)
    def wrapper(*args, **kwargs):
        print(f"Executando {funcao.__name__} em {datetime.now().isoformat()}")
        return funcao(*args, **kwargs)
    return wrapper

# Gerador para ler dados do CSV
def ler_dados_vendas(arquivo_csv):
    """Gerador que lê dados de vendas de um arquivo CSV linha por linha."""
    with open(arquivo_csv, 'r', newline='') as arquivo:
        leitor = csv.DictReader(arquivo)
        for linha in leitor:
            yield linha

# Funções de validação com expressões regulares
def validar_data(data):
    """Valida e formata uma data no formato DD/MM/AAAA."""
    padrao = r'^(\d{2})/(\d{2})/(\d{4})$'
    match = re.match(padrao, data)
    if match:
        dia, mes, ano = match.groups()
        # Verificar se a data é válida
        try:
            datetime(int(ano), int(mes), int(dia))
            return True
        except ValueError:
            return False
    return False

def validar_valor(valor):
    """Valida e formata um valor monetário."""
    # Remover espaços e substituir vírgula por ponto
    valor_limpo = valor.strip().replace(',', '.')
    # Verificar se é um número válido
    padrao = r'^\d+(\.\d{1,2})?$'
    return bool(re.match(padrao, valor_limpo))

def limpar_valor(valor):
    """Limpa e converte um valor monetário para float."""
    valor_limpo = valor.strip().replace(',', '.')
    try:
        return float(valor_limpo)
    except ValueError:
```

```
        return 0.0

# Funções de análise
@registrar_tempo
@registrar_operacao
def calcular_total_vendas(dados):
    """Calcula o total de vendas."""
    return sum(limpar_valor(venda['valor']) for venda in dados)

@registrar_tempo
@registrar_operacao
def calcular_vendas_por_categoria(dados):
    """Calcula o total de vendas por categoria."""
    vendas_por_categoria = {}
    for venda in dados:
        categoria = venda['categoria']
        valor = limpar_valor(venda['valor'])

        if categoria in vendas_por_categoria:
            vendas_por_categoria[categoria] += valor
        else:
            vendas_por_categoria[categoria] = valor

    return vendas_por_categoria

@registrar_tempo
@registrar_operacao
def encontrar_maior_venda(dados):
    """Encontra a maior venda."""
    if not dados:
        return None

    return max(dados, key=lambda venda: limpar_valor(venda['valor']))

@registrar_tempo
@registrar_operacao
def filtrar_vendas_por_data(dados, data_inicio, data_fim):
    """Filtra vendas por intervalo de data."""
    def converter_data(data_str):
        """Converte string de data para objeto datetime."""
        dia, mes, ano = map(int, data_str.split('/'))
        return datetime(ano, mes, dia)

    data_inicio_obj = converter_data(data_inicio)
    data_fim_obj = converter_data(data_fim)

    return [
        venda for venda in dados
        if validar_data(venda['data']) and
        data_inicio_obj <= converter_data(venda['data']) <= data_fim_obj
    ]

# Processamento paralelo
```

```python
@registrar_tempo
def analisar_vendas_paralelo(arquivo_csv, num_workers=4):
    """Analisa dados de vendas usando processamento paralelo."""
    # Ler todos os dados primeiro
    dados = list(ler_dados_vendas(arquivo_csv))

    # Dividir os dados em chunks para processamento paralelo
    tamanho_chunk = len(dados) // num_workers
    chunks = [dados[i:i+tamanho_chunk] for i in range(0, len(dados), tamanho_chunk)]

    # Função para processar um chunk de dados
    def processar_chunk(chunk):
        total = sum(limpar_valor(venda['valor']) for venda in chunk)
        maior = max(chunk, key=lambda venda: limpar_valor(venda['valor']))
        return {
            'total': total,
            'maior_venda': maior
        }

    # Processar chunks em paralelo
    resultados = []
    with concurrent.futures.ProcessPoolExecutor(max_workers=num_workers) as executor:
        resultados = list(executor.map(processar_chunk, chunks))

    # Combinar resultados
    total_geral = sum(r['total'] for r in resultados)
    maior_venda = max(resultados, key=lambda r: limpar_valor(r['maior_venda']['valor']))['maior_venda']

    return {
        'total_geral': total_geral,
        'maior_venda': maior_venda
    }

# Função principal
@registrar_tempo
def analisar_dados_vendas(arquivo_csv):
    """Função principal para analisar dados de vendas."""
    print(f"Iniciando análise do arquivo: {arquivo_csv}")

    # Verificar se o arquivo existe
    if not os.path.exists(arquivo_csv):
        print(f"Erro: O arquivo {arquivo_csv} não existe.")
        return

    # Ler e validar dados
    dados_validos = []
    dados_invalidos = []

    for venda in ler_dados_vendas(arquivo_csv):
        if validar_data(venda.get('data', '')) and validar_valor(venda.get('valor', '')):
            dados_validos.append(venda)
        else:
            dados_invalidos.append(venda)
```

```
    print(f"Total de registros: {len(dados_validos) + len(dados_invalidos)}")
    print(f"Registros válidos: {len(dados_validos)}")
    print(f"Registros inválidos: {len(dados_invalidos)}")

    if not dados_validos:
        print("Nenhum dado válido para analisar.")
        return

    # Análises básicas
    total_vendas = calcular_total_vendas(dados_validos)
    print(f"Total de vendas: R$ {total_vendas:.2f}")

    vendas_por_categoria = calcular_vendas_por_categoria(dados_validos)
    print("\nVendas por categoria:")
    for categoria, valor in sorted(vendas_por_categoria.items(), key=lambda x: x[1], reverse=True):
        print(f"- {categoria}: R$ {valor:.2f}")

    maior_venda = encontrar_maior_venda(dados_validos)
    print(f"\nMaior venda: {maior_venda['produto']} - R$ {limpar_valor(maior_venda['valor']):.2f} ({mai

    # Análise por período (último mês)
    ultimo_mes = filtrar_vendas_por_data(dados_validos, "01/01/2023", "31/01/2023")
    print(f"\nVendas no último mês: {len(ultimo_mes)}")
    print(f"Total de vendas no último mês: R$ {calcular_total_vendas(ultimo_mes):.2f}")

    # Análise paralela para grandes conjuntos de dados
    print("\nExecutando análise paralela...")
    resultado_paralelo = analisar_vendas_paralelo(arquivo_csv)
    print(f"Resultado da análise paralela:")
    print(f"- Total geral: R$ {resultado_paralelo['total_geral']:.2f}")
    print(f"- Maior venda: {resultado_paralelo['maior_venda']['produto']} - R$ {limpar_valor(resultado_

    print("\nAnálise concluída com sucesso!")
    return {
        'total_vendas': total_vendas,
        'vendas_por_categoria': vendas_por_categoria,
        'maior_venda': maior_venda,
        'vendas_ultimo_mes': {
            'quantidade': len(ultimo_mes),
            'total': calcular_total_vendas(ultimo_mes)
        }
    }

# Executar a análise se o script for executado diretamente
if __name__ == "__main__":
    # Criar um arquivo CSV de exemplo se não existir
    arquivo_exemplo = "vendas_exemplo.csv"

    if not os.path.exists(arquivo_exemplo):
        print(f"Criando arquivo de exemplo: {arquivo_exemplo}")
        with open(arquivo_exemplo, 'w', newline='') as arquivo:
            escritor = csv.writer(arquivo)
            escritor.writerow(['data', 'produto', 'categoria', 'valor'])
```

```
            escritor.writerow(['15/01/2023', 'Notebook', 'Eletrônicos', '3500,00'])
            escritor.writerow(['16/01/2023', 'Smartphone', 'Eletrônicos', '1200,00'])
            escritor.writerow(['17/01/2023', 'Tablet', 'Eletrônicos', '800,00'])
            escritor.writerow(['18/01/2023', 'Monitor', 'Informática', '950,00'])
            escritor.writerow(['19/01/2023', 'Teclado', 'Informática', '120,00'])
            escritor.writerow(['20/01/2023', 'Mouse', 'Informática', '80,00'])
            escritor.writerow(['21/01/2023', 'Headphone', 'Áudio', '250,00'])
            escritor.writerow(['22/01/2023', 'Caixa de Som', 'Áudio', '180,00'])
            escritor.writerow(['23/01/2023', 'Câmera', 'Fotografia', '1500,00'])
            escritor.writerow(['24/01/2023', 'Tripé', 'Fotografia', '120,00'])
            # Adicionar alguns dados inválidos para teste
            escritor.writerow(['32/01/2023', 'Produto Inválido', 'Categoria', '100,00'])  # Data inváli
            escritor.writerow(['25/01/2023', 'Produto com Valor Inválido', 'Categoria', 'abc'])  # Valo

    # Executar a análise
    analisar_dados_vendas(arquivo_exemplo)
```

```
# test_analisador_vendas.py
import unittest
from unittest.mock import patch, mock_open
import csv
import io
from analisador_vendas import (
    validar_data, validar_valor, limpar_valor,
    calcular_total_vendas, calcular_vendas_por_categoria,
    encontrar_maior_venda, filtrar_vendas_por_data
)

class TestAnalisadorVendas(unittest.TestCase):

    def setUp(self):
        # Dados de teste
        self.dados_teste = [
            {'data': '15/01/2023', 'produto': 'Notebook', 'categoria': 'Eletrônicos', 'valor': '3500,00
            {'data': '16/01/2023', 'produto': 'Smartphone', 'categoria': 'Eletrônicos', 'valor': '1200,
            {'data': '17/01/2023', 'produto': 'Tablet', 'categoria': 'Eletrônicos', 'valor': '800,00'},
            {'data': '18/01/2023', 'produto': 'Monitor', 'categoria': 'Informática', 'valor': '950,00'}
            {'data': '19/01/2023', 'produto': 'Teclado', 'categoria': 'Informática', 'valor': '120,00'}
        ]

    def test_validar_data(self):
        """Testa a validação de datas."""
        self.assertTrue(validar_data('15/01/2023'))
        self.assertTrue(validar_data('29/02/2020'))  # Ano bissexto
        self.assertFalse(validar_data('32/01/2023'))  # Dia inválido
        self.assertFalse(validar_data('15/13/2023'))  # Mês inválido
        self.assertFalse(validar_data('29/02/2023'))  # 2023 não é bissexto
        self.assertFalse(validar_data('15-01-2023'))  # Formato inválido
        self.assertFalse(validar_data('2023/01/15'))  # Formato inválido

    def test_validar_valor(self):
        """Testa a validação de valores monetários."""
        self.assertTrue(validar_valor('3500,00'))
        self.assertTrue(validar_valor('3500.00'))
```

```
        self.assertTrue(validar_valor('3500'))
        self.assertTrue(validar_valor('0.5'))
        self.assertFalse(validar_valor('abc'))
        self.assertFalse(validar_valor('3,500.00'))  # Formato inválido
        self.assertFalse(validar_valor('-100'))  # Valor negativo

    def test_limpar_valor(self):
        """Testa a limpeza e conversão de valores monetários."""
        self.assertEqual(limpar_valor('3500,00'), 3500.00)
        self.assertEqual(limpar_valor('3500.00'), 3500.00)
        self.assertEqual(limpar_valor('3500'), 3500.00)
        self.assertEqual(limpar_valor('0.5'), 0.5)
        self.assertEqual(limpar_valor('abc'), 0.0)  # Valor inválido retorna 0

    def test_calcular_total_vendas(self):
        """Testa o cálculo do total de vendas."""
        total = calcular_total_vendas(self.dados_teste)
        self.assertEqual(total, 6570.00)

    def test_calcular_vendas_por_categoria(self):
        """Testa o cálculo de vendas por categoria."""
        vendas_por_categoria = calcular_vendas_por_categoria(self.dados_teste)
        self.assertEqual(vendas_por_categoria['Eletrônicos'], 5500.00)
        self.assertEqual(vendas_por_categoria['Informática'], 1070.00)

    def test_encontrar_maior_venda(self):
        """Testa encontrar a maior venda."""
        maior_venda = encontrar_maior_venda(self.dados_teste)
        self.assertEqual(maior_venda['produto'], 'Notebook')
        self.assertEqual(limpar_valor(maior_venda['valor']), 3500.00)

    def test_filtrar_vendas_por_data(self):
        """Testa filtrar vendas por intervalo de data."""
        # Filtrar vendas de 16/01/2023 a 18/01/2023
        vendas_filtradas = filtrar_vendas_por_data(self.dados_teste, '16/01/2023', '18/01/2023')
        self.assertEqual(len(vendas_filtradas), 3)
        self.assertEqual(vendas_filtradas[0]['produto'], 'Smartphone')
        self.assertEqual(vendas_filtradas[2]['produto'], 'Monitor')

        # Filtrar vendas de um único dia
        vendas_dia = filtrar_vendas_por_data(self.dados_teste, '15/01/2023', '15/01/2023')
        self.assertEqual(len(vendas_dia), 1)
        self.assertEqual(vendas_dia[0]['produto'], 'Notebook')

        # Filtrar vendas de um período sem vendas
        vendas_vazias = filtrar_vendas_por_data(self.dados_teste, '01/01/2023', '10/01/2023')
        self.assertEqual(len(vendas_vazias), 0)

if __name__ == '__main__':
    unittest.main()
```

Conclusão

Estes exercícios práticos foram projetados para consolidar os conhecimentos avançados de Python que adquirimos neste módulo. Eles cobrem uma ampla gama de tópicos, desde decoradores e geradores até concorrência e testes unitários, permitindo que você aplique esses conceitos em cenários realistas.

Ao resolver estes exercícios, você não apenas reforça sua compreensão dos conceitos individuais, mas também aprende a combiná-los para criar soluções mais poderosas e elegantes. A prática constante é essencial para se tornar proficiente em programação avançada em Python.

No próximo módulo, exploraremos a aplicação de Python na ciência de dados e machine learning, onde muitos desses conceitos avançados serão úteis para processar e analisar grandes conjuntos de dados de forma eficiente.

Módulo 4: Ciência de Dados e Machine Learning

Módulo 4: Introdução à Ciência de Dados e Machine Learning

Bibliotecas Fundamentais: NumPy e Pandas

A ciência de dados e o machine learning em Python são amplamente baseados em algumas bibliotecas fundamentais que fornecem as estruturas de dados e funções necessárias para manipular, analisar e visualizar dados de forma eficiente. Neste tópico, vamos explorar duas das bibliotecas mais importantes: NumPy e Pandas.

NumPy: Computação Numérica em Python

NumPy (Numerical Python) é a biblioteca fundamental para computação científica em Python. Ela fornece suporte para arrays multidimensionais, funções matemáticas de alto nível e ferramentas para integração com código C/C++ e Fortran.

Por que usar NumPy?

- **Eficiência**: Os arrays NumPy são mais rápidos e usam menos memória que as listas Python padrão.
- **Funcionalidade**: Oferece funções matemáticas avançadas sem precisar escrever loops.
- **Interoperabilidade**: É a base para muitas outras bibliotecas científicas em Python.

Instalação do NumPy

```
pip install numpy
```

Arrays NumPy

O objeto principal do NumPy é o array multidimensional, chamado `ndarray`. Diferente das listas Python, os arrays NumPy têm tamanho fixo e contêm elementos do mesmo tipo.

```
import numpy as np

# Criando arrays
array_1d = np.array([1, 2, 3, 4, 5])
array_2d = np.array([[1, 2, 3], [4, 5, 6]])

print(array_1d)  # [1 2 3 4 5]
print(array_2d)  # [[1 2 3]
                 #  [4 5 6]]

# Verificando propriedades
print(array_1d.shape)  # (5,)
```

```
print(array_2d.shape)  # (2, 3)
print(array_1d.dtype)  # int64
print(array_2d.ndim)   # 2 (dimensões)
```

Criando Arrays Especiais

NumPy oferece várias funções para criar arrays com padrões específicos:

```
# Array de zeros
zeros = np.zeros((3, 4))
print(zeros)
# [[0. 0. 0. 0.]
#  [0. 0. 0. 0.]
#  [0. 0. 0. 0.]]

# Array de uns
ones = np.ones((2, 3))
print(ones)
# [[1. 1. 1.]
#  [1. 1. 1.]]

# Array com valor constante
full = np.full((2, 2), 7)
print(full)
# [[7 7]
#  [7 7]]

# Matriz identidade
identity = np.eye(3)
print(identity)
# [[1. 0. 0.]
#  [0. 1. 0.]
#  [0. 0. 1.]]

# Array com valores espaçados uniformemente
linear = np.linspace(0, 10, 5)  # 5 valores entre 0 e 10
print(linear)
# [ 0.   2.5  5.   7.5 10. ]

# Array com valores em progressão aritmética
arange = np.arange(0, 10, 2)  # De 0 a 10 (exclusivo) com passo 2
print(arange)
# [0 2 4 6 8]

# Array com valores aleatórios
random = np.random.random((2, 3))
print(random)
# [[0.12345678 0.23456789 0.34567891]
#  [0.45678912 0.56789123 0.67891234]]
```

Indexação e Fatiamento

Os arrays NumPy podem ser indexados e fatiados de forma semelhante às listas Python, mas com recursos adicionais:

```python
array = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

# Acessando elementos
print(array[0, 0])   # 1 (primeira linha, primeira coluna)
print(array[2, 3])   # 12 (terceira linha, quarta coluna)

# Fatiamento
print(array[0:2, 1:3])
# [[2 3]
#  [6 7]]

# Usando listas para indexação
print(array[[0, 2], [1, 3]])  # [2 12] (elementos nas posições (0,1) e (2,3))

# Indexação booleana
mask = array > 5
print(mask)
# [[False False False False]
#  [False  True  True  True]
#  [ True  True  True  True]]
print(array[mask])  # [6 7 8 9 10 11 12]
```

Operações com Arrays

NumPy permite realizar operações vetorizadas em arrays, o que é muito mais eficiente que usar loops:

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

# Operações aritméticas
print(a + b)      # [5 7 9]
print(a - b)      # [-3 -3 -3]
print(a * b)      # [4 10 18]
print(a / b)      # [0.25 0.4 0.5]
print(a ** 2)     # [1 4 9]

# Operações com broadcasting
print(a + 2)      # [3 4 5]
print(a * 3)      # [3 6 9]

# Funções matemáticas
print(np.sqrt(a))  # [1. 1.41421356 1.73205081]
print(np.exp(a))   # [ 2.71828183  7.3890561  20.08553692]
print(np.sin(a))   # [0.84147098 0.90929743 0.14112001]

# Estatísticas
print(np.mean(a))      # 2.0
print(np.median(a))    # 2.0
print(np.std(a))       # 0.816496580927726
print(np.min(a))       # 1
print(np.max(a))       # 3
```

Reshape e Transposição

Podemos alterar a forma de um array ou transpô-lo:

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

# Reshape
reshaped = array.reshape(3, 4)
print(reshaped)
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]

# Transposição
transposed = reshaped.T
print(transposed)
# [[ 0  4  8]
#  [ 1  5  9]
#  [ 2  6 10]
#  [ 3  7 11]]

# Flatten (achatar)
flattened = reshaped.flatten()
print(flattened)  # [ 0  1  2  3  4  5  6  7  8  9 10 11]
```

Concatenação e Divisão

Podemos combinar arrays ou dividi-los:

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

# Concatenação
horizontal = np.hstack((a, b))
print(horizontal)
# [[1 2 5 6]
#  [3 4 7 8]]

vertical = np.vstack((a, b))
print(vertical)
# [[1 2]
#  [3 4]
#  [5 6]
#  [7 8]]

# Divisão
array = np.arange(16).reshape(4, 4)
print(array)
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]
#  [12 13 14 15]]

# Dividir horizontalmente
```

```
h_splits = np.hsplit(array, 2)
print(h_splits[0])
# [[ 0  1]
#  [ 4  5]
#  [ 8  9]
#  [12 13]]

# Dividir verticalmente
v_splits = np.vsplit(array, 2)
print(v_splits[0])
# [[0 1 2 3]
#  [4 5 6 7]]
```

Álgebra Linear

NumPy fornece funções para operações de álgebra linear:

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

# Produto matricial
print(np.dot(a, b))
# [[19 22]
#  [43 50]]

# Ou usando o operador @
print(a @ b)
# [[19 22]
#  [43 50]]

# Determinante
print(np.linalg.det(a))  # -2.0

# Inversa
print(np.linalg.inv(a))
# [[-2.   1. ]
#  [ 1.5 -0.5]]

# Autovalores e autovetores
eigenvalues, eigenvectors = np.linalg.eig(a)
print(eigenvalues)      # [5.37228132 -0.37228132]
print(eigenvectors)
# [[ 0.41597356  0.82456484]
#  [ 0.90937671 -0.56576746]]

# Resolução de sistemas lineares
c = np.array([7, 10])
x = np.linalg.solve(a, c)
print(x)  # [3. 2.]
```

Funções Universais (ufuncs)

As ufuncs são funções que operam elemento a elemento em arrays NumPy:

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

# Funções matemáticas
print(np.sqrt(a))    # [1.         1.41421356 1.73205081 2.        ]
print(np.log(a))     # [0.         0.69314718 1.09861229 1.38629436]
print(np.sin(a))     # [0.84147098 0.90929743 0.14112001 -0.7568025]

# Funções de arredondamento
print(np.ceil(np.array([1.1, 2.5, 3.9])))   # [2. 3. 4.]
print(np.floor(np.array([1.1, 2.5, 3.9])))  # [1. 2. 3.]
print(np.round(np.array([1.1, 2.5, 3.9])))  # [1. 2. 4.]
```

Exemplo Prático: Análise de Dados com NumPy

Vamos usar NumPy para analisar um conjunto de dados simples:

```python
# Dados de temperatura diária (°C) para uma semana
temperaturas = np.array([
    [22, 24, 23, 26, 25, 21, 20],  # Cidade A
    [18, 19, 17, 20, 21, 19, 18],  # Cidade B
    [30, 31, 29, 32, 33, 31, 30]   # Cidade C
])

# Estatísticas básicas
print("Média por cidade:")
print(np.mean(temperaturas, axis=1))  # [23. 18.86 30.86]

print("\nMédia por dia:")
print(np.mean(temperaturas, axis=0))  # [23.33 24.67 23.   26.   26.33 23.67 22.67]

print("\nTemperatura máxima:", np.max(temperaturas))  # 33
print("Temperatura mínima:", np.min(temperaturas))    # 17

# Encontrar dias mais quentes que 25°C para cada cidade
dias_quentes = temperaturas > 25
print("\nDias quentes:")
print(dias_quentes)
# [[False False False  True False False False]
#  [False False False False False False False]
#  [ True  True  True  True  True  True  True]]

# Contar dias quentes por cidade
print("\nNúmero de dias quentes por cidade:")
print(np.sum(dias_quentes, axis=1))  # [1 0 7]

# Calcular a amplitude térmica (max - min) para cada cidade
print("\nAmplitude térmica por cidade:")
print(np.max(temperaturas, axis=1) - np.min(temperaturas, axis=1))  # [6 4 4]
```

Pandas: Análise e Manipulação de Dados

Pandas é uma biblioteca Python que fornece estruturas de dados flexíveis e ferramentas para análise de dados. Construída sobre NumPy, ela é essencial para tarefas de limpeza, transformação, agregação e visualização de dados.

Por que usar Pandas?

- **Estruturas de dados poderosas**: DataFrame e Series para manipulação eficiente de dados tabulares.
- **Manipulação de dados**: Funções para filtrar, transformar e agregar dados.
- **Tratamento de dados ausentes**: Ferramentas para lidar com valores nulos.
- **Entrada/saída de dados**: Suporte para leitura e escrita em diversos formatos (CSV, Excel, SQL, etc.).

Instalação do Pandas

```
pip install pandas
```

Estruturas de Dados do Pandas

Pandas tem duas estruturas de dados principais:

1. **Series**: Array unidimensional rotulado, semelhante a uma coluna em uma tabela.
2. **DataFrame**: Estrutura bidimensional rotulada, semelhante a uma tabela ou planilha.

```
import pandas as pd
import numpy as np

# Criando uma Series
s = pd.Series([1, 3, 5, np.nan, 6, 8])
print(s)
# 0    1.0
# 1    3.0
# 2    5.0
# 3    NaN
# 4    6.0
# 5    8.0
# dtype: float64

# Series com índices personalizados
s = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
print(s)
# a    1
# b    2
# c    3
# d    4
# dtype: int64

# Acessando elementos
print(s['a'])  # 1
print(s[['a', 'c']])
# a    1
# c    3
# dtype: int64

# Series a partir de dicionário
d = {'a': 1, 'b': 2, 'c': 3}
s = pd.Series(d)
print(s)
# a    1
```

```
# b    2
# c    3
# dtype: int64
```

Series

```
# Criando um DataFrame a partir de um dicionário
data = {
    'Nome': ['João', 'Maria', 'Pedro', 'Ana'],
    'Idade': [28, 34, 29, 42],
    'Cidade': ['São Paulo', 'Rio de Janeiro', 'Belo Horizonte', 'Brasília']
}

df = pd.DataFrame(data)
print(df)
#     Nome  Idade           Cidade
# 0   João     28        São Paulo
# 1  Maria     34  Rio de Janeiro
# 2  Pedro     29   Belo Horizonte
# 3    Ana     42         Brasília

# DataFrame com índices personalizados
df = pd.DataFrame(data, index=['p1', 'p2', 'p3', 'p4'])
print(df)
#      Nome  Idade           Cidade
# p1   João     28        São Paulo
# p2  Maria     34  Rio de Janeiro
# p3  Pedro     29   Belo Horizonte
# p4    Ana     42         Brasília

# Criando DataFrame a partir de array NumPy
array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
df = pd.DataFrame(array, columns=['A', 'B', 'C'])
print(df)
#    A  B  C
# 0  1  2  3
# 1  4  5  6
# 2  7  8  9
```

DataFrame

Visualização de Dados

```
# Visualizando as primeiras linhas
print(df.head(2))
#    A  B  C
# 0  1  2  3
# 1  4  5  6

# Visualizando as últimas linhas
print(df.tail(2))
#    A  B  C
# 1  4  5  6
```

```
# 2  7  8  9

# Informações sobre o DataFrame
print(df.info())
# <class 'pandas.core.frame.DataFrame'>
# RangeIndex: 3 entries, 0 to 2
# Data columns (total 3 columns):
#  #   Column  Non-Null Count  Dtype
# ---  ------  --------------  -----
#  0   A       3 non-null      int64
#  1   B       3 non-null      int64
#  2   C       3 non-null      int64
# dtypes: int64(3)
# memory usage: 200.0 bytes

# Estatísticas descritivas
print(df.describe())
#               A         B         C
# count  3.000000  3.000000  3.000000
# mean   4.000000  5.000000  6.000000
# std    3.000000  3.000000  3.000000
# min    1.000000  2.000000  3.000000
# 25%    2.500000  3.500000  4.500000
# 50%    4.000000  5.000000  6.000000
# 75%    5.500000  6.500000  7.500000
# max    7.000000  8.000000  9.000000
```

Seleção e Indexação

Pandas oferece várias maneiras de selecionar dados:

```
# Usando o exemplo anterior
data = {
    'Nome': ['João', 'Maria', 'Pedro', 'Ana'],
    'Idade': [28, 34, 29, 42],
    'Cidade': ['São Paulo', 'Rio de Janeiro', 'Belo Horizonte', 'Brasília'],
    'Salário': [5000, 6000, 4500, 7500]
}
df = pd.DataFrame(data)

# Selecionando uma coluna (retorna Series)
print(df['Nome'])
# 0     João
# 1    Maria
# 2    Pedro
# 3      Ana
# Name: Nome, dtype: object

# Selecionando múltiplas colunas (retorna DataFrame)
print(df[['Nome', 'Idade']])
#     Nome  Idade
# 0   João     28
# 1  Maria     34
# 2  Pedro     29
```

```
# 3    Ana     42

# Selecionando por posição com iloc
print(df.iloc[0])  # Primeira linha
# Nome        João
# Idade         28
# Cidade    São Paulo
# Salário     5000
# Name: 0, dtype: object

print(df.iloc[0:2, 1:3])  # Primeiras 2 linhas, colunas 1 e 2
#    Idade           Cidade
# 0     28        São Paulo
# 1     34  Rio de Janeiro

# Selecionando por rótulo com loc
print(df.loc[0, 'Nome'])  # 'João'
print(df.loc[0:2, ['Nome', 'Salário']])
#     Nome  Salário
# 0   João     5000
# 1  Maria     6000
# 2  Pedro     4500

# Filtragem booleana
print(df[df['Idade'] > 30])
#     Nome  Idade           Cidade  Salário
# 1  Maria     34  Rio de Janeiro     6000
# 3    Ana     42         Brasília     7500

# Filtragem com múltiplas condições
print(df[(df['Idade'] > 30) & (df['Salário'] > 6500)])
#   Nome  Idade    Cidade  Salário
# 3  Ana     42  Brasília     7500
```

Manipulação de Dados

```
# Adicionando uma nova coluna
df['Departamento'] = ['TI', 'Marketing', 'Vendas', 'Financeiro']
print(df)
#     Nome  Idade           Cidade  Salário Departamento
# 0   João     28        São Paulo     5000           TI
# 1  Maria     34  Rio de Janeiro     6000    Marketing
# 2  Pedro     29   Belo Horizonte     4500       Vendas
# 3    Ana     42         Brasília     7500   Financeiro

# Removendo uma coluna
df_sem_salario = df.drop('Salário', axis=1)
print(df_sem_salario)
#     Nome  Idade           Cidade Departamento
# 0   João     28        São Paulo           TI
# 1  Maria     34  Rio de Janeiro    Marketing
# 2  Pedro     29   Belo Horizonte       Vendas
# 3    Ana     42         Brasília   Financeiro
```

```
# Removendo múltiplas colunas
df_reduzido = df.drop(['Salário', 'Departamento'], axis=1)
print(df_reduzido)
#     Nome  Idade           Cidade
# 0   João     28        São Paulo
# 1  Maria     34  Rio de Janeiro
# 2  Pedro     29   Belo Horizonte
# 3    Ana     42         Brasília
```

Adicionando e Removendo Colunas

```
# Adicionando uma nova linha
nova_linha = pd.Series(['Carlos', 31, 'Porto Alegre', 5500, 'RH'],
                       index=df.columns)
df = df.append(nova_linha, ignore_index=True)
print(df)
#     Nome  Idade           Cidade  Salário Departamento
# 0   João     28        São Paulo     5000           TI
# 1  Maria     34  Rio de Janeiro     6000    Marketing
# 2  Pedro     29   Belo Horizonte     4500       Vendas
# 3    Ana     42         Brasília     7500   Financeiro
# 4 Carlos     31    Porto Alegre     5500           RH

# Removendo linhas
df_sem_linha = df.drop(0)  # Remove a primeira linha
print(df_sem_linha)
#     Nome  Idade           Cidade  Salário Departamento
# 1  Maria     34  Rio de Janeiro     6000    Marketing
# 2  Pedro     29   Belo Horizonte     4500       Vendas
# 3    Ana     42         Brasília     7500   Financeiro
# 4 Carlos     31    Porto Alegre     5500           RH
```

Adicionando e Removendo Linhas

```
# Criando um DataFrame com valores ausentes
df_na = pd.DataFrame({
    'A': [1, 2, np.nan, 4],
    'B': [5, np.nan, np.nan, 8],
    'C': [9, 10, 11, np.nan]
})
print(df_na)
#      A    B     C
# 0  1.0  5.0   9.0
# 1  2.0  NaN  10.0
# 2  NaN  NaN  11.0
# 3  4.0  8.0   NaN

# Verificando valores ausentes
print(df_na.isna())
#        A      B      C
# 0  False  False  False
```

```
# 1  False   True  False
# 2   True   True  False
# 3  False  False   True

print(df_na.isna().sum())
# A    1
# B    2
# C    1
# dtype: int64

# Preenchendo valores ausentes
print(df_na.fillna(0))  # Preenche com zero
#      A    B     C
# 0  1.0  5.0   9.0
# 1  2.0  0.0  10.0
# 2  0.0  0.0  11.0
# 3  4.0  8.0   0.0

print(df_na.fillna(method='ffill'))  # Preenche com o valor anterior
#      A    B     C
# 0  1.0  5.0   9.0
# 1  2.0  5.0  10.0
# 2  2.0  5.0  11.0
# 3  4.0  8.0  11.0

# Removendo linhas com valores ausentes
print(df_na.dropna())
#      A    B    C
# 0  1.0  5.0  9.0

# Removendo colunas com valores ausentes
print(df_na.dropna(axis=1))
# Empty DataFrame
# Columns: []
# Index: [0, 1, 2, 3]
```

Tratamento de Valores Ausentes

Operações com Dados

```
# Aplicando função a uma coluna
df['Idade_em_meses'] = df['Idade'].apply(lambda x: x * 12)
print(df)
#     Nome  Idade           Cidade  Salário Departamento  Idade_em_meses
# 0   João     28        São Paulo     5000           TI             336
# 1  Maria     34  Rio de Janeiro     6000    Marketing             408
# 2  Pedro     29   Belo Horizonte     4500       Vendas             348
# 3    Ana     42         Brasília     7500   Financeiro             504
# 4 Carlos     31    Porto Alegre     5500           RH             372

# Aplicando função a múltiplas colunas
df[['Idade', 'Salário']] = df[['Idade', 'Salário']].apply(pd.to_numeric)
print(df.dtypes)
```

```
# Nome              object
# Idade              int64
# Cidade            object
# Salário            int64
# Departamento      object
# Idade_em_meses     int64
# dtype: object

# Aplicando função a cada elemento
df_numerico = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6]
})
print(df_numerico.applymap(lambda x: x**2))
#    A   B
# 0  1  16
# 1  4  25
# 2  9  36
```

Aplicando Funções

```
# Criando um DataFrame para exemplo
data = {
    'Departamento': ['TI', 'Marketing', 'Vendas', 'TI', 'Marketing', 'Vendas', 'TI'],
    'Funcionário': ['João', 'Maria', 'Pedro', 'Ana', 'Carlos', 'Paula', 'Lucas'],
    'Salário': [5000, 6000, 4500, 7500, 5500, 4800, 6200]
}
df = pd.DataFrame(data)
print(df)
#   Departamento Funcionário  Salário
# 0           TI        João     5000
# 1    Marketing       Maria     6000
# 2       Vendas       Pedro     4500
# 3           TI         Ana     7500
# 4    Marketing      Carlos     5500
# 5       Vendas       Paula     4800
# 6           TI       Lucas     6200

# Agrupando por departamento e calculando estatísticas
grupo = df.groupby('Departamento')

# Média de salário por departamento
print(grupo['Salário'].mean())
# Departamento
# Marketing    5750.0
# TI           6233.3
# Vendas       4650.0
# Name: Salário, dtype: float64

# Múltiplas estatísticas
print(grupo['Salário'].agg(['mean', 'min', 'max', 'count']))
#              mean   min   max  count
# Departamento
```

```
# Marketing   5750.0  5500  6000      2
# TI          6233.3  5000  7500      3
# Vendas      4650.0  4500  4800      2

# Agrupando por múltiplas colunas
df['Sexo'] = ['M', 'F', 'M', 'F', 'M', 'F', 'M']
grupo_multi = df.groupby(['Departamento', 'Sexo'])
print(grupo_multi['Salário'].mean())
# Departamento  Sexo
# Marketing     F       6000.0
#               M       5500.0
# TI            F       7500.0
#               M       5600.0
# Vendas        F       4800.0
#               M       4500.0
# Name: Salário, dtype: float64
```

Agrupamento e Agregação

```
# Criando um DataFrame para exemplo
data = {
    'Data': ['2023-01-01', '2023-01-01', '2023-01-02', '2023-01-02', '2023-01-03', '2023-01-03'],
    'Produto': ['A', 'B', 'A', 'B', 'A', 'B'],
    'Região': ['Norte', 'Sul', 'Norte', 'Sul', 'Norte', 'Sul'],
    'Vendas': [100, 150, 120, 180, 90, 160]
}
df = pd.DataFrame(data)
print(df)
#          Data Produto Região  Vendas
# 0  2023-01-01       A  Norte     100
# 1  2023-01-01       B    Sul     150
# 2  2023-01-02       A  Norte     120
# 3  2023-01-02       B    Sul     180
# 4  2023-01-03       A  Norte      90
# 5  2023-01-03       B    Sul     160

# Tabela pivot
pivot = df.pivot(index='Data', columns='Produto', values='Vendas')
print(pivot)
# Produto        A      B
# Data
# 2023-01-01  100.0  150.0
# 2023-01-02  120.0  180.0
# 2023-01-03   90.0  160.0

# Tabela pivot com múltiplos valores
pivot_multi = pd.pivot_table(df, values='Vendas', index=['Data'],
                             columns=['Produto', 'Região'], aggfunc='sum')
print(pivot_multi)
# Produto        A        B
# Região     Norte     Sul
# Data
# 2023-01-01  100.0     NaN  150.0
```

```
# 2023-01-02  120.0     NaN  180.0
# 2023-01-03   90.0     NaN  160.0
```

Pivotagem

```
# Criando DataFrames para exemplo
df1 = pd.DataFrame({
    'ID': [1, 2, 3, 4],
    'Nome': ['João', 'Maria', 'Pedro', 'Ana'],
    'Departamento': ['TI', 'Marketing', 'Vendas', 'RH']
})

df2 = pd.DataFrame({
    'ID': [1, 2, 3, 5],
    'Salário': [5000, 6000, 4500, 7000],
    'Bônus': [1000, 1500, 800, 2000]
})

# Merge (similar a JOIN em SQL)
# Inner join (padrão)
merge_inner = pd.merge(df1, df2, on='ID')
print(merge_inner)
#    ID   Nome Departamento  Salário  Bônus
# 0   1   João           TI     5000   1000
# 1   2  Maria    Marketing     6000   1500
# 2   3  Pedro       Vendas     4500    800

# Left join
merge_left = pd.merge(df1, df2, on='ID', how='left')
print(merge_left)
#    ID   Nome Departamento  Salário  Bônus
# 0   1   João           TI   5000.0  1000.0
# 1   2  Maria    Marketing   6000.0  1500.0
# 2   3  Pedro       Vendas   4500.0   800.0
# 3   4    Ana           RH      NaN     NaN

# Right join
merge_right = pd.merge(df1, df2, on='ID', how='right')
print(merge_right)
#    ID   Nome Departamento  Salário  Bônus
# 0   1   João           TI     5000   1000
# 1   2  Maria    Marketing     6000   1500
# 2   3  Pedro       Vendas     4500    800
# 3   5    NaN          NaN     7000   2000

# Outer join
merge_outer = pd.merge(df1, df2, on='ID', how='outer')
print(merge_outer)
#    ID   Nome Departamento  Salário  Bônus
# 0   1   João           TI   5000.0  1000.0
# 1   2  Maria    Marketing   6000.0  1500.0
# 2   3  Pedro       Vendas   4500.0   800.0
# 3   4    Ana           RH      NaN     NaN
```

```
# 4   5    NaN          NaN   7000.0  2000.0

# Concatenação
df3 = pd.DataFrame({
    'ID': [6, 7],
    'Nome': ['Carlos', 'Paula'],
    'Departamento': ['TI', 'Marketing']
})

# Concatenar verticalmente (empilhar)
concat_v = pd.concat([df1, df3])
print(concat_v)
#    ID    Nome Departamento
# 0   1    João           TI
# 1   2   Maria    Marketing
# 2   3   Pedro       Vendas
# 3   4     Ana           RH
# 0   6  Carlos           TI
# 1   7   Paula    Marketing

# Concatenar horizontalmente (lado a lado)
df4 = pd.DataFrame({
    'Cidade': ['São Paulo', 'Rio de Janeiro', 'Belo Horizonte', 'Brasília'],
    'Estado': ['SP', 'RJ', 'MG', 'DF']
})

concat_h = pd.concat([df1, df4], axis=1)
print(concat_h)
#    ID   Nome Departamento           Cidade Estado
# 0   1   João           TI        São Paulo     SP
# 1   2  Maria    Marketing  Rio de Janeiro     RJ
# 2   3  Pedro       Vendas   Belo Horizonte     MG
# 3   4    Ana           RH         Brasília     DF
```

Mesclagem (Merge) e Concatenação

Entrada e Saída de Dados

Pandas suporta leitura e escrita em diversos formatos:

```
# CSV
df.to_csv('dados.csv', index=False)
df_lido = pd.read_csv('dados.csv')

# Excel
df.to_excel('dados.xlsx', sheet_name='Planilha1', index=False)
df_lido = pd.read_excel('dados.xlsx', sheet_name='Planilha1')

# JSON
df.to_json('dados.json', orient='records')
df_lido = pd.read_json('dados.json')

# SQL (requer SQLAlchemy)
from sqlalchemy import create_engine
```

```
engine = create_engine('sqlite:///banco.db')
df.to_sql('tabela', engine, if_exists='replace', index=False)
df_lido = pd.read_sql('SELECT * FROM tabela', engine)
```

Exemplo Prático: Análise de Dados com Pandas

Vamos analisar um conjunto de dados de vendas:

```
# Criando um DataFrame de vendas
data = {
    'Data': pd.date_range(start='2023-01-01', periods=10),
    'Produto': ['A', 'B', 'A', 'C', 'B', 'A', 'C', 'B', 'A', 'C'],
    'Quantidade': [10, 15, 8, 12, 20, 5, 10, 18, 7, 9],
    'Preço': [100, 150, 100, 200, 150, 100, 200, 150, 100, 200],
    'Vendedor': ['João', 'Maria', 'João', 'Pedro', 'Maria', 'João', 'Pedro', 'Maria', 'João', 'Pedro']
}

df_vendas = pd.DataFrame(data)
print(df_vendas)
#         Data Produto  Quantidade  Preço Vendedor
# 0 2023-01-01       A          10    100     João
# 1 2023-01-02       B          15    150    Maria
# 2 2023-01-03       A           8    100     João
# 3 2023-01-04       C          12    200    Pedro
# 4 2023-01-05       B          20    150    Maria
# 5 2023-01-06       A           5    100     João
# 6 2023-01-07       C          10    200    Pedro
# 7 2023-01-08       B          18    150    Maria
# 8 2023-01-09       A           7    100     João
# 9 2023-01-10       C           9    200    Pedro

# Adicionando coluna de valor total
df_vendas['Valor_Total'] = df_vendas['Quantidade'] * df_vendas['Preço']
print(df_vendas)
#         Data Produto  Quantidade  Preço Vendedor  Valor_Total
# 0 2023-01-01       A          10    100     João         1000
# 1 2023-01-02       B          15    150    Maria         2250
# 2 2023-01-03       A           8    100     João          800
# 3 2023-01-04       C          12    200    Pedro         2400
# 4 2023-01-05       B          20    150    Maria         3000
# 5 2023-01-06       A           5    100     João          500
# 6 2023-01-07       C          10    200    Pedro         2000
# 7 2023-01-08       B          18    150    Maria         2700
# 8 2023-01-09       A           7    100     João          700
# 9 2023-01-10       C           9    200    Pedro         1800

# Análise por produto
vendas_por_produto = df_vendas.groupby('Produto').agg({
    'Quantidade': 'sum',
    'Valor_Total': 'sum'
})
print("\nVendas por produto:")
print(vendas_por_produto)
#         Quantidade  Valor_Total
```

```
# Produto
# A               30         3000
# B               53         7950
# C               31         6200

# Análise por vendedor
vendas_por_vendedor = df_vendas.groupby('Vendedor').agg({
    'Quantidade': 'sum',
    'Valor_Total': ['sum', 'mean']
})
print("\nVendas por vendedor:")
print(vendas_por_vendedor)
#          Quantidade Valor_Total
#                 sum         sum   mean
# Vendedor
# João             30        3000  750.0
# Maria            53        7950 2650.0
# Pedro            31        6200 2066.7

# Vendas diárias
df_vendas['Data'] = pd.to_datetime(df_vendas['Data'])
df_vendas['Dia_Semana'] = df_vendas['Data'].dt.day_name()
vendas_por_dia = df_vendas.groupby('Dia_Semana')['Valor_Total'].sum()
print("\nVendas por dia da semana:")
print(vendas_por_dia)
# Dia_Semana
# Friday       3000
# Monday       1000
# Saturday     2000
# Sunday       2700
# Thursday     2400
# Tuesday      2250
# Wednesday     800
# Name: Valor_Total, dtype: int64

# Produto mais vendido por vendedor
produto_por_vendedor = df_vendas.groupby(['Vendedor', 'Produto'])['Quantidade'].sum().unstack()
print("\nQuantidade vendida por vendedor e produto:")
print(produto_por_vendedor)
# Produto       A     B     C
# Vendedor
# João        30.0   NaN   NaN
# Maria        NaN  53.0   NaN
# Pedro        NaN   NaN  31.0

# Evolução das vendas ao longo do tempo
vendas_diarias = df_vendas.groupby('Data')['Valor_Total'].sum()
print("\nEvolução das vendas:")
print(vendas_diarias)
# Data
# 2023-01-01    1000
# 2023-01-02    2250
# 2023-01-03     800
```

```
# 2023-01-04    2400
# 2023-01-05    3000
# 2023-01-06     500
# 2023-01-07    2000
# 2023-01-08    2700
# 2023-01-09     700
# 2023-01-10    1800
# Name: Valor_Total, dtype: int64
```

Integração NumPy e Pandas

NumPy e Pandas trabalham muito bem juntos, já que Pandas é construído sobre NumPy:

```
# Convertendo DataFrame para array NumPy
df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6]
})
array = df.to_numpy()
print(array)
# [[1 4]
#  [2 5]
#  [3 6]]

# Aplicando funções NumPy em DataFrames
df['C'] = np.sqrt(df['A'])
df['D'] = np.log(df['B'])
print(df)
#    A  B         C         D
# 0  1  4  1.000000  1.386294
# 1  2  5  1.414214  1.609438
# 2  3  6  1.732051  1.791759

# Operações estatísticas
print("Média:", np.mean(df['A']))
print("Desvio padrão:", np.std(df['B']))
print("Correlação:", np.corrcoef(df['A'], df['B'])[0, 1])
```

Conclusão

NumPy e Pandas são bibliotecas fundamentais para ciência de dados e machine learning em Python. NumPy fornece a base para computação numérica eficiente, enquanto Pandas oferece estruturas de dados flexíveis e ferramentas poderosas para manipulação e análise de dados.

Dominar essas bibliotecas é essencial para qualquer pessoa que deseje trabalhar com dados em Python, pois elas formam a base para bibliotecas mais avançadas de visualização, machine learning e deep learning que veremos nos próximos tópicos.

No próximo tópico, exploraremos as bibliotecas de visualização de dados Matplotlib e Seaborn, que nos permitirão criar gráficos e visualizações a partir dos dados que manipulamos com NumPy e Pandas.

Módulo 4: Introdução à Ciência de Dados e Machine Learning

Visualização de Dados: Matplotlib e Seaborn

A visualização de dados é uma parte fundamental da análise de dados e do machine learning. Ela nos permite compreender padrões, tendências e relações nos dados que seriam difíceis de identificar apenas olhando para números. Em Python, as bibliotecas Matplotlib e Seaborn são ferramentas poderosas para criar visualizações informativas e atraentes.

Matplotlib

Matplotlib é a biblioteca de visualização mais antiga e amplamente utilizada em Python. Ela fornece uma interface flexível para criar uma grande variedade de gráficos estáticos, interativos e animados.

Instalação do Matplotlib

```
pip install matplotlib
```

Estrutura do Matplotlib

Matplotlib é organizado em uma hierarquia de objetos:

- **Figure**: O contêiner de nível superior que pode conter múltiplos eixos (plots).
- **Axes**: O objeto que contém os elementos do gráfico (linhas, pontos, legendas, etc.).
- **Axis**: Os eixos x e y que compõem um gráfico.

Existem duas interfaces principais para usar o Matplotlib:

1. **Interface orientada a objetos**: Mais flexível e poderosa, ideal para gráficos complexos.
2. **Interface pyplot**: Mais simples e semelhante ao MATLAB, ideal para gráficos rápidos.

Gráficos Básicos com Matplotlib

Vamos começar com alguns exemplos básicos:

```
import matplotlib.pyplot as plt
import numpy as np

# Dados para o gráfico
x = np.linspace(0, 10, 100)  # 100 pontos entre 0 e 10
y = np.sin(x)

# Criando um gráfico de linha simples
```

```
plt.figure(figsize=(10, 6))  # Tamanho da figura em polegadas
plt.plot(x, y, 'b-', label='sen(x)')  # 'b-' significa linha azul contínua
plt.title('Gráfico da Função Seno')
plt.xlabel('x')
plt.ylabel('sen(x)')
plt.grid(True)
plt.legend()
plt.savefig('seno.png')  # Salvar o gráfico como imagem
plt.show()  # Exibir o gráfico
```

Múltiplos Gráficos

Podemos criar múltiplos gráficos na mesma figura:

```
# Criando múltiplos gráficos
plt.figure(figsize=(12, 6))

# Primeiro gráfico: seno
plt.subplot(1, 2, 1)  # 1 linha, 2 colunas, primeiro gráfico
plt.plot(x, np.sin(x), 'b-', label='sen(x)')
plt.title('Função Seno')
plt.xlabel('x')
plt.ylabel('sen(x)')
plt.grid(True)
plt.legend()

# Segundo gráfico: cosseno
plt.subplot(1, 2, 2)  # 1 linha, 2 colunas, segundo gráfico
plt.plot(x, np.cos(x), 'r-', label='cos(x)')
plt.title('Função Cosseno')
plt.xlabel('x')
plt.ylabel('cos(x)')
plt.grid(True)
plt.legend()

plt.tight_layout()  # Ajusta automaticamente o espaçamento entre os gráficos
plt.savefig('seno_cosseno.png')
plt.show()
```

Interface Orientada a Objetos

A interface orientada a objetos oferece mais controle sobre os gráficos:

```
# Usando a interface orientada a objetos
fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(x, np.sin(x), 'b-', label='sen(x)')
ax.plot(x, np.cos(x), 'r--', label='cos(x)')  # 'r--' significa linha vermelha tracejada
ax.set_title('Funções Seno e Cosseno')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.grid(True)
ax.legend()
```

```
plt.savefig('seno_cosseno_oo.png')
plt.show()
```

Tipos de Gráficos em Matplotlib

Matplotlib suporta uma ampla variedade de tipos de gráficos:

```
# Dados aleatórios
np.random.seed(42)  # Para reprodutibilidade
x = np.random.rand(50)
y = np.random.rand(50)
cores = np.random.rand(50)
tamanhos = 1000 * np.random.rand(50)

plt.figure(figsize=(10, 6))
plt.scatter(x, y, c=cores, s=tamanhos, alpha=0.5, cmap='viridis')
plt.colorbar(label='Valor da Cor')
plt.title('Gráfico de Dispersão')
plt.xlabel('Eixo X')
plt.ylabel('Eixo Y')
plt.grid(True)
plt.savefig('dispersao.png')
plt.show()
```

Gráfico de Dispersão (Scatter Plot)

```
# Dados para o gráfico de barras
categorias = ['A', 'B', 'C', 'D', 'E']
valores = [25, 40, 30, 55, 15]

plt.figure(figsize=(10, 6))
plt.bar(categorias, valores, color='skyblue')
plt.title('Gráfico de Barras')
plt.xlabel('Categorias')
plt.ylabel('Valores')
plt.grid(True, axis='y')
plt.savefig('barras.png')
plt.show()

# Gráfico de barras horizontais
plt.figure(figsize=(10, 6))
plt.barh(categorias, valores, color='salmon')
plt.title('Gráfico de Barras Horizontais')
plt.xlabel('Valores')
plt.ylabel('Categorias')
plt.grid(True, axis='x')
plt.savefig('barras_horizontais.png')
plt.show()
```

Gráfico de Barras

```
# Dados para o histograma
dados = np.random.randn(1000)  # 1000 pontos de uma distribuição normal

plt.figure(figsize=(10, 6))
plt.hist(dados, bins=30, color='skyblue', edgecolor='black', alpha=0.7)
plt.title('Histograma')
plt.xlabel('Valor')
plt.ylabel('Frequência')
plt.grid(True, axis='y')
plt.savefig('histograma.png')
plt.show()
```

Histograma

```
# Dados para o gráfico de pizza
categorias = ['A', 'B', 'C', 'D', 'E']
valores = [25, 40, 30, 55, 15]

plt.figure(figsize=(10, 6))
plt.pie(valores, labels=categorias, autopct='%1.1f%%', startangle=90,
        shadow=True, explode=(0, 0.1, 0, 0, 0))
plt.title('Gráfico de Pizza')
plt.axis('equal')  # Garante que o gráfico seja circular
plt.savefig('pizza.png')
plt.show()
```

Gráfico de Pizza

```
# Dados para o box plot
dados = [np.random.normal(0, std, 100) for std in range(1, 6)]

plt.figure(figsize=(10, 6))
plt.boxplot(dados, labels=['A', 'B', 'C', 'D', 'E'])
plt.title('Box Plot')
plt.xlabel('Categorias')
plt.ylabel('Valores')
plt.grid(True, axis='y')
plt.savefig('boxplot.png')
plt.show()
```

Gráfico de Caixa (Box Plot)

```
plt.figure(figsize=(10, 6))
plt.violinplot(dados)
plt.title('Violin Plot')
plt.xlabel('Categorias')
plt.ylabel('Valores')
plt.xticks(np.arange(1, 6), ['A', 'B', 'C', 'D', 'E'])
plt.grid(True, axis='y')
```

```
plt.savefig('violinplot.png')
plt.show()
```

Gráfico de Violino (Violin Plot)

```
from mpl_toolkits.mplot3d import Axes3D

# Dados para o gráfico 3D
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))

# Gráfico de contorno
plt.figure(figsize=(10, 6))
contour = plt.contourf(X, Y, Z, 20, cmap='viridis')
plt.colorbar(contour, label='z = sin(sqrt(x² + y²))')
plt.title('Gráfico de Contorno')
plt.xlabel('x')
plt.ylabel('y')
plt.grid(True)
plt.savefig('contorno.png')
plt.show()

# Gráfico de superfície 3D
fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection='3d')
surface = ax.plot_surface(X, Y, Z, cmap='viridis', edgecolor='none')
fig.colorbar(surface, ax=ax, shrink=0.5, aspect=5, label='z = sin(sqrt(x² + y²))')
ax.set_title('Gráfico de Superfície 3D')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
plt.savefig('superficie3d.png')
plt.show()
```

Gráfico de Contorno e Superfície 3D

Personalização de Gráficos

Matplotlib oferece muitas opções para personalizar gráficos:

```
# Personalização de gráficos
plt.figure(figsize=(12, 8))

# Dados
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Gráfico com personalização
plt.plot(x, y1, 'b-', linewidth=2, label='sen(x)')
plt.plot(x, y2, 'r--', linewidth=2, label='cos(x)')
```

```
# Título e rótulos
plt.title('Funções Trigonométricas', fontsize=18, fontweight='bold')
plt.xlabel('x', fontsize=14)
plt.ylabel('y', fontsize=14)

# Limites dos eixos
plt.xlim(0, 10)
plt.ylim(-1.5, 1.5)

# Grade
plt.grid(True, linestyle='--', alpha=0.7)

# Legenda
plt.legend(fontsize=12, loc='upper right')

# Anotações
plt.annotate('Ponto Máximo', xy=(1.57, 1), xytext=(3, 1.3),
             arrowprops=dict(facecolor='black', shrink=0.05))

# Marcadores nos eixos
plt.xticks(np.arange(0, 11, 2), fontsize=12)
plt.yticks(np.arange(-1.5, 1.6, 0.5), fontsize=12)

# Estilo de fundo
plt.style.use('seaborn-whitegrid')

plt.tight_layout()
plt.savefig('grafico_personalizado.png', dpi=300)
plt.show()
```

Estilos Predefinidos

Matplotlib vem com vários estilos predefinidos:

```
# Listar estilos disponíveis
print(plt.style.available)

# Usar um estilo específico
plt.style.use('ggplot')

plt.figure(figsize=(10, 6))
plt.plot(x, np.sin(x), label='sen(x)')
plt.plot(x, np.cos(x), label='cos(x)')
plt.title('Gráfico com Estilo ggplot')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.savefig('estilo_ggplot.png')
plt.show()
```

Integração com Pandas

Matplotlib integra-se perfeitamente com Pandas:

```
import pandas as pd

# Criar um DataFrame
df = pd.DataFrame({
    'x': np.linspace(0, 10, 100),
    'y1': np.sin(np.linspace(0, 10, 100)),
    'y2': np.cos(np.linspace(0, 10, 100))
})

# Plotar diretamente do DataFrame
plt.figure(figsize=(10, 6))
df.plot(x='x', y=['y1', 'y2'], figsize=(10, 6))
plt.title('Gráfico a partir de DataFrame')
plt.xlabel('x')
plt.ylabel('y')
plt.grid(True)
plt.savefig('pandas_plot.png')
plt.show()

# Outros tipos de gráficos com Pandas
df = pd.DataFrame({
    'A': np.random.randn(1000) + 1,
    'B': np.random.randn(1000),
    'C': np.random.randn(1000) - 1
})

# Histograma
df.hist(bins=30, figsize=(12, 8))
plt.suptitle('Histogramas das Colunas')
plt.tight_layout()
plt.subplots_adjust(top=0.95)
plt.savefig('pandas_hist.png')
plt.show()

# Box plot
df.boxplot(figsize=(10, 6))
plt.title('Box Plot das Colunas')
plt.grid(True, axis='y')
plt.savefig('pandas_boxplot.png')
plt.show()
```

Seaborn

Seaborn é uma biblioteca de visualização de dados baseada no Matplotlib, mas com uma interface de nível mais alto e estilos mais atraentes. Ela é especialmente útil para visualizar dados estatísticos e relações entre variáveis.

Instalação do Seaborn

```
pip install seaborn
```

Gráficos Básicos com Seaborn

```
import seaborn as sns
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Configurar o estilo
sns.set_theme(style="whitegrid")

# Carregar um dataset de exemplo
tips = sns.load_dataset("tips")
print(tips.head())

# Gráfico de dispersão
plt.figure(figsize=(10, 6))
sns.scatterplot(x="total_bill", y="tip", data=tips)
plt.title('Gorjeta vs. Valor Total da Conta')
plt.savefig('seaborn_scatter.png')
plt.show()

# Adicionar uma terceira dimensão com a cor
plt.figure(figsize=(10, 6))
sns.scatterplot(x="total_bill", y="tip", hue="time", data=tips)
plt.title('Gorjeta vs. Valor Total da Conta (por Horário)')
plt.savefig('seaborn_scatter_hue.png')
plt.show()

# Adicionar uma quarta dimensão com o tamanho
plt.figure(figsize=(10, 6))
sns.scatterplot(x="total_bill", y="tip", hue="time", size="size", data=tips)
plt.title('Gorjeta vs. Valor Total da Conta (por Horário e Tamanho da Mesa)')
plt.savefig('seaborn_scatter_hue_size.png')
plt.show()
```

Gráficos de Distribuição

Seaborn é excelente para visualizar distribuições:

```
# Histograma
plt.figure(figsize=(10, 6))
sns.histplot(tips["total_bill"], kde=True, bins=30)
plt.title('Distribuição do Valor Total da Conta')
plt.savefig('seaborn_hist.png')
plt.show()

# Gráfico de densidade
plt.figure(figsize=(10, 6))
sns.kdeplot(data=tips, x="total_bill", hue="time", fill=True)
plt.title('Densidade do Valor Total da Conta por Horário')
plt.savefig('seaborn_kde.png')
```

```
plt.show()

# Gráfico de violino
plt.figure(figsize=(10, 6))
sns.violinplot(x="day", y="total_bill", hue="sex", data=tips, split=True)
plt.title('Distribuição do Valor Total da Conta por Dia e Sexo')
plt.savefig('seaborn_violin.png')
plt.show()

# Box plot
plt.figure(figsize=(10, 6))
sns.boxplot(x="day", y="total_bill", hue="time", data=tips)
plt.title('Box Plot do Valor Total da Conta por Dia e Horário')
plt.savefig('seaborn_boxplot.png')
plt.show()

# Strip plot (gráfico de dispersão categórico)
plt.figure(figsize=(10, 6))
sns.stripplot(x="day", y="total_bill", hue="sex", data=tips, dodge=True)
plt.title('Strip Plot do Valor Total da Conta por Dia e Sexo')
plt.savefig('seaborn_stripplot.png')
plt.show()

# Swarm plot (similar ao strip plot, mas evita sobreposição)
plt.figure(figsize=(10, 6))
sns.swarmplot(x="day", y="total_bill", hue="sex", data=tips, dodge=True)
plt.title('Swarm Plot do Valor Total da Conta por Dia e Sexo')
plt.savefig('seaborn_swarmplot.png')
plt.show()
```

Gráficos de Relação

Seaborn facilita a visualização de relações entre variáveis:

```
# Gráfico de dispersão com linha de regressão
plt.figure(figsize=(10, 6))
sns.regplot(x="total_bill", y="tip", data=tips)
plt.title('Relação entre Valor Total da Conta e Gorjeta com Linha de Regressão')
plt.savefig('seaborn_regplot.png')
plt.show()

# Gráfico de dispersão com linha de regressão por categoria
plt.figure(figsize=(10, 6))
sns.lmplot(x="total_bill", y="tip", hue="smoker", data=tips, height=6, aspect=1.5)
plt.title('Relação entre Valor Total da Conta e Gorjeta por Status de Fumante')
plt.savefig('seaborn_lmplot.png')
plt.show()

# Matriz de gráficos de dispersão
plt.figure(figsize=(12, 10))
sns.pairplot(tips, hue="sex")
plt.suptitle('Matriz de Gráficos de Dispersão', y=1.02)
plt.savefig('seaborn_pairplot.png')
plt.show()
```

```
# Mapa de calor de correlação
# Criar um DataFrame com mais variáveis numéricas
iris = sns.load_dataset("iris")
plt.figure(figsize=(10, 8))
correlation = iris.corr()
sns.heatmap(correlation, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Mapa de Calor de Correlação')
plt.tight_layout()
plt.savefig('seaborn_heatmap.png')
plt.show()
```

Gráficos Categóricos

Seaborn tem funções específicas para dados categóricos:

```
# Gráfico de barras
plt.figure(figsize=(10, 6))
sns.barplot(x="day", y="total_bill", hue="sex", data=tips)
plt.title('Valor Médio da Conta por Dia e Sexo')
plt.savefig('seaborn_barplot.png')
plt.show()

# Gráfico de contagem
plt.figure(figsize=(10, 6))
sns.countplot(x="day", hue="sex", data=tips)
plt.title('Contagem de Clientes por Dia e Sexo')
plt.savefig('seaborn_countplot.png')
plt.show()

# Gráfico de pontos
plt.figure(figsize=(10, 6))
sns.pointplot(x="day", y="tip", hue="sex", data=tips)
plt.title('Gorjeta Média por Dia e Sexo')
plt.savefig('seaborn_pointplot.png')
plt.show()

# Gráfico de caixa e bigodes categórico
plt.figure(figsize=(12, 6))
sns.catplot(x="day", y="total_bill", hue="sex", kind="box", data=tips, height=6, aspect=1.5)
plt.title('Box Plot do Valor Total da Conta por Dia e Sexo')
plt.savefig('seaborn_catplot_box.png')
plt.show()
```

Gráficos de Distribuição Conjunta

```
# Gráfico de distribuição conjunta
plt.figure(figsize=(10, 8))
sns.jointplot(x="total_bill", y="tip", data=tips, kind="scatter")
plt.suptitle('Distribuição Conjunta do Valor Total da Conta e Gorjeta', y=1.02)
plt.savefig('seaborn_jointplot.png')
plt.show()

# Gráfico de distribuição conjunta com regressão
```

```
plt.figure(figsize=(10, 8))
sns.jointplot(x="total_bill", y="tip", data=tips, kind="reg")
plt.suptitle('Distribuição Conjunta com Regressão', y=1.02)
plt.savefig('seaborn_jointplot_reg.png')
plt.show()

# Gráfico de distribuição conjunta com hexbin
plt.figure(figsize=(10, 8))
sns.jointplot(x="total_bill", y="tip", data=tips, kind="hex")
plt.suptitle('Distribuição Conjunta com Hexbin', y=1.02)
plt.savefig('seaborn_jointplot_hex.png')
plt.show()

# Gráfico de distribuição conjunta com KDE
plt.figure(figsize=(10, 8))
sns.jointplot(x="total_bill", y="tip", data=tips, kind="kde")
plt.suptitle('Distribuição Conjunta com KDE', y=1.02)
plt.savefig('seaborn_jointplot_kde.png')
plt.show()
```

Personalização de Gráficos no Seaborn

Seaborn permite personalizar a aparência dos gráficos:

```
# Definir o estilo
sns.set_style("whitegrid")  # Outros estilos: darkgrid, white, dark, ticks

# Definir a paleta de cores
sns.set_palette("Set2")  # Outras paletas: deep, muted, pastel, bright, dark, colorblind

# Definir o contexto (tamanho dos elementos)
sns.set_context("notebook")  # Outros contextos: paper, talk, poster

# Gráfico com personalização
plt.figure(figsize=(12, 8))
sns.scatterplot(x="total_bill", y="tip", hue="time", size="size",
                sizes=(20, 200), palette="viridis", data=tips)
plt.title('Gorjeta vs. Valor Total da Conta', fontsize=18)
plt.xlabel('Valor Total da Conta ($)', fontsize=14)
plt.ylabel('Gorjeta ($)', fontsize=14)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.legend(title='Horário', fontsize=12, title_fontsize=14)
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.savefig('seaborn_personalizado.png', dpi=300)
plt.show()
```

FacetGrid: Múltiplos Gráficos por Categoria

```
# Criar múltiplos gráficos por categoria
g = sns.FacetGrid(tips, col="time", row="sex", height=4, aspect=1.5)
g.map_dataframe(sns.scatterplot, x="total_bill", y="tip")
g.add_legend()
```

```
g.fig.suptitle('Gorjeta vs. Valor Total da Conta por Sexo e Horário', y=1.05)
g.set_axis_labels("Valor Total da Conta ($)", "Gorjeta ($)")
g.set_titles(col_template="{col_name}", row_template="{row_name}")
plt.tight_layout()
plt.savefig('seaborn_facetgrid.png')
plt.show()
```

Combinando Matplotlib e Seaborn

Como Seaborn é construído sobre Matplotlib, podemos combinar as duas bibliotecas:

```
# Configurar o estilo do Seaborn
sns.set_theme(style="whitegrid")

# Criar uma figura com subplots
fig, axes = plt.subplots(2, 2, figsize=(15, 12))

# Gráfico 1: Histograma com Matplotlib
axes[0, 0].hist(tips['total_bill'], bins=20, color='skyblue', edgecolor='black')
axes[0, 0].set_title('Histograma do Valor Total (Matplotlib)')
axes[0, 0].set_xlabel('Valor Total da Conta ($)')
axes[0, 0].set_ylabel('Frequência')
axes[0, 0].grid(True, linestyle='--', alpha=0.7)

# Gráfico 2: Histograma com Seaborn
sns.histplot(tips['total_bill'], bins=20, kde=True, ax=axes[0, 1])
axes[0, 1].set_title('Histograma do Valor Total (Seaborn)')
axes[0, 1].set_xlabel('Valor Total da Conta ($)')
axes[0, 1].set_ylabel('Frequência')

# Gráfico 3: Gráfico de dispersão com Matplotlib
scatter = axes[1, 0].scatter(tips['total_bill'], tips['tip'],
                             c=tips['size'], cmap='viridis', alpha=0.7)
axes[1, 0].set_title('Gorjeta vs. Valor Total (Matplotlib)')
axes[1, 0].set_xlabel('Valor Total da Conta ($)')
axes[1, 0].set_ylabel('Gorjeta ($)')
axes[1, 0].grid(True, linestyle='--', alpha=0.7)
plt.colorbar(scatter, ax=axes[1, 0], label='Tamanho da Mesa')

# Gráfico 4: Gráfico de dispersão com Seaborn
sns.scatterplot(x='total_bill', y='tip', hue='size', palette='viridis',
                data=tips, ax=axes[1, 1])
axes[1, 1].set_title('Gorjeta vs. Valor Total (Seaborn)')
axes[1, 1].set_xlabel('Valor Total da Conta ($)')
axes[1, 1].set_ylabel('Gorjeta ($)')

# Ajustar o layout
plt.tight_layout()
plt.suptitle('Comparação entre Matplotlib e Seaborn', fontsize=20, y=1.05)
plt.savefig('matplotlib_seaborn_comparacao.png')
plt.show()
```

Visualizações Interativas

Embora Matplotlib e Seaborn sejam principalmente para visualizações estáticas, podemos criar gráficos interativos usando bibliotecas como Plotly:

```
# Instalação: pip install plotly

import plotly.express as px
import plotly.graph_objects as go
import pandas as pd

# Carregar dados
tips = px.data.tips()

# Gráfico de dispersão interativo
fig = px.scatter(tips, x="total_bill", y="tip", color="sex", size="size",
                 hover_data=["day", "time", "smoker"],
                 title="Gorjeta vs. Valor Total da Conta")
fig.show()

# Salvar como HTML interativo
fig.write_html("plotly_scatter.html")

# Gráfico de barras interativo
fig = px.bar(tips, x="day", y="total_bill", color="sex", barmode="group",
             title="Valor Total da Conta por Dia e Sexo")
fig.show()
fig.write_html("plotly_bar.html")

# Gráfico de linha interativo
df = pd.DataFrame({
    'x': range(10),
    'y1': [1, 3, 2, 5, 4, 7, 6, 9, 8, 10],
    'y2': [10, 8, 9, 6, 7, 4, 5, 2, 3, 1]
})

fig = px.line(df, x='x', y=['y1', 'y2'], title="Gráfico de Linha Interativo")
fig.show()
fig.write_html("plotly_line.html")

# Gráfico 3D interativo
df = px.data.iris()
fig = px.scatter_3d(df, x='sepal_length', y='sepal_width', z='petal_length',
                    color='species', size='petal_width',
                    title="Gráfico 3D Interativo")
fig.show()
fig.write_html("plotly_3d.html")
```

Exemplo Prático: Análise Exploratória de Dados

Vamos combinar Pandas, Matplotlib e Seaborn para realizar uma análise exploratória de dados:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
import seaborn as sns

# Configurar o estilo
sns.set_theme(style="whitegrid")
plt.rcParams['figure.figsize'] = (12, 8)

# Carregar o dataset Titanic
titanic = sns.load_dataset('titanic')
print(titanic.head())

# 1. Visão geral dos dados
print("\nInformações do dataset:")
print(titanic.info())

print("\nEstatísticas descritivas:")
print(titanic.describe())

# 2. Análise univariada
# Distribuição de idades
plt.figure()
sns.histplot(titanic['age'].dropna(), kde=True, bins=30)
plt.title('Distribuição de Idades dos Passageiros')
plt.xlabel('Idade')
plt.ylabel('Frequência')
plt.savefig('titanic_age_dist.png')
plt.show()

# Contagem por classe
plt.figure()
sns.countplot(x='class', data=titanic)
plt.title('Número de Passageiros por Classe')
plt.xlabel('Classe')
plt.ylabel('Contagem')
plt.savefig('titanic_class_count.png')
plt.show()

# Contagem por sexo
plt.figure()
sns.countplot(x='sex', data=titanic)
plt.title('Número de Passageiros por Sexo')
plt.xlabel('Sexo')
plt.ylabel('Contagem')
plt.savefig('titanic_sex_count.png')
plt.show()

# Contagem de sobreviventes
plt.figure()
sns.countplot(x='survived', data=titanic)
plt.title('Número de Sobreviventes')
plt.xlabel('Sobreviveu (0 = Não, 1 = Sim)')
plt.ylabel('Contagem')
plt.savefig('titanic_survived_count.png')
plt.show()
```

```
# 3. Análise bivariada
# Sobrevivência por sexo
plt.figure()
sns.countplot(x='sex', hue='survived', data=titanic)
plt.title('Sobrevivência por Sexo')
plt.xlabel('Sexo')
plt.ylabel('Contagem')
plt.legend(title='Sobreviveu', labels=['Não', 'Sim'])
plt.savefig('titanic_survival_by_sex.png')
plt.show()

# Sobrevivência por classe
plt.figure()
sns.countplot(x='class', hue='survived', data=titanic)
plt.title('Sobrevivência por Classe')
plt.xlabel('Classe')
plt.ylabel('Contagem')
plt.legend(title='Sobreviveu', labels=['Não', 'Sim'])
plt.savefig('titanic_survival_by_class.png')
plt.show()

# Idade vs. Sobrevivência
plt.figure()
sns.boxplot(x='survived', y='age', data=titanic)
plt.title('Idade vs. Sobrevivência')
plt.xlabel('Sobreviveu (0 = Não, 1 = Sim)')
plt.ylabel('Idade')
plt.savefig('titanic_age_vs_survival.png')
plt.show()

# 4. Análise multivariada
# Sobrevivência por sexo e classe
plt.figure()
sns.catplot(x='sex', y='survived', hue='class', kind='bar', data=titanic, height=6, aspect=1.5)
plt.title('Taxa de Sobrevivência por Sexo e Classe')
plt.xlabel('Sexo')
plt.ylabel('Taxa de Sobrevivência')
plt.savefig('titanic_survival_by_sex_class.png')
plt.show()

# Distribuição de idades por sexo e classe
plt.figure()
g = sns.FacetGrid(titanic, col='sex', row='class', height=3, aspect=1.5)
g.map_dataframe(sns.histplot, 'age', kde=True)
g.set_axis_labels('Idade', 'Frequência')
g.set_titles(col_template='{col_name}', row_template='{row_name}')
g.fig.suptitle('Distribuição de Idades por Sexo e Classe', y=1.05)
plt.tight_layout()
plt.savefig('titanic_age_dist_by_sex_class.png')
plt.show()

# 5. Mapa de calor de correlação
plt.figure()
```

```
numeric_data = titanic.select_dtypes(include=[np.number])
correlation = numeric_data.corr()
sns.heatmap(correlation, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Mapa de Calor de Correlação')
plt.tight_layout()
plt.savefig('titanic_correlation_heatmap.png')
plt.show()

# 6. Pairplot para variáveis numéricas
plt.figure()
sns.pairplot(titanic[['survived', 'age', 'fare', 'pclass']], hue='survived')
plt.suptitle('Pairplot de Variáveis Numéricas', y=1.02)
plt.savefig('titanic_pairplot.png')
plt.show()

# 7. Gráfico de violino para tarifa por classe e sobrevivência
plt.figure()
sns.violinplot(x='class', y='fare', hue='survived', split=True, data=titanic)
plt.title('Distribuição de Tarifas por Classe e Sobrevivência')
plt.xlabel('Classe')
plt.ylabel('Tarifa ($)')
plt.legend(title='Sobreviveu', labels=['Não', 'Sim'])
plt.savefig('titanic_fare_by_class_survival.png')
plt.show()

# 8. Gráfico de barras empilhadas para sobrevivência por embarque e sexo
plt.figure()
survival_by_embark_sex = titanic.groupby(['embark_town', 'sex'])['survived'].mean().unstack()
survival_by_embark_sex.plot(kind='bar', stacked=False)
plt.title('Taxa de Sobrevivência por Porto de Embarque e Sexo')
plt.xlabel('Porto de Embarque')
plt.ylabel('Taxa de Sobrevivência')
plt.legend(title='Sexo')
plt.grid(axis='y')
plt.savefig('titanic_survival_by_embark_sex.png')
plt.show()

# 9. Gráfico de dispersão para idade vs. tarifa, colorido por sobrevivência
plt.figure()
sns.scatterplot(x='age', y='fare', hue='survived', size='pclass',
                sizes=(50, 200), data=titanic, palette='viridis')
plt.title('Idade vs. Tarifa por Sobrevivência e Classe')
plt.xlabel('Idade')
plt.ylabel('Tarifa ($)')
plt.legend(title='Sobreviveu', labels=['Não', 'Sim'])
plt.savefig('titanic_age_fare_scatter.png')
plt.show()

# 10. Dashboard final com subplots
plt.figure(figsize=(20, 15))

# Configuração dos subplots
plt.subplot(3, 3, 1)
```

```
sns.countplot(x='survived', data=titanic)
plt.title('Sobreviventes vs. Não Sobreviventes')
plt.xlabel('Sobreviveu (0 = Não, 1 = Sim)')
plt.ylabel('Contagem')

plt.subplot(3, 3, 2)
sns.countplot(x='sex', hue='survived', data=titanic)
plt.title('Sobrevivência por Sexo')
plt.xlabel('Sexo')
plt.ylabel('Contagem')
plt.legend(title='Sobreviveu', labels=['Não', 'Sim'])

plt.subplot(3, 3, 3)
sns.countplot(x='class', hue='survived', data=titanic)
plt.title('Sobrevivência por Classe')
plt.xlabel('Classe')
plt.ylabel('Contagem')
plt.legend(title='Sobreviveu', labels=['Não', 'Sim'])

plt.subplot(3, 3, 4)
sns.histplot(titanic['age'].dropna(), kde=True, bins=30)
plt.title('Distribuição de Idades')
plt.xlabel('Idade')
plt.ylabel('Frequência')

plt.subplot(3, 3, 5)
sns.boxplot(x='survived', y='age', data=titanic)
plt.title('Idade vs. Sobrevivência')
plt.xlabel('Sobreviveu (0 = Não, 1 = Sim)')
plt.ylabel('Idade')

plt.subplot(3, 3, 6)
sns.boxplot(x='class', y='age', hue='survived', data=titanic)
plt.title('Idade por Classe e Sobrevivência')
plt.xlabel('Classe')
plt.ylabel('Idade')
plt.legend(title='Sobreviveu', labels=['Não', 'Sim'])

plt.subplot(3, 3, 7)
sns.histplot(titanic['fare'].dropna(), kde=True, bins=30)
plt.title('Distribuição de Tarifas')
plt.xlabel('Tarifa ($)')
plt.ylabel('Frequência')

plt.subplot(3, 3, 8)
sns.boxplot(x='survived', y='fare', data=titanic)
plt.title('Tarifa vs. Sobrevivência')
plt.xlabel('Sobreviveu (0 = Não, 1 = Sim)')
plt.ylabel('Tarifa ($)')

plt.subplot(3, 3, 9)
sns.countplot(x='embark_town', hue='survived', data=titanic)
plt.title('Sobrevivência por Porto de Embarque')
```

```
plt.xlabel('Porto de Embarque')
plt.ylabel('Contagem')
plt.legend(title='Sobreviveu', labels=['Não', 'Sim'])

plt.tight_layout()
plt.suptitle('Análise Exploratória do Dataset Titanic', fontsize=24, y=1.02)
plt.savefig('titanic_dashboard.png', dpi=300)
plt.show()
```

Conclusão

Matplotlib e Seaborn são ferramentas poderosas para visualização de dados em Python. Matplotlib oferece grande flexibilidade e controle sobre cada aspecto dos gráficos, enquanto Seaborn fornece uma interface de alto nível com estilos atraentes e funções específicas para visualização estatística.

Ao dominar essas bibliotecas, você será capaz de:

1. Criar visualizações informativas e atraentes para explorar e comunicar dados.
2. Identificar padrões, tendências e relações nos dados.
3. Personalizar gráficos para atender às necessidades específicas de análise.
4. Combinar diferentes tipos de visualizações para criar dashboards completos.

A visualização de dados é uma habilidade essencial em ciência de dados e machine learning, pois permite compreender os dados antes de aplicar algoritmos complexos e comunicar resultados de forma eficaz.

No próximo tópico, exploraremos a manipulação e análise de dados em maior profundidade, preparando o terreno para a aplicação de algoritmos de machine learning.

Módulo 4: Introdução à Ciência de Dados e Machine Learning

Manipulação e Análise de Dados

A manipulação e análise de dados são etapas cruciais no processo de ciência de dados e machine learning. Antes de aplicar algoritmos sofisticados, é essencial preparar os dados adequadamente, o que envolve limpeza, transformação, agregação e exploração. Neste tópico, vamos explorar técnicas avançadas de manipulação e análise de dados usando Python.

Fluxo de Trabalho em Ciência de Dados

Um fluxo de trabalho típico em ciência de dados inclui as seguintes etapas:

1. **Coleta de dados**: Obtenção de dados de diversas fontes.
2. **Limpeza de dados**: Tratamento de valores ausentes, duplicados e outliers.
3. **Exploração de dados**: Análise estatística e visualização para entender os dados.
4. **Engenharia de features**: Criação de novas variáveis a partir das existentes.
5. **Preparação para modelagem**: Transformação, normalização e divisão dos dados.
6. **Modelagem**: Aplicação de algoritmos de machine learning.
7. **Avaliação**: Medição do desempenho do modelo.
8. **Implantação**: Colocação do modelo em produção.

Neste tópico, vamos focar nas etapas 2 a 5, que envolvem a manipulação e análise de dados.

Coleta de Dados

Antes de manipular e analisar dados, precisamos obtê-los. Python oferece várias maneiras de coletar dados:

Leitura de Arquivos

```
import pandas as pd

# CSV
df_csv = pd.read_csv('dados.csv')

# Excel
df_excel = pd.read_excel('dados.xlsx', sheet_name='Planilha1')

# JSON
df_json = pd.read_json('dados.json')

# Texto delimitado por tabulação
```

```
df_txt = pd.read_csv('dados.txt', sep='\t')

# Arquivos compactados
df_zip = pd.read_csv('dados.csv.gz', compression='gzip')
```

Conexão com Bancos de Dados

```
import pandas as pd
import sqlite3
from sqlalchemy import create_engine

# SQLite
conn = sqlite3.connect('banco.db')
df = pd.read_sql('SELECT * FROM tabela', conn)
conn.close()

# MySQL, PostgreSQL, etc. com SQLAlchemy
engine = create_engine('postgresql://usuario:senha@localhost:5432/banco')
df = pd.read_sql('SELECT * FROM tabela', engine)
```

APIs Web

```
import requests
import pandas as pd

# Requisição a uma API REST
response = requests.get('https://api.exemplo.com/dados')
dados = response.json()
df = pd.DataFrame(dados)

# API com autenticação
headers = {'Authorization': 'Bearer token123'}
response = requests.get('https://api.exemplo.com/dados', headers=headers)
dados = response.json()
df = pd.DataFrame(dados)
```

Web Scraping

```
import pandas as pd
from bs4 import BeautifulSoup
import requests

# Obter conteúdo HTML
url = 'https://exemplo.com/tabela'
response = requests.get(url)
soup = BeautifulSoup(response.text, 'html.parser')

# Extrair dados de uma tabela
tabela = soup.find('table')
df = pd.read_html(str(tabela))[0]
```

Limpeza de Dados

A limpeza de dados é uma etapa crucial que envolve tratar valores ausentes, duplicados, outliers e inconsistências.

Identificação e Tratamento de Valores Ausentes

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Criar um DataFrame com valores ausentes
df = pd.DataFrame({
    'A': [1, 2, np.nan, 4, 5],
    'B': [np.nan, 2, 3, np.nan, 5],
    'C': [1, 2, 3, 4, 5]
})

# Verificar valores ausentes
print("Valores ausentes por coluna:")
print(df.isna().sum())

# Visualizar valores ausentes
plt.figure(figsize=(10, 6))
sns.heatmap(df.isna(), cbar=False, cmap='viridis')
plt.title('Mapa de Calor de Valores Ausentes')
plt.tight_layout()
plt.show()

# Estratégias para lidar com valores ausentes

# 1. Remoção de linhas com valores ausentes
df_drop = df.dropna()
print("\nDataFrame após remover linhas com valores ausentes:")
print(df_drop)

# 2. Remoção de colunas com valores ausentes
df_drop_cols = df.dropna(axis=1)
print("\nDataFrame após remover colunas com valores ausentes:")
print(df_drop_cols)

# 3. Preenchimento com um valor constante
df_fill = df.fillna(0)
print("\nDataFrame após preencher valores ausentes com 0:")
print(df_fill)

# 4. Preenchimento com a média da coluna
df_mean = df.fillna(df.mean())
print("\nDataFrame após preencher valores ausentes com a média:")
print(df_mean)

# 5. Preenchimento com a mediana da coluna
df_median = df.fillna(df.median())
```

```
print("\nDataFrame após preencher valores ausentes com a mediana:")
print(df_median)

# 6. Preenchimento com o valor anterior ou posterior
df_ffill = df.fillna(method='ffill')  # forward fill
print("\nDataFrame após preencher valores ausentes com o valor anterior:")
print(df_ffill)

df_bfill = df.fillna(method='bfill')  # backward fill
print("\nDataFrame após preencher valores ausentes com o valor posterior:")
print(df_bfill)

# 7. Interpolação
df_interp = df.interpolate()
print("\nDataFrame após interpolação linear:")
print(df_interp)
```

Identificação e Tratamento de Duplicatas

```
# Criar um DataFrame com duplicatas
df = pd.DataFrame({
    'A': [1, 2, 2, 3, 3],
    'B': [5, 6, 6, 7, 7],
    'C': [9, 10, 10, 11, 11]
})

# Verificar duplicatas
print("Linhas duplicadas:")
print(df.duplicated())
print("\nNúmero de linhas duplicadas:", df.duplicated().sum())

# Remover duplicatas
df_unique = df.drop_duplicates()
print("\nDataFrame após remover duplicatas:")
print(df_unique)

# Remover duplicatas com base em colunas específicas
df_unique_cols = df.drop_duplicates(subset=['A'])
print("\nDataFrame após remover duplicatas com base na coluna A:")
print(df_unique_cols)

# Manter a última ocorrência em vez da primeira
df_unique_last = df.drop_duplicates(keep='last')
print("\nDataFrame após remover duplicatas (mantendo a última ocorrência):")
print(df_unique_last)
```

Identificação e Tratamento de Outliers

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

```python
# Criar um DataFrame com outliers
np.random.seed(42)
dados_normais = np.random.normal(0, 1, 100)
outliers = np.array([10, -10, 12, -12])
dados = np.concatenate([dados_normais, outliers])

df = pd.DataFrame({'valor': dados})

# Visualizar a distribuição dos dados
plt.figure(figsize=(10, 6))
sns.histplot(df['valor'], kde=True)
plt.title('Distribuição dos Dados com Outliers')
plt.xlabel('Valor')
plt.ylabel('Frequência')
plt.grid(True)
plt.show()

# Box plot para identificar outliers
plt.figure(figsize=(10, 6))
sns.boxplot(x=df['valor'])
plt.title('Box Plot para Identificação de Outliers')
plt.xlabel('Valor')
plt.grid(True)
plt.show()

# Identificar outliers usando o método IQR (Intervalo Interquartil)
Q1 = df['valor'].quantile(0.25)
Q3 = df['valor'].quantile(0.75)
IQR = Q3 - Q1

limite_inferior = Q1 - 1.5 * IQR
limite_superior = Q3 + 1.5 * IQR

print(f"Q1: {Q1}")
print(f"Q3: {Q3}")
print(f"IQR: {IQR}")
print(f"Limite inferior: {limite_inferior}")
print(f"Limite superior: {limite_superior}")

# Identificar outliers
outliers = df[(df['valor'] < limite_inferior) | (df['valor'] > limite_superior)]
print("\nOutliers identificados:")
print(outliers)
print(f"Número de outliers: {len(outliers)}")

# Estratégias para lidar com outliers

# 1. Remoção de outliers
df_sem_outliers = df[(df['valor'] >= limite_inferior) & (df['valor'] <= limite_superior)]
print("\nDataFrame após remover outliers:")
print(f"Tamanho original: {len(df)}")
print(f"Tamanho após remoção: {len(df_sem_outliers)}")
```

```
# 2. Substituição por limites (capping)
df_capped = df.copy()
df_capped.loc[df_capped['valor'] < limite_inferior, 'valor'] = limite_inferior
df_capped.loc[df_capped['valor'] > limite_superior, 'valor'] = limite_superior
print("\nDataFrame após substituir outliers pelos limites:")
print(df_capped.describe())

# 3. Transformação logarítmica (para dados positivos)
df_positivos = df[df['valor'] > 0].copy()
df_positivos['log_valor'] = np.log1p(df_positivos['valor'])  # log(1+x) para lidar com zeros
print("\nDataFrame após transformação logarítmica (apenas valores positivos):")
print(df_positivos[['valor', 'log_valor']].head())

# Visualizar a distribuição após a transformação
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
sns.histplot(df_positivos['valor'], kde=True)
plt.title('Distribuição Original')
plt.xlabel('Valor')
plt.ylabel('Frequência')
plt.grid(True)

plt.subplot(1, 2, 2)
sns.histplot(df_positivos['log_valor'], kde=True)
plt.title('Distribuição após Transformação Logarítmica')
plt.xlabel('Log(Valor)')
plt.ylabel('Frequência')
plt.grid(True)

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

Tratamento de Inconsistências

```
# Criar um DataFrame com inconsistências
df = pd.DataFrame({
    'nome': ['João', 'joão', 'JOÃO', 'Maria', 'maria'],
    'idade': [25, '25', 25.0, 30, '30'],
    'cidade': ['São Paulo', 'Sao Paulo', 'SAO PAULO', 'Rio de Janeiro', 'RIO DE JANEIRO']
})

print("DataFrame original:")
print(df)

# Padronizar strings (converter para minúsculas)
df['nome'] = df['nome'].str.lower()
df['cidade'] = df['cidade'].str.lower()
print("\nDataFrame após padronizar strings:")
print(df)

# Converter tipos de dados
df['idade'] = pd.to_numeric(df['idade'])
print("\nDataFrame após converter tipos de dados:")
```

```
print(df.dtypes)

# Padronizar valores categóricos
mapeamento_cidades = {
    'são paulo': 'São Paulo',
    'sao paulo': 'São Paulo',
    'rio de janeiro': 'Rio de Janeiro'
}

df['cidade'] = df['cidade'].map(mapeamento_cidades)
print("\nDataFrame após padronizar valores categóricos:")
print(df)

# Remover caracteres especiais e espaços extras
df_texto = pd.DataFrame({
    'texto': ['  Exemplo com espaços  ', 'Exemplo\ncom\nquebras', 'Exemplo@com@caracteres']
})

df_texto['texto_limpo'] = df_texto['texto'].str.strip()  # Remove espaços no início e fim
df_texto['texto_limpo'] = df_texto['texto_limpo'].str.replace('\n', ' ')  # Remove quebras de linha
df_texto['texto_limpo'] = df_texto['texto_limpo'].str.replace('@', ' ')  # Remove caracteres especiais
print("\nDataFrame após limpar textos:")
print(df_texto)
```

Exploração de Dados

A exploração de dados envolve a análise estatística e visualização para entender melhor os dados.

Estatísticas Descritivas

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Carregar um dataset de exemplo
df = sns.load_dataset('tips')
print("Primeiras linhas do dataset:")
print(df.head())

# Informações gerais sobre o dataset
print("\nInformações do dataset:")
print(df.info())

# Estatísticas descritivas básicas
print("\nEstatísticas descritivas:")
print(df.describe())

# Estatísticas descritivas para variáveis categóricas
print("\nEstatísticas para variáveis categóricas:")
print(df.describe(include=['object']))

# Contagem de valores para variáveis categóricas
```

```
print("\nContagem de valores para a coluna 'day':")
print(df['day'].value_counts())
print("\nContagem de valores para a coluna 'sex':")
print(df['sex'].value_counts())

# Estatísticas por grupo
print("\nEstatísticas por grupo (média):")
print(df.groupby('day').mean())

# Tabela de contingência (crosstab)
print("\nTabela de contingência entre 'day' e 'sex':")
print(pd.crosstab(df['day'], df['sex']))

# Correlação entre variáveis numéricas
print("\nMatriz de correlação:")
print(df.corr())

# Visualizar a matriz de correlação
plt.figure(figsize=(10, 8))
sns.heatmap(df.corr(), annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Matriz de Correlação')
plt.tight_layout()
plt.show()
```

Análise Univariada

```
# Análise univariada para variáveis numéricas
for coluna in ['total_bill', 'tip', 'size']:
    plt.figure(figsize=(12, 5))

    # Histograma
    plt.subplot(1, 2, 1)
    sns.histplot(df[coluna], kde=True)
    plt.title(f'Distribuição de {coluna}')
    plt.xlabel(coluna)
    plt.ylabel('Frequência')
    plt.grid(True)

    # Box plot
    plt.subplot(1, 2, 2)
    sns.boxplot(x=df[coluna])
    plt.title(f'Box Plot de {coluna}')
    plt.xlabel(coluna)
    plt.grid(True)

    plt.tight_layout()
    plt.show()

    # Estatísticas descritivas
    print(f"\nEstatísticas para {coluna}:")
    print(f"Média: {df[coluna].mean():.2f}")
    print(f"Mediana: {df[coluna].median():.2f}")
    print(f"Desvio padrão: {df[coluna].std():.2f}")
```

```
    print(f"Mínimo: {df[coluna].min():.2f}")
    print(f"Máximo: {df[coluna].max():.2f}")
    print(f"Assimetria: {df[coluna].skew():.2f}")
    print(f"Curtose: {df[coluna].kurtosis():.2f}")

# Análise univariada para variáveis categóricas
for coluna in ['sex', 'smoker', 'day', 'time']:
    plt.figure(figsize=(10, 6))

    # Gráfico de contagem
    sns.countplot(x=coluna, data=df)
    plt.title(f'Contagem de {coluna}')
    plt.xlabel(coluna)
    plt.ylabel('Contagem')
    plt.grid(True, axis='y')

    plt.tight_layout()
    plt.show()

    # Estatísticas
    print(f"\nEstatísticas para {coluna}:")
    contagem = df[coluna].value_counts()
    percentual = df[coluna].value_counts(normalize=True) * 100

    for valor in contagem.index:
        print(f"{valor}: {contagem[valor]} ({percentual[valor]:.1f}%)")
```

Análise Bivariada

```
# Análise bivariada: numérica vs. numérica
plt.figure(figsize=(10, 6))
sns.scatterplot(x='total_bill', y='tip', data=df)
plt.title('Gorjeta vs. Valor Total da Conta')
plt.xlabel('Valor Total da Conta ($)')
plt.ylabel('Gorjeta ($)')
plt.grid(True)
plt.show()

# Adicionar linha de regressão
plt.figure(figsize=(10, 6))
sns.regplot(x='total_bill', y='tip', data=df)
plt.title('Gorjeta vs. Valor Total da Conta com Linha de Regressão')
plt.xlabel('Valor Total da Conta ($)')
plt.ylabel('Gorjeta ($)')
plt.grid(True)
plt.show()

# Análise bivariada: numérica vs. categórica
plt.figure(figsize=(10, 6))
sns.boxplot(x='day', y='total_bill', data=df)
plt.title('Valor Total da Conta por Dia')
plt.xlabel('Dia')
plt.ylabel('Valor Total da Conta ($)')
```

```
plt.grid(True)
plt.show()

# Análise bivariada: categórica vs. categórica
plt.figure(figsize=(10, 6))
sns.countplot(x='day', hue='sex', data=df)
plt.title('Contagem por Dia e Sexo')
plt.xlabel('Dia')
plt.ylabel('Contagem')
plt.grid(True, axis='y')
plt.show()

# Tabela de contingência com percentuais
tabela = pd.crosstab(df['day'], df['sex'], normalize='index') * 100
print("\nTabela de contingência (percentuais por linha):")
print(tabela)

plt.figure(figsize=(10, 6))
tabela.plot(kind='bar', stacked=True)
plt.title('Distribuição de Sexo por Dia (%)')
plt.xlabel('Dia')
plt.ylabel('Percentual (%)')
plt.grid(True, axis='y')
plt.legend(title='Sexo')
plt.show()
```

Análise Multivariada

```
# Análise multivariada: numérica vs. numérica com terceira variável categórica
plt.figure(figsize=(10, 6))
sns.scatterplot(x='total_bill', y='tip', hue='sex', style='smoker', data=df)
plt.title('Gorjeta vs. Valor Total da Conta por Sexo e Status de Fumante')
plt.xlabel('Valor Total da Conta ($)')
plt.ylabel('Gorjeta ($)')
plt.grid(True)
plt.show()

# Pairplot para visualizar relações entre múltiplas variáveis numéricas
plt.figure(figsize=(12, 10))
sns.pairplot(df, hue='sex')
plt.suptitle('Matriz de Gráficos de Dispersão', y=1.02)
plt.show()

# Análise multivariada com facetas
g = sns.FacetGrid(df, col='day', row='sex', height=4, aspect=1.5)
g.map_dataframe(sns.scatterplot, x='total_bill', y='tip')
g.add_legend()
g.fig.suptitle('Gorjeta vs. Valor Total da Conta por Dia e Sexo', y=1.05)
g.set_axis_labels('Valor Total da Conta ($)', 'Gorjeta ($)')
g.set_titles(col_template='{col_name}', row_template='{row_name}')
plt.tight_layout()
plt.show()
```

```
# Análise multivariada com gráfico de calor
tabela_pivot = pd.pivot_table(df, values='tip',
                              index='day',
                              columns='time',
                              aggfunc='mean')
print("\nTabela pivot de gorjeta média por dia e horário:")
print(tabela_pivot)

plt.figure(figsize=(10, 6))
sns.heatmap(tabela_pivot, annot=True, cmap='YlGnBu', fmt='.2f')
plt.title('Gorjeta Média por Dia e Horário')
plt.tight_layout()
plt.show()
```

Engenharia de Features

A engenharia de features é o processo de criar novas variáveis a partir das existentes para melhorar o desempenho dos modelos de machine learning.

Criação de Novas Features

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Carregar um dataset de exemplo
df = sns.load_dataset('tips')
print("Dataset original:")
print(df.head())

# 1. Criar features baseadas em operações matemáticas
df['tip_percent'] = df['tip'] / df['total_bill'] * 100
print("\nDataset com percentual de gorjeta:")
print(df.head())

# 2. Criar features baseadas em agregações
media_gorjeta_por_dia = df.groupby('day')['tip'].mean().to_dict()
df['tip_vs_day_avg'] = df.apply(lambda x: x['tip'] / media_gorjeta_por_dia[x['day']], axis=1)
print("\nDataset com gorjeta relativa à média do dia:")
print(df.head())

# 3. Criar features baseadas em transformações
df['log_total_bill'] = np.log1p(df['total_bill'])
print("\nDataset com transformação logarítmica:")
print(df.head())

# 4. Criar features baseadas em datas
# Vamos criar um dataset com datas para este exemplo
datas = pd.date_range(start='2023-01-01', periods=len(df))
df['date'] = datas
print("\nDataset com datas:")
print(df.head())
```

```
# Extrair componentes da data
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
df['day_of_week'] = df['date'].dt.day_name()
df['is_weekend'] = df['day_of_week'].isin(['Saturday', 'Sunday']).astype(int)
print("\nDataset com features baseadas em datas:")
print(df.head())

# 5. Criar features baseadas em texto
# Vamos adicionar uma coluna de texto para este exemplo
comentarios = ['Ótimo serviço!', 'Comida deliciosa', 'Atendimento lento',
               'Ambiente agradável', 'Preço justo'] * (len(df) // 5 + 1)
df['comentario'] = comentarios[:len(df)]
print("\nDataset com comentários:")
print(df.head())

# Extrair features de texto
df['tamanho_comentario'] = df['comentario'].str.len()
df['tem_otimo'] = df['comentario'].str.contains('Ótimo|ótimo').astype(int)
df['tem_lento'] = df['comentario'].str.contains('lento|Lento').astype(int)
print("\nDataset com features baseadas em texto:")
print(df.head())

# 6. Criar features de interação
df['total_bill_per_person'] = df['total_bill'] / df['size']
df['tip_per_person'] = df['tip'] / df['size']
print("\nDataset com features de interação:")
print(df.head())

# Visualizar a relação entre as novas features
plt.figure(figsize=(10, 6))
sns.scatterplot(x='total_bill_per_person', y='tip_per_person', hue='sex', data=df)
plt.title('Gorjeta por Pessoa vs. Valor Total por Pessoa')
plt.xlabel('Valor Total por Pessoa ($)')
plt.ylabel('Gorjeta por Pessoa ($)')
plt.grid(True)
plt.show()
```

Transformação de Variáveis

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler
from sklearn.preprocessing import PowerTransformer, QuantileTransformer

# Criar um dataset com distribuição assimétrica
np.random.seed(42)
dados = np.random.exponential(scale=2, size=1000)
df = pd.DataFrame({'valor': dados})

# Visualizar a distribuição original
```

```
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
sns.histplot(df['valor'], kde=True)
plt.title('Distribuição Original')
plt.xlabel('Valor')
plt.ylabel('Frequência')
plt.grid(True)

plt.subplot(1, 2, 2)
sns.boxplot(x=df['valor'])
plt.title('Box Plot Original')
plt.xlabel('Valor')
plt.grid(True)

plt.tight_layout()
plt.show()

# 1. Transformação logarítmica
df['log_valor'] = np.log1p(df['valor'])  # log(1+x) para lidar com zeros

# 2. Transformação de raiz quadrada
df['sqrt_valor'] = np.sqrt(df['valor'])

# 3. Transformação Box-Cox
pt = PowerTransformer(method='box-cox')
df['boxcox_valor'] = pt.fit_transform(df[['valor']])

# 4. Transformação Yeo-Johnson
pt_yj = PowerTransformer(method='yeo-johnson')
df['yeojohnson_valor'] = pt_yj.fit_transform(df[['valor']])

# 5. Transformação de quantil (para distribuição normal)
qt = QuantileTransformer(output_distribution='normal')
df['quantile_valor'] = qt.fit_transform(df[['valor']])

# Visualizar as transformações
transformacoes = ['valor', 'log_valor', 'sqrt_valor', 'boxcox_valor', 'yeojohnson_valor', 'quantile_valo
fig, axes = plt.subplots(3, 2, figsize=(15, 12))
axes = axes.flatten()

for i, col in enumerate(transformacoes):
    sns.histplot(df[col], kde=True, ax=axes[i])
    axes[i].set_title(f'Distribuição após {col}')
    axes[i].set_xlabel(col)
    axes[i].set_ylabel('Frequência')
    axes[i].grid(True)

plt.tight_layout()
plt.show()

# Normalização e Padronização

# 1. Padronização (StandardScaler)
```

```
scaler = StandardScaler()
df['standard_valor'] = scaler.fit_transform(df[['valor']])

# 2. Normalização (MinMaxScaler)
min_max_scaler = MinMaxScaler()
df['minmax_valor'] = min_max_scaler.fit_transform(df[['valor']])

# 3. Escala robusta (RobustScaler)
robust_scaler = RobustScaler()
df['robust_valor'] = robust_scaler.fit_transform(df[['valor']])

# Visualizar as escalas
escalas = ['valor', 'standard_valor', 'minmax_valor', 'robust_valor']
fig, axes = plt.subplots(2, 2, figsize=(15, 10))
axes = axes.flatten()

for i, col in enumerate(escalas):
    sns.histplot(df[col], kde=True, ax=axes[i])
    axes[i].set_title(f'Distribuição após {col}')
    axes[i].set_xlabel(col)
    axes[i].set_ylabel('Frequência')
    axes[i].grid(True)

plt.tight_layout()
plt.show()

# Comparar estatísticas
print("Estatísticas das transformações:")
print(df[transformacoes + ['standard_valor', 'minmax_valor', 'robust_valor']].describe().T)
```

Codificação de Variáveis Categóricas

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder, OneHotEncoder, OrdinalEncoder

# Criar um dataset com variáveis categóricas
df = pd.DataFrame({
    'cor': ['vermelho', 'azul', 'verde', 'azul', 'vermelho', 'verde'],
    'tamanho': ['pequeno', 'médio', 'grande', 'médio', 'pequeno', 'grande'],
    'avaliacao': ['ruim', 'bom', 'excelente', 'bom', 'ruim', 'excelente']
})

print("Dataset original:")
print(df)

# 1. Label Encoding (para variáveis ordinais)
label_encoder = LabelEncoder()
df['avaliacao_label'] = label_encoder.fit_transform(df['avaliacao'])
print("\nDataset após Label Encoding:")
print(df)
```

```
print("\nMapeamento do Label Encoder:")
for i, categoria in enumerate(label_encoder.classes_):
    print(f"{categoria} -> {i}")

# 2. One-Hot Encoding (para variáveis nominais)
# Usando pandas get_dummies
df_onehot = pd.get_dummies(df['cor'], prefix='cor')
df = pd.concat([df, df_onehot], axis=1)
print("\nDataset após One-Hot Encoding (usando pandas):")
print(df)

# Usando scikit-learn
onehot_encoder = OneHotEncoder(sparse=False)
tamanho_encoded = onehot_encoder.fit_transform(df[['tamanho']])
tamanho_cols = [f'tamanho_{cat}' for cat in onehot_encoder.categories_[0]]
df_tamanho = pd.DataFrame(tamanho_encoded, columns=tamanho_cols, index=df.index)
df = pd.concat([df, df_tamanho], axis=1)
print("\nDataset após One-Hot Encoding para 'tamanho' (usando scikit-learn):")
print(df)

# 3. Ordinal Encoding (para variáveis ordinais com ordem conhecida)
# Definir a ordem das categorias
ordem_tamanho = ['pequeno', 'médio', 'grande']
ordem_avaliacao = ['ruim', 'bom', 'excelente']

ordinal_encoder_tamanho = OrdinalEncoder(categories=[ordem_tamanho])
df['tamanho_ordinal'] = ordinal_encoder_tamanho.fit_transform(df[['tamanho']])

ordinal_encoder_avaliacao = OrdinalEncoder(categories=[ordem_avaliacao])
df['avaliacao_ordinal'] = ordinal_encoder_avaliacao.fit_transform(df[['avaliacao']])

print("\nDataset após Ordinal Encoding:")
print(df)

# 4. Target Encoding (codificação baseada na variável alvo)
# Vamos adicionar uma variável alvo para este exemplo
df['target'] = [0, 1, 1, 0, 0, 1]

# Calcular a média da variável alvo para cada categoria
target_encoding = df.groupby('cor')['target'].mean().to_dict()
df['cor_target_encoded'] = df['cor'].map(target_encoding)

print("\nDataset após Target Encoding:")
print(df)
print("\nMapeamento do Target Encoding:")
print(target_encoding)

# 5. Frequency Encoding (codificação baseada na frequência)
freq_encoding = df['cor'].value_counts(normalize=True).to_dict()
df['cor_freq_encoded'] = df['cor'].map(freq_encoding)

print("\nDataset após Frequency Encoding:")
print(df)
```

```
print("\nMapeamento do Frequency Encoding:")
print(freq_encoding)

# Visualizar as diferentes codificações
plt.figure(figsize=(15, 10))

# Gráfico para Label Encoding
plt.subplot(2, 2, 1)
sns.scatterplot(x=range(len(df)), y='avaliacao_label', hue='avaliacao', data=df)
plt.title('Label Encoding para Avaliação')
plt.xlabel('Índice')
plt.ylabel('Valor Codificado')
plt.grid(True)

# Gráfico para Ordinal Encoding
plt.subplot(2, 2, 2)
sns.scatterplot(x=range(len(df)), y='tamanho_ordinal', hue='tamanho', data=df)
plt.title('Ordinal Encoding para Tamanho')
plt.xlabel('Índice')
plt.ylabel('Valor Codificado')
plt.grid(True)

# Gráfico para Target Encoding
plt.subplot(2, 2, 3)
sns.scatterplot(x=range(len(df)), y='cor_target_encoded', hue='cor', data=df)
plt.title('Target Encoding para Cor')
plt.xlabel('Índice')
plt.ylabel('Valor Codificado')
plt.grid(True)

# Gráfico para Frequency Encoding
plt.subplot(2, 2, 4)
sns.scatterplot(x=range(len(df)), y='cor_freq_encoded', hue='cor', data=df)
plt.title('Frequency Encoding para Cor')
plt.xlabel('Índice')
plt.ylabel('Valor Codificado')
plt.grid(True)

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

Seleção de Features

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_boston
from sklearn.feature_selection import SelectKBest, f_regression, RFE
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.preprocessing import StandardScaler
```

```
# Carregar o dataset Boston Housing
boston = load_boston()
X = pd.DataFrame(boston.data, columns=boston.feature_names)
y = boston.target

print("Dataset Boston Housing:")
print(X.head())

# 1. Seleção baseada em correlação
correlation = X.corrwith(pd.Series(y))
correlation_abs = correlation.abs().sort_values(ascending=False)
print("\nCorrelação com a variável alvo (em ordem decrescente de valor absoluto):")
print(correlation_abs)

# Visualizar as correlações
plt.figure(figsize=(12, 6))
correlation.abs().sort_values().plot(kind='barh')
plt.title('Correlação Absoluta com a Variável Alvo')
plt.xlabel('Correlação Absoluta')
plt.grid(True)
plt.show()

# 2. Seleção baseada em testes estatísticos (SelectKBest)
selector = SelectKBest(score_func=f_regression, k=5)
X_selected = selector.fit_transform(X, y)
selected_features = X.columns[selector.get_support()]

print("\nFeatures selecionadas pelo SelectKBest:")
print(selected_features)
print("\nScores:")
for feature, score in zip(X.columns, selector.scores_):
    print(f"{feature}: {score:.2f}")

# Visualizar os scores
plt.figure(figsize=(12, 6))
scores = pd.Series(selector.scores_, index=X.columns)
scores.sort_values().plot(kind='barh')
plt.title('Scores do SelectKBest (f_regression)')
plt.xlabel('Score')
plt.grid(True)
plt.show()

# 3. Seleção baseada em modelos (RFE - Recursive Feature Elimination)
model = LinearRegression()
rfe = RFE(estimator=model, n_features_to_select=5)
X_rfe = rfe.fit_transform(X, y)
selected_features_rfe = X.columns[rfe.support_]

print("\nFeatures selecionadas pelo RFE:")
print(selected_features_rfe)
print("\nRanking (1 = selecionada):")
for feature, ranking in zip(X.columns, rfe.ranking_):
    print(f"{feature}: {ranking}")
```

```
# Visualizar o ranking
plt.figure(figsize=(12, 6))
ranking = pd.Series(rfe.ranking_, index=X.columns)
ranking.sort_values().plot(kind='barh')
plt.title('Ranking do RFE (1 = selecionada)')
plt.xlabel('Ranking')
plt.grid(True)
plt.show()

# 4. Seleção baseada em importância de features (Random Forest)
rf = RandomForestRegressor(n_estimators=100, random_state=42)
rf.fit(X, y)
importance = rf.feature_importances_

print("\nImportância das features (Random Forest):")
for feature, importance in zip(X.columns, rf.feature_importances_):
    print(f"{feature}: {importance:.4f}")

# Visualizar a importância
plt.figure(figsize=(12, 6))
importance_df = pd.DataFrame({'feature': X.columns, 'importance': importance})
importance_df = importance_df.sort_values('importance', ascending=False)
sns.barplot(x='importance', y='feature', data=importance_df)
plt.title('Importância das Features (Random Forest)')
plt.xlabel('Importância')
plt.grid(True, axis='x')
plt.show()

# 5. Seleção baseada em PCA (Principal Component Analysis)
from sklearn.decomposition import PCA

# Padronizar os dados
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Aplicar PCA
pca = PCA()
X_pca = pca.fit_transform(X_scaled)

# Variância explicada
explained_variance = pca.explained_variance_ratio_
cumulative_variance = np.cumsum(explained_variance)

print("\nVariância explicada por cada componente principal:")
for i, var in enumerate(explained_variance):
    print(f"PC{i+1}: {var:.4f} ({cumulative_variance[i]:.4f} acumulada)")

# Visualizar a variância explicada
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.bar(range(1, len(explained_variance) + 1), explained_variance)
plt.title('Variância Explicada por Componente')
plt.xlabel('Componente Principal')
```

```
plt.ylabel('Variância Explicada')
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(range(1, len(cumulative_variance) + 1), cumulative_variance, marker='o')
plt.axhline(y=0.95, color='r', linestyle='--', label='95% de Variância')
plt.title('Variância Explicada Acumulada')
plt.xlabel('Número de Componentes')
plt.ylabel('Variância Explicada Acumulada')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

# Visualizar os loadings (contribuição de cada feature para os componentes principais)
loadings = pca.components_
loadings_df = pd.DataFrame(loadings.T, columns=[f'PC{i+1}' for i in range(loadings.shape[0])], index=X.
print("\nLoadings dos componentes principais:")
print(loadings_df.head())

# Visualizar os loadings dos dois primeiros componentes
plt.figure(figsize=(10, 8))
sns.heatmap(loadings_df.iloc[:, :2], annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Loadings dos Dois Primeiros Componentes Principais')
plt.tight_layout()
plt.show()
```

Preparação para Modelagem

A preparação para modelagem envolve a transformação final dos dados para que possam ser usados em algoritmos de machine learning.

Divisão em Conjuntos de Treino e Teste

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

# Carregar o dataset Iris
iris = load_iris()
X = pd.DataFrame(iris.data, columns=iris.feature_names)
y = iris.target

print("Dataset Iris:")
print(X.head())

# Divisão simples em treino e teste (80% treino, 20% teste)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)

print(f"\nTamanho do conjunto de treino: {X_train.shape[0]} amostras")
```

```
print(f"Tamanho do conjunto de teste: {X_test.shape[0]} amostras")

# Verificar a distribuição das classes
print("\nDistribuição das classes no conjunto completo:")
print(pd.Series(y).value_counts(normalize=True))

print("\nDistribuição das classes no conjunto de treino:")
print(pd.Series(y_train).value_counts(normalize=True))

print("\nDistribuição das classes no conjunto de teste:")
print(pd.Series(y_test).value_counts(normalize=True))

# Divisão estratificada (mantém a proporção das classes)
X_train_strat, X_test_strat, y_train_strat, y_test_strat = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y)

print("\nDistribuição das classes no conjunto de treino (estratificado):")
print(pd.Series(y_train_strat).value_counts(normalize=True))

print("\nDistribuição das classes no conjunto de teste (estratificado):")
print(pd.Series(y_test_strat).value_counts(normalize=True))

# Divisão em treino, validação e teste
X_temp, X_test, y_temp, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y)

X_train, X_val, y_train, y_val = train_test_split(
    X_temp, y_temp, test_size=0.25, random_state=42, stratify=y_temp)  # 0.25 * 0.8 = 0.2

print(f"\nTamanho do conjunto de treino: {X_train.shape[0]} amostras")
print(f"Tamanho do conjunto de validação: {X_val.shape[0]} amostras")
print(f"Tamanho do conjunto de teste: {X_test.shape[0]} amostras")

# Validação cruzada
from sklearn.model_selection import KFold, StratifiedKFold, cross_val_score
from sklearn.linear_model import LogisticRegression

# Criar um modelo
model = LogisticRegression(max_iter=200)

# K-Fold Cross Validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(model, X, y, cv=kf)

print("\nScores da validação cruzada K-Fold:")
print(f"Scores individuais: {scores}")
print(f"Média: {scores.mean():.4f}, Desvio padrão: {scores.std():.4f}")

# Stratified K-Fold Cross Validation
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
scores_strat = cross_val_score(model, X, y, cv=skf)

print("\nScores da validação cruzada Stratified K-Fold:")
```

```
print(f"Scores individuais: {scores_strat}")
print(f"Média: {scores_strat.mean():.4f}, Desvio padrão: {scores_strat.std():.4f}")
```

Pipeline de Pré-processamento

```
import pandas as pd
import numpy as np
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report

# Criar um dataset com variáveis numéricas e categóricas
np.random.seed(42)
n_samples = 1000

# Variáveis numéricas
idade = np.random.normal(40, 10, n_samples)
renda = np.random.exponential(scale=30000, size=n_samples)
score_credito = np.random.normal(700, 100, n_samples)

# Adicionar alguns valores ausentes
idade[np.random.choice(n_samples, 50, replace=False)] = np.nan
renda[np.random.choice(n_samples, 100, replace=False)] = np.nan
score_credito[np.random.choice(n_samples, 30, replace=False)] = np.nan

# Variáveis categóricas
educacao = np.random.choice(['Ensino Médio', 'Graduação', 'Pós-graduação'], n_samples)
estado_civil = np.random.choice(['Solteiro', 'Casado', 'Divorciado', 'Viúvo'], n_samples)
tem_casa_propria = np.random.choice(['Sim', 'Não'], n_samples)

# Adicionar alguns valores ausentes
educacao[np.random.choice(n_samples, 20, replace=False)] = np.nan
estado_civil[np.random.choice(n_samples, 15, replace=False)] = np.nan
tem_casa_propria[np.random.choice(n_samples, 10, replace=False)] = np.nan

# Variável alvo (inadimplente ou não)
inadimplente = (0.7 * (score_credito < 600) +
                0.2 * (renda < 20000) +
                0.1 * np.random.random(n_samples)) > 0.5

# Criar o DataFrame
df = pd.DataFrame({
    'idade': idade,
    'renda': renda,
    'score_credito': score_credito,
    'educacao': educacao,
    'estado_civil': estado_civil,
    'tem_casa_propria': tem_casa_propria,
    'inadimplente': inadimplente
```

```
})

print("Dataset de exemplo:")
print(df.head())
print("\nInformações do dataset:")
print(df.info())
print("\nValores ausentes por coluna:")
print(df.isna().sum())

# Dividir em features e target
X = df.drop('inadimplente', axis=1)
y = df['inadimplente']

# Dividir em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Definir as colunas numéricas e categóricas
numeric_features = ['idade', 'renda', 'score_credito']
categorical_features = ['educacao', 'estado_civil', 'tem_casa_propria']

# Criar transformadores para cada tipo de coluna
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

# Combinar os transformadores usando ColumnTransformer
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)
    ])

# Criar o pipeline completo com o modelo
pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', RandomForestClassifier(n_estimators=100, random_state=42))
])

# Treinar o modelo
pipeline.fit(X_train, y_train)

# Fazer previsões
y_pred = pipeline.predict(X_test)

# Avaliar o modelo
print("\nAcurácia:", accuracy_score(y_test, y_pred))
print("\nRelatório de classificação:")
```

```
print(classification_report(y_test, y_pred))

# Usar o pipeline para fazer previsões em novos dados
novos_dados = pd.DataFrame({
    'idade': [35, 45, np.nan],
    'renda': [50000, 30000, 20000],
    'score_credito': [720, 650, 580],
    'educacao': ['Graduação', 'Pós-graduação', 'Ensino Médio'],
    'estado_civil': ['Casado', 'Solteiro', np.nan],
    'tem_casa_propria': ['Sim', 'Não', 'Sim']
})

print("\nNovos dados:")
print(novos_dados)

print("\nPrevisões para os novos dados:")
previsoes = pipeline.predict(novos_dados)
probabilidades = pipeline.predict_proba(novos_dados)[:, 1]

for i, (pred, prob) in enumerate(zip(previsoes, probabilidades)):
    print(f"Cliente {i+1}: {'Inadimplente' if pred else 'Adimplente'} (Probabilidade de inadimplência:
```

Conclusão

A manipulação e análise de dados são etapas fundamentais no processo de ciência de dados e machine learning. Neste tópico, exploramos técnicas avançadas para limpar, transformar, explorar e preparar dados para modelagem.

Principais pontos abordados:

1. **Limpeza de dados**: Tratamento de valores ausentes, duplicados, outliers e inconsistências.
2. **Exploração de dados**: Análise estatística e visualização para entender melhor os dados.
3. **Engenharia de features**: Criação de novas variáveis para melhorar o desempenho dos modelos.
4. **Transformação de variáveis**: Normalização, padronização e transformações para lidar com distribuições assimétricas.
5. **Codificação de variáveis categóricas**: Técnicas para converter variáveis categóricas em numéricas.
6. **Seleção de features**: Métodos para identificar as variáveis mais importantes.
7. **Preparação para modelagem**: Divisão dos dados e criação de pipelines de pré-processamento.

Dominar essas técnicas é essencial para construir modelos de machine learning eficazes, pois a qualidade dos dados e das features tem um impacto significativo no desempenho dos modelos.

No próximo tópico, exploraremos os conceitos básicos de machine learning e como aplicar algoritmos de aprendizado supervisionado e não supervisionado aos dados que preparamos.

Módulo 4: Introdução à Ciência de Dados e Machine Learning

Conceitos Básicos de Machine Learning

O Machine Learning (Aprendizado de Máquina) é um subcampo da Inteligência Artificial que se concentra no desenvolvimento de algoritmos e técnicas que permitem aos computadores aprender a partir de dados e fazer previsões ou tomar decisões sem serem explicitamente programados para cada tarefa específica. Neste tópico, vamos explorar os conceitos fundamentais de Machine Learning e como eles são aplicados na prática.

O que é Machine Learning?

Machine Learning é a ciência de programar computadores para que eles possam aprender a partir de dados. Em vez de escrever regras explícitas para resolver um problema, fornecemos dados ao algoritmo e permitimos que ele descubra padrões e relações por conta própria.

A definição clássica de Tom Mitchell (1997) diz:

> "Um programa de computador aprende a partir da experiência E com respeito a alguma classe de tarefas T e medida de desempenho P, se seu desempenho em tarefas T, medido por P, melhora com a experiência E."

Em termos mais simples, um algoritmo de Machine Learning melhora seu desempenho em uma tarefa específica à medida que é exposto a mais dados.

Por que usar Machine Learning?

O Machine Learning é particularmente útil nas seguintes situações:

1. **Problemas complexos**: Quando as regras para resolver um problema são difíceis de definir explicitamente.
2. **Ambientes dinâmicos**: Quando o problema muda constantemente e as soluções precisam se adaptar.
3. **Personalização**: Quando é necessário adaptar soluções para diferentes usuários ou contextos.
4. **Automação**: Quando se deseja automatizar tarefas que normalmente requerem inteligência humana.
5. **Descoberta de padrões**: Quando se busca identificar padrões ocultos em grandes volumes de dados.

Tipos de Machine Learning

Existem três principais paradigmas de aprendizado em Machine Learning:

1. Aprendizado Supervisionado

No aprendizado supervisionado, o algoritmo é treinado em um conjunto de dados rotulado, onde cada exemplo consiste em um par de entrada e saída desejada. O objetivo é aprender um mapeamento da entrada para a saída.

Exemplos de tarefas: - Classificação: Prever uma categoria (ex: spam ou não spam) - Regressão: Prever um valor numérico (ex: preço de uma casa)

Algoritmos comuns: - Regressão Linear e Logística - Árvores de Decisão e Random Forest - Support Vector Machines (SVM) - k-Nearest Neighbors (k-NN) - Redes Neurais

2. Aprendizado Não Supervisionado

No aprendizado não supervisionado, o algoritmo é treinado em dados não rotulados e deve descobrir padrões ou estruturas por conta própria.

Exemplos de tarefas: - Clustering: Agrupar dados similares (ex: segmentação de clientes) - Redução de dimensionalidade: Reduzir o número de variáveis (ex: PCA) - Detecção de anomalias: Identificar outliers (ex: detecção de fraudes) - Regras de associação: Descobrir relações entre variáveis (ex: análise de cesta de compras)

Algoritmos comuns: - k-Means - DBSCAN - Hierarchical Clustering - Principal Component Analysis (PCA) - t-SNE - Autoencoders

3. Aprendizado por Reforço

No aprendizado por reforço, um agente aprende a tomar decisões interagindo com um ambiente. O agente recebe recompensas ou penalidades com base em suas ações e aprende a maximizar a recompensa total.

Exemplos de tarefas: - Jogos (ex: AlphaGo, Atari) - Robótica - Sistemas de recomendação - Otimização de processos

Algoritmos comuns: - Q-Learning - Deep Q-Network (DQN) - Policy Gradient - Actor-Critic - Proximal Policy Optimization (PPO)

Terminologia Básica em Machine Learning

Antes de mergulharmos nos algoritmos específicos, é importante entender alguns termos fundamentais:

Dados e Features

- **Dataset**: Conjunto de dados usado para treinar e testar um modelo.
- **Feature (Atributo)**: Variável de entrada usada para fazer previsões.
- **Target (Alvo)**: Variável que queremos prever.
- **Amostra (Exemplo)**: Um único ponto de dados no dataset.
- **Feature Engineering**: Processo de criação de novas features a partir das existentes.

Modelos e Parâmetros

- **Modelo**: Representação matemática aprendida a partir dos dados.
- **Parâmetros**: Valores internos do modelo que são ajustados durante o treinamento.
- **Hiperparâmetros**: Configurações do algoritmo que são definidas antes do treinamento.
- **Função de Custo (Loss Function)**: Medida de quão erradas estão as previsões do modelo.
- **Otimização**: Processo de ajustar os parâmetros do modelo para minimizar a função de custo.

Avaliação e Validação

- **Conjunto de Treino**: Dados usados para treinar o modelo.
- **Conjunto de Validação**: Dados usados para ajustar hiperparâmetros e evitar overfitting.
- **Conjunto de Teste**: Dados usados para avaliar o desempenho final do modelo.
- **Validação Cruzada**: Técnica para avaliar a capacidade de generalização do modelo.
- **Métricas de Avaliação**: Medidas para quantificar o desempenho do modelo (ex: acurácia, precisão, recall).

Desafios Comuns

- **Overfitting**: Quando o modelo se ajusta demais aos dados de treino e não generaliza bem.
- **Underfitting**: Quando o modelo é muito simples e não captura a complexidade dos dados.
- **Bias-Variance Tradeoff**: Equilíbrio entre um modelo muito simples (alto viés) e muito complexo (alta variância).
- **Dados Desbalanceados**: Quando algumas classes têm muito mais exemplos que outras.
- **Curse of Dimensionality**: Problemas que surgem ao trabalhar com dados de alta dimensionalidade.

Fluxo de Trabalho em Machine Learning

Um projeto típico de Machine Learning segue estas etapas:

1. **Definição do Problema**: Entender o problema e definir os objetivos.
2. **Coleta de Dados**: Obter dados relevantes para o problema.
3. **Exploração e Visualização**: Analisar os dados para entender suas características.
4. **Pré-processamento**: Limpar, transformar e preparar os dados para modelagem.
5. **Seleção de Features**: Escolher as variáveis mais relevantes.
6. **Seleção do Modelo**: Escolher algoritmos apropriados para o problema.
7. **Treinamento do Modelo**: Ajustar os parâmetros do modelo aos dados de treino.
8. **Avaliação do Modelo**: Medir o desempenho do modelo em dados não vistos.
9. **Ajuste de Hiperparâmetros**: Otimizar as configurações do modelo.
10. **Implantação**: Colocar o modelo em produção.
11. **Monitoramento**: Acompanhar o desempenho do modelo ao longo do tempo.

Bibliotecas de Machine Learning em Python

Python oferece várias bibliotecas poderosas para Machine Learning:

- **Scikit-learn**: Biblioteca abrangente com implementações de diversos algoritmos.
- **TensorFlow**: Biblioteca de aprendizado profundo desenvolvida pelo Google.
- **PyTorch**: Biblioteca de aprendizado profundo desenvolvida pelo Facebook.
- **XGBoost**: Biblioteca especializada em algoritmos de gradient boosting.
- **LightGBM**: Implementação eficiente de gradient boosting da Microsoft.
- **Statsmodels**: Biblioteca para modelos estatísticos e econométricos.

Exemplo Prático: Classificação com Scikit-learn

Vamos implementar um exemplo simples de classificação usando o dataset Iris e o algoritmo k-Nearest Neighbors:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Carregar o dataset Iris
iris = load_iris()
X = iris.data
y = iris.target
feature_names = iris.feature_names
```

```
target_names = iris.target_names

# Criar um DataFrame para facilitar a visualização
df = pd.DataFrame(X, columns=feature_names)
df['species'] = [target_names[i] for i in y]

print("Primeiras linhas do dataset:")
print(df.head())

# Visualizar a distribuição das classes
plt.figure(figsize=(10, 6))
sns.countplot(x='species', data=df)
plt.title('Distribuição das Espécies')
plt.xlabel('Espécie')
plt.ylabel('Contagem')
plt.grid(True, axis='y')
plt.show()

# Visualizar a relação entre as features
plt.figure(figsize=(12, 10))
sns.pairplot(df, hue='species')
plt.suptitle('Matriz de Gráficos de Dispersão por Espécie', y=1.02)
plt.show()

# Dividir os dados em conjuntos de treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)

# Padronizar as features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Criar e treinar o modelo k-NN
k = 5  # Número de vizinhos
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train_scaled, y_train)

# Fazer previsões
y_pred = knn.predict(X_test_scaled)

# Avaliar o modelo
accuracy = accuracy_score(y_test, y_pred)
print(f"\nAcurácia: {accuracy:.4f}")

print("\nRelatório de Classificação:")
print(classification_report(y_test, y_pred, target_names=target_names))

# Matriz de confusão
plt.figure(figsize=(10, 8))
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=target_names, yticklabels=target_names)
plt.title('Matriz de Confusão')
plt.xlabel('Previsão')
```

```
plt.ylabel('Valor Real')
plt.tight_layout()
plt.show()

# Visualizar as fronteiras de decisão (para 2 features)
def plot_decision_boundaries(X, y, model, feature_indices=[0, 1], feature_names=None):
    # Selecionar duas features para visualização
    X_subset = X[:, feature_indices]

    # Criar uma malha para visualizar as fronteiras de decisão
    h = 0.02  # Tamanho do passo da malha
    x_min, x_max = X_subset[:, 0].min() - 1, X_subset[:, 0].max() + 1
    y_min, y_max = X_subset[:, 1].min() - 1, X_subset[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    # Fazer previsões para cada ponto da malha
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plotar as fronteiras de decisão
    plt.figure(figsize=(10, 8))
    plt.contourf(xx, yy, Z, alpha=0.3, cmap='viridis')

    # Plotar os pontos de dados
    scatter = plt.scatter(X_subset[:, 0], X_subset[:, 1], c=y, edgecolors='k', cmap='viridis')
    plt.xlabel(feature_names[feature_indices[0]] if feature_names else f'Feature {feature_indices[0]}')
    plt.ylabel(feature_names[feature_indices[1]] if feature_names else f'Feature {feature_indices[1]}')
    plt.title('Fronteiras de Decisão do k-NN')
    plt.legend(handles=scatter.legend_elements()[0], labels=target_names)
    plt.tight_layout()
    plt.show()

# Treinar um modelo k-NN com apenas 2 features para visualização
knn_2d = KNeighborsClassifier(n_neighbors=k)
feature_indices = [0, 1]  # Comprimento e largura da sépala
X_2d_train = X_train_scaled[:, feature_indices]
knn_2d.fit(X_2d_train, y_train)

# Visualizar as fronteiras de decisão
plot_decision_boundaries(X_train_scaled, y_train, knn_2d, feature_indices, feature_names)

# Testar diferentes valores de k
k_values = range(1, 31)
train_scores = []
test_scores = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train_scaled, y_train)
    train_scores.append(knn.score(X_train_scaled, y_train))
    test_scores.append(knn.score(X_test_scaled, y_test))

# Plotar os resultados
```

```
plt.figure(figsize=(12, 6))
plt.plot(k_values, train_scores, 'o-', label='Treino')
plt.plot(k_values, test_scores, 'o-', label='Teste')
plt.xlabel('Número de Vizinhos (k)')
plt.ylabel('Acurácia')
plt.title('Acurácia vs. Número de Vizinhos')
plt.legend()
plt.grid(True)
plt.xticks(k_values)
plt.show()

# Encontrar o melhor valor de k
best_k = k_values[np.argmax(test_scores)]
print(f"\nMelhor valor de k: {best_k}")
print(f"Melhor acurácia de teste: {max(test_scores):.4f}")
```

Exemplo Prático: Regressão com Scikit-learn

Agora, vamos implementar um exemplo de regressão usando o dataset Boston Housing e o algoritmo de Regressão Linear:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.feature_selection import SelectKBest, f_regression

# Carregar o dataset Boston Housing
boston = load_boston()
X = boston.data
y = boston.target
feature_names = boston.feature_names

# Criar um DataFrame para facilitar a visualização
df = pd.DataFrame(X, columns=feature_names)
df['PRICE'] = y

print("Primeiras linhas do dataset:")
print(df.head())

# Visualizar a distribuição do preço das casas
plt.figure(figsize=(10, 6))
sns.histplot(df['PRICE'], kde=True, bins=30)
plt.title('Distribuição do Preço das Casas')
plt.xlabel('Preço ($1000s)')
plt.ylabel('Frequência')
plt.grid(True)
plt.show()
```

```
# Visualizar a correlação entre as features e o preço
plt.figure(figsize=(12, 8))
correlation = df.corr()['PRICE'].sort_values(ascending=False)
sns.barplot(x=correlation.index, y=correlation.values)
plt.title('Correlação com o Preço das Casas')
plt.xlabel('Feature')
plt.ylabel('Correlação')
plt.xticks(rotation=90)
plt.grid(True, axis='y')
plt.tight_layout()
plt.show()

# Selecionar as features mais importantes
selector = SelectKBest(score_func=f_regression, k=5)
X_new = selector.fit_transform(X, y)
selected_indices = selector.get_support(indices=True)
selected_features = [feature_names[i] for i in selected_indices]

print("\nFeatures mais importantes:")
for feature, score in zip(selected_features, selector.scores_[selected_indices]):
    print(f"{feature}: {score:.2f}")

# Dividir os dados em conjuntos de treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Padronizar as features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Criar e treinar o modelo de Regressão Linear
lr = LinearRegression()
lr.fit(X_train_scaled, y_train)

# Fazer previsões
y_train_pred = lr.predict(X_train_scaled)
y_test_pred = lr.predict(X_test_scaled)

# Avaliar o modelo
train_mse = mean_squared_error(y_train, y_train_pred)
test_mse = mean_squared_error(y_test, y_test_pred)
train_rmse = np.sqrt(train_mse)
test_rmse = np.sqrt(test_mse)
train_r2 = r2_score(y_train, y_train_pred)
test_r2 = r2_score(y_test, y_test_pred)

print("\nAvaliação do Modelo:")
print(f"RMSE (Treino): {train_rmse:.4f}")
print(f"RMSE (Teste): {test_rmse:.4f}")
print(f"R² (Treino): {train_r2:.4f}")
print(f"R² (Teste): {test_r2:.4f}")

# Visualizar os coeficientes do modelo
```

```
plt.figure(figsize=(12, 8))
coef_df = pd.DataFrame({'Feature': feature_names, 'Coefficient': lr.coef_})
coef_df = coef_df.sort_values('Coefficient', ascending=False)
sns.barplot(x='Coefficient', y='Feature', data=coef_df)
plt.title('Coeficientes do Modelo de Regressão Linear')
plt.xlabel('Coeficiente')
plt.ylabel('Feature')
plt.grid(True, axis='x')
plt.tight_layout()
plt.show()

# Visualizar previsões vs. valores reais
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_test_pred, alpha=0.7)
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--')
plt.title('Valores Reais vs. Previsões')
plt.xlabel('Valores Reais')
plt.ylabel('Previsões')
plt.grid(True)
plt.tight_layout()
plt.show()

# Visualizar os resíduos
plt.figure(figsize=(10, 6))
residuos = y_test - y_test_pred
sns.histplot(residuos, kde=True, bins=30)
plt.title('Distribuição dos Resíduos')
plt.xlabel('Resíduo')
plt.ylabel('Frequência')
plt.grid(True)
plt.tight_layout()
plt.show()

plt.figure(figsize=(10, 6))
plt.scatter(y_test_pred, residuos, alpha=0.7)
plt.axhline(y=0, color='r', linestyle='--')
plt.title('Resíduos vs. Valores Previstos')
plt.xlabel('Valores Previstos')
plt.ylabel('Resíduos')
plt.grid(True)
plt.tight_layout()
plt.show()
```

Overfitting e Underfitting

Um dos desafios mais comuns em Machine Learning é encontrar o equilíbrio certo entre overfitting e underfitting:

- **Overfitting**: Ocorre quando o modelo se ajusta demais aos dados de treino, capturando ruído e detalhes específicos que não generalizam bem para novos dados.
- **Underfitting**: Ocorre quando o modelo é muito simples e não consegue capturar a complexidade dos dados, resultando em baixo desempenho tanto nos dados de treino quanto nos de teste.

Vamos visualizar esses conceitos com um exemplo:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error

# Gerar dados sintéticos
np.random.seed(42)
X = np.sort(np.random.rand(100, 1), axis=0)
y = np.sin(2 * np.pi * X).ravel() + np.random.normal(0, 0.1, 100)

# Dividir em treino e teste
X_train, X_test = X[:80], X[80:]
y_train, y_test = y[:80], y[80:]

# Criar modelos com diferentes graus de complexidade
degrees = [1, 3, 5, 15]
plt.figure(figsize=(14, 10))

for i, degree in enumerate(degrees):
    ax = plt.subplot(2, 2, i + 1)

    # Criar um modelo de regressão polinomial
    model = Pipeline([
        ('poly', PolynomialFeatures(degree=degree)),
        ('linear', LinearRegression())
    ])

    # Treinar o modelo
    model.fit(X_train, y_train)

    # Fazer previsões
    X_test_sorted = np.sort(X_test, axis=0)
    y_pred = model.predict(X_test_sorted)

    # Calcular o erro
    train_error = mean_squared_error(y_train, model.predict(X_train))
    test_error = mean_squared_error(y_test, model.predict(X_test))

    # Plotar os dados e o modelo
    plt.scatter(X_train, y_train, color='blue', s=30, alpha=0.5, label='Treino')
    plt.scatter(X_test, y_test, color='green', s=30, alpha=0.5, label='Teste')
    plt.plot(X_test_sorted, y_pred, color='red', label='Modelo')
    plt.title(f'Polinômio de Grau {degree}\nErro Treino: {train_error:.4f}, Erro Teste: {test_error:.4f}
    plt.xlabel('x')
    plt.ylabel('y')
    plt.ylim(-1.5, 1.5)
    plt.legend()
    plt.grid(True)

plt.tight_layout()
plt.suptitle('Overfitting vs. Underfitting', y=1.05, fontsize=16)
```

```
plt.show()
```

Regularização

A regularização é uma técnica para prevenir o overfitting, adicionando uma penalidade à função de custo para modelos complexos:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import Ridge, Lasso, ElasticNet
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_boston
from sklearn.metrics import mean_squared_error, r2_score

# Carregar o dataset Boston Housing
boston = load_boston()
X = boston.data
y = boston.target
feature_names = boston.feature_names

# Dividir em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Criar pipelines com diferentes tipos de regularização
pipelines = {
    'Linear Regression': Pipeline([
        ('scaler', StandardScaler()),
        ('model', LinearRegression())
    ]),
    'Ridge (L2)': Pipeline([
        ('scaler', StandardScaler()),
        ('model', Ridge(alpha=1.0))
    ]),
    'Lasso (L1)': Pipeline([
        ('scaler', StandardScaler()),
        ('model', Lasso(alpha=0.1))
    ]),
    'ElasticNet (L1+L2)': Pipeline([
        ('scaler', StandardScaler()),
        ('model', ElasticNet(alpha=0.1, l1_ratio=0.5))
    ])
}

# Treinar e avaliar cada modelo
results = {}

for name, pipeline in pipelines.items():
    pipeline.fit(X_train, y_train)
    y_train_pred = pipeline.predict(X_train)
    y_test_pred = pipeline.predict(X_test)

    train_mse = mean_squared_error(y_train, y_train_pred)
```

```
    test_mse = mean_squared_error(y_test, y_test_pred)
    train_r2 = r2_score(y_train, y_train_pred)
    test_r2 = r2_score(y_test, y_test_pred)

    results[name] = {
        'train_mse': train_mse,
        'test_mse': test_mse,
        'train_r2': train_r2,
        'test_r2': test_r2
    }

    print(f"\n{name}:")
    print(f"RMSE (Treino): {np.sqrt(train_mse):.4f}")
    print(f"RMSE (Teste): {np.sqrt(test_mse):.4f}")
    print(f"R² (Treino): {train_r2:.4f}")
    print(f"R² (Teste): {test_r2:.4f}")

# Visualizar os coeficientes dos diferentes modelos
plt.figure(figsize=(14, 10))

for i, (name, pipeline) in enumerate(pipelines.items()):
    if hasattr(pipeline['model'], 'coef_'):
        coef = pipeline['model'].coef_
        plt.subplot(2, 2, i + 1)
        plt.bar(range(len(coef)), coef)
        plt.title(f'Coeficientes - {name}')
        plt.xlabel('Feature')
        plt.ylabel('Coeficiente')
        plt.xticks(range(len(coef)), feature_names, rotation=90)
        plt.grid(True)

plt.tight_layout()
plt.suptitle('Comparação dos Coeficientes com Diferentes Regularizações', y=1.05, fontsize=16)
plt.show()

# Comparar o desempenho dos modelos
metrics = ['train_mse', 'test_mse', 'train_r2', 'test_r2']
labels = ['RMSE (Treino)', 'RMSE (Teste)', 'R² (Treino)', 'R² (Teste)']

plt.figure(figsize=(14, 10))

for i, (metric, label) in enumerate(zip(metrics, labels)):
    plt.subplot(2, 2, i + 1)
    values = [np.sqrt(results[name][metric]) if 'mse' in metric else results[name][metric] for name in p
    plt.bar(pipelines.keys(), values)
    plt.title(label)
    plt.ylabel(label.split(' ')[0])
    plt.xticks(rotation=45)
    plt.grid(True)

plt.tight_layout()
plt.suptitle('Comparação de Desempenho com Diferentes Regularizações', y=1.05, fontsize=16)
plt.show()
```

Validação Cruzada

A validação cruzada é uma técnica para avaliar a capacidade de generalização de um modelo, dividindo os dados em múltiplos conjuntos de treino e teste:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import cross_val_score, KFold, learning_curve
from sklearn.linear_model import LinearRegression
from sklearn.datasets import load_boston
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# Carregar o dataset Boston Housing
boston = load_boston()
X = boston.data
y = boston.target

# Criar um pipeline com padronização e regressão linear
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('model', LinearRegression())
])

# Realizar validação cruzada com 5 folds
cv = KFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(pipeline, X, y, cv=cv, scoring='neg_mean_squared_error')

# Converter MSE negativo para RMSE
rmse_scores = np.sqrt(-scores)

print("Resultados da Validação Cruzada (RMSE):")
print(f"Média: {rmse_scores.mean():.4f}")
print(f"Desvio Padrão: {rmse_scores.std():.4f}")
print(f"Scores individuais: {rmse_scores}")

# Visualizar os resultados da validação cruzada
plt.figure(figsize=(10, 6))
plt.bar(range(1, 6), rmse_scores)
plt.axhline(y=rmse_scores.mean(), color='r', linestyle='--', label=f'Média: {rmse_scores.mean():.4f}')
plt.xlabel('Fold')
plt.ylabel('RMSE')
plt.title('Resultados da Validação Cruzada (5-Fold)')
plt.xticks(range(1, 6))
plt.legend()
plt.grid(True)
plt.show()

# Curva de aprendizado
train_sizes, train_scores, test_scores = learning_curve(
    pipeline, X, y, train_sizes=np.linspace(0.1, 1.0, 10),
    cv=cv, scoring='neg_mean_squared_error')

# Converter MSE negativo para RMSE
```

```
train_rmse = np.sqrt(-train_scores).mean(axis=1)
test_rmse = np.sqrt(-test_scores).mean(axis=1)

plt.figure(figsize=(10, 6))
plt.plot(train_sizes, train_rmse, 'o-', color='r', label='Treino')
plt.plot(train_sizes, test_rmse, 'o-', color='g', label='Validação')
plt.xlabel('Tamanho do Conjunto de Treino')
plt.ylabel('RMSE')
plt.title('Curva de Aprendizado')
plt.legend()
plt.grid(True)
plt.show()
```

Ajuste de Hiperparâmetros

O ajuste de hiperparâmetros é o processo de encontrar as melhores configurações para um algoritmo de Machine Learning:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from sklearn.ensemble import RandomForestRegressor
from sklearn.datasets import load_boston
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error, r2_score

# Carregar o dataset Boston Housing
boston = load_boston()
X = boston.data
y = boston.target

# Criar um pipeline com padronização e Random Forest
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('model', RandomForestRegressor(random_state=42))
])

# Definir os hiperparâmetros a serem testados
param_grid = {
    'model__n_estimators': [50, 100, 200],
    'model__max_depth': [None, 10, 20, 30],
    'model__min_samples_split': [2, 5, 10],
    'model__min_samples_leaf': [1, 2, 4]
}

# Realizar Grid Search
grid_search = GridSearchCV(
    pipeline, param_grid, cv=5, scoring='neg_mean_squared_error',
    n_jobs=-1, verbose=1
)

grid_search.fit(X, y)
```

```
# Melhores hiperparâmetros e score
print("Melhores Hiperparâmetros (Grid Search):")
print(grid_search.best_params_)
print(f"Melhor RMSE: {np.sqrt(-grid_search.best_score_):.4f}")

# Avaliar o modelo com os melhores hiperparâmetros
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X)
mse = mean_squared_error(y, y_pred)
r2 = r2_score(y, y_pred)

print(f"\nAvaliação do Melhor Modelo:")
print(f"RMSE: {np.sqrt(mse):.4f}")
print(f"R²: {r2:.4f}")

# Visualizar a importância das features
if hasattr(best_model['model'], 'feature_importances_'):
    importances = best_model['model'].feature_importances_
    indices = np.argsort(importances)[::-1]

    plt.figure(figsize=(12, 6))
    plt.bar(range(X.shape[1]), importances[indices])
    plt.xticks(range(X.shape[1]), [boston.feature_names[i] for i in indices], rotation=90)
    plt.xlabel('Feature')
    plt.ylabel('Importância')
    plt.title('Importância das Features (Random Forest)')
    plt.grid(True, axis='y')
    plt.tight_layout()
    plt.show()

# Visualizar os resultados do Grid Search
results = pd.DataFrame(grid_search.cv_results_)
results['mean_rmse'] = np.sqrt(-results['mean_test_score'])
results = results.sort_values('mean_rmse')

plt.figure(figsize=(12, 6))
plt.errorbar(range(len(results)), results['mean_rmse'], yerr=results['std_test_score'])
plt.xlabel('Combinação de Hiperparâmetros')
plt.ylabel('RMSE')
plt.title('Resultados do Grid Search')
plt.grid(True)
plt.tight_layout()
plt.show()

# Realizar Random Search (mais eficiente para grandes espaços de hiperparâmetros)
param_dist = {
    'model__n_estimators': np.arange(50, 201, 10),
    'model__max_depth': [None] + list(np.arange(10, 31, 2)),
    'model__min_samples_split': np.arange(2, 11),
    'model__min_samples_leaf': np.arange(1, 5)
}

random_search = RandomizedSearchCV(
```

```
    pipeline, param_distributions=param_dist, n_iter=20, cv=5,
    scoring='neg_mean_squared_error', n_jobs=-1, random_state=42, verbose=1
)

random_search.fit(X, y)

# Melhores hiperparâmetros e score
print("\nMelhores Hiperparâmetros (Random Search):")
print(random_search.best_params_)
print(f"Melhor RMSE: {np.sqrt(-random_search.best_score_):.4f}")
```

Conclusão

Neste tópico, exploramos os conceitos fundamentais de Machine Learning, incluindo os diferentes tipos de aprendizado, terminologia básica, fluxo de trabalho e desafios comuns. Também implementamos exemplos práticos de classificação e regressão usando Scikit-learn, e abordamos técnicas importantes como regularização, validação cruzada e ajuste de hiperparâmetros.

O Machine Learning é um campo vasto e em constante evolução, com aplicações em praticamente todas as áreas da ciência e da indústria. Nos próximos tópicos, vamos explorar algoritmos específicos de aprendizado supervisionado e não supervisionado, bem como técnicas mais avançadas como redes neurais e deep learning.

Lembre-se de que a prática é essencial para dominar o Machine Learning. Experimente diferentes algoritmos, datasets e técnicas para desenvolver sua intuição e habilidades nessa área fascinante.

Módulo 4: Introdução à Ciência de Dados e Machine Learning

Algoritmos de Aprendizado Supervisionado

O aprendizado supervisionado é um paradigma de machine learning onde o algoritmo aprende a partir de dados rotulados, ou seja, exemplos onde a resposta correta (saída desejada) é fornecida. O objetivo é aprender um mapeamento da entrada para a saída que possa ser usado para fazer previsões em novos dados não vistos. Neste tópico, vamos explorar os principais algoritmos de aprendizado supervisionado, suas características, vantagens, desvantagens e aplicações.

Classificação vs. Regressão

Os problemas de aprendizado supervisionado podem ser divididos em duas categorias principais:

1. **Classificação**: Quando a variável alvo é categórica (discreta), como prever se um e-mail é spam ou não.
2. **Regressão**: Quando a variável alvo é contínua, como prever o preço de uma casa.

Vamos explorar os algoritmos mais comuns para cada tipo de problema.

Algoritmos de Classificação

1. Regressão Logística

Apesar do nome, a regressão logística é um algoritmo de classificação que estima a probabilidade de um exemplo pertencer a uma determinada classe.

Características: - Modelo linear para classificação binária (pode ser estendido para multiclasse) - Usa a função logística (sigmoid) para mapear valores para probabilidades entre 0 e 1 - Fronteira de decisão linear

Vantagens: - Simples e interpretável - Eficiente para treinar - Fornece probabilidades bem calibradas - Funciona bem com dados linearmente separáveis

Desvantagens: - Não captura relações não lineares sem engenharia de features - Pode sofrer com multicolinearidade - Sensível a outliers

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report, roc_curve, auc
import seaborn as sns

# Carregar o dataset de câncer de mama
data = load_breast_cancer()
X = data.data
y = data.target
feature_names = data.feature_names
target_names = data.target_names

# Dividir em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Padronizar os dados
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Criar e treinar o modelo
model = LogisticRegression(max_iter=1000, random_state=42)
model.fit(X_train_scaled, y_train)

# Fazer previsões
y_pred = model.predict(X_test_scaled)
y_prob = model.predict_proba(X_test_scaled)[:, 1]

# Avaliar o modelo
accuracy = accuracy_score(y_test, y_pred)
print(f"Acurácia: {accuracy:.4f}")

print("\nMatriz de Confusão:")
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=target_names, yticklabels=target_names)
plt.xlabel('Previsão')
plt.ylabel('Valor Real')
plt.title('Matriz de Confusão')
plt.show()

print("\nRelatório de Classificação:")
print(classification_report(y_test, y_pred, target_names=target_names))

# Curva ROC
fpr, tpr, _ = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('Taxa de Falsos Positivos')
```

```
plt.ylabel('Taxa de Verdadeiros Positivos')
plt.title('Curva ROC')
plt.legend(loc="lower right")
plt.grid(True)
plt.show()

# Visualizar os coeficientes (importância das features)
coef = model.coef_[0]
indices = np.argsort(np.abs(coef))[::-1]

plt.figure(figsize=(12, 8))
plt.bar(range(len(coef)), coef[indices])
plt.xticks(range(len(coef)), [feature_names[i] for i in indices], rotation=90)
plt.xlabel('Feature')
plt.ylabel('Coeficiente')
plt.title('Importância das Features na Regressão Logística')
plt.grid(True, axis='y')
plt.tight_layout()
plt.show()
```

2. k-Nearest Neighbors (k-NN)

O k-NN é um algoritmo baseado em instância que classifica um novo exemplo com base na maioria dos k vizinhos mais próximos.

Características: - Algoritmo não paramétrico (não faz suposições sobre a distribuição dos dados) - Classificação baseada na proximidade (distância) entre os pontos - Fronteira de decisão não linear

Vantagens: - Simples e intuitivo - Não requer treinamento (lazy learning) - Funciona bem com dados de baixa dimensionalidade - Adaptável a problemas complexos

Desvantagens: - Computacionalmente intensivo para grandes conjuntos de dados - Sensível à escala das features - Sensível a ruído e outliers - Sofre com a "maldição da dimensionalidade"

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import seaborn as sns

# Carregar o dataset Iris
data = load_iris()
X = data.data
y = data.target
feature_names = data.feature_names
target_names = data.target_names

# Usar apenas duas features para visualização
X = X[:, :2]
feature_names = feature_names[:2]
```

```
# Dividir em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Padronizar os dados
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Criar e treinar o modelo
k = 5
model = KNeighborsClassifier(n_neighbors=k)
model.fit(X_train_scaled, y_train)

# Fazer previsões
y_pred = model.predict(X_test_scaled)

# Avaliar o modelo
accuracy = accuracy_score(y_test, y_pred)
print(f"Acurácia: {accuracy:.4f}")

print("\nMatriz de Confusão:")
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=target_names, yticklabels=target_names)
plt.xlabel('Previsão')
plt.ylabel('Valor Real')
plt.title('Matriz de Confusão')
plt.show()

print("\nRelatório de Classificação:")
print(classification_report(y_test, y_pred, target_names=target_names))

# Visualizar a fronteira de decisão
def plot_decision_boundary(X, y, model, scaler):
    h = 0.02  # Tamanho do passo da malha
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    # Padronizar os pontos da malha
    mesh_points = np.c_[xx.ravel(), yy.ravel()]
    mesh_points_scaled = scaler.transform(mesh_points)

    # Fazer previsões para cada ponto da malha
    Z = model.predict(mesh_points_scaled)
    Z = Z.reshape(xx.shape)

    # Plotar a fronteira de decisão
    plt.figure(figsize=(10, 8))
    plt.contourf(xx, yy, Z, alpha=0.3, cmap='viridis')

    # Plotar os pontos de dados
```

```
    for i, target in enumerate(target_names):
        plt.scatter(X[y == i, 0], X[y == i, 1], label=target)

    plt.xlabel(feature_names[0])
    plt.ylabel(feature_names[1])
    plt.title(f'Fronteira de Decisão do k-NN (k={k})')
    plt.legend()
    plt.grid(True)
    plt.show()

# Visualizar a fronteira de decisão
plot_decision_boundary(X, y, model, scaler)

# Testar diferentes valores de k
k_values = range(1, 31)
train_scores = []
test_scores = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train_scaled, y_train)
    train_scores.append(knn.score(X_train_scaled, y_train))
    test_scores.append(knn.score(X_test_scaled, y_test))

# Plotar os resultados
plt.figure(figsize=(10, 6))
plt.plot(k_values, train_scores, 'o-', label='Treino')
plt.plot(k_values, test_scores, 'o-', label='Teste')
plt.xlabel('Número de Vizinhos (k)')
plt.ylabel('Acurácia')
plt.title('Acurácia vs. Número de Vizinhos')
plt.legend()
plt.grid(True)
plt.xticks(k_values[::2])
plt.show()

# Encontrar o melhor valor de k
best_k = k_values[np.argmax(test_scores)]
print(f"\nMelhor valor de k: {best_k}")
print(f"Melhor acurácia de teste: {max(test_scores):.4f}")
```

3. Árvores de Decisão

As árvores de decisão são modelos que dividem o espaço de features em regiões, tomando decisões sequenciais baseadas nos valores das features.

Características: - Estrutura hierárquica de decisões (nós e folhas) - Divisões baseadas em regras simples (if-then) - Fronteira de decisão não linear e ortogonal aos eixos

Vantagens: - Fácil interpretação e visualização - Não requer normalização dos dados - Lida bem com features categóricas e numéricas - Captura relações não lineares

Desvantagens: - Tendência a overfitting (especialmente árvores profundas) - Instabilidade (pequenas mudanças nos dados podem gerar árvores muito diferentes) - Viés para features com mais níveis (no caso de categóricas)

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import seaborn as sns

# Carregar o dataset Iris
data = load_iris()
X = data.data
y = data.target
feature_names = data.feature_names
target_names = data.target_names

# Dividir em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Criar e treinar o modelo
model = DecisionTreeClassifier(max_depth=3, random_state=42)
model.fit(X_train, y_train)

# Fazer previsões
y_pred = model.predict(X_test)

# Avaliar o modelo
accuracy = accuracy_score(y_test, y_pred)
print(f"Acurácia: {accuracy:.4f}")

print("\nMatriz de Confusão:")
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=target_names, yticklabels=target_names)
plt.xlabel('Previsão')
plt.ylabel('Valor Real')
plt.title('Matriz de Confusão')
plt.show()

print("\nRelatório de Classificação:")
print(classification_report(y_test, y_pred, target_names=target_names))

# Visualizar a árvore de decisão
plt.figure(figsize=(20, 10))
plot_tree(model, filled=True, feature_names=feature_names, class_names=target_names, rounded=True)
plt.title('Árvore de Decisão')
plt.show()

# Importância das features
importances = model.feature_importances_
indices = np.argsort(importances)[::-1]
```

```
plt.figure(figsize=(10, 6))
plt.bar(range(X.shape[1]), importances[indices])
plt.xticks(range(X.shape[1]), [feature_names[i] for i in indices], rotation=90)
plt.xlabel('Feature')
plt.ylabel('Importância')
plt.title('Importância das Features na Árvore de Decisão')
plt.grid(True, axis='y')
plt.tight_layout()
plt.show()

# Testar diferentes profundidades da árvore
max_depths = range(1, 21)
train_scores = []
test_scores = []

for depth in max_depths:
    dt = DecisionTreeClassifier(max_depth=depth, random_state=42)
    dt.fit(X_train, y_train)
    train_scores.append(dt.score(X_train, y_train))
    test_scores.append(dt.score(X_test, y_test))

# Plotar os resultados
plt.figure(figsize=(10, 6))
plt.plot(max_depths, train_scores, 'o-', label='Treino')
plt.plot(max_depths, test_scores, 'o-', label='Teste')
plt.xlabel('Profundidade Máxima')
plt.ylabel('Acurácia')
plt.title('Acurácia vs. Profundidade da Árvore')
plt.legend()
plt.grid(True)
plt.xticks(max_depths)
plt.show()

# Encontrar a melhor profundidade
best_depth = max_depths[np.argmax(test_scores)]
print(f"\nMelhor profundidade: {best_depth}")
print(f"Melhor acurácia de teste: {max(test_scores):.4f}")
```

4. Random Forest

Random Forest é um algoritmo de ensemble que combina múltiplas árvores de decisão para melhorar a precisão e reduzir o overfitting.

Características: - Ensemble de árvores de decisão - Cada árvore é treinada em um subconjunto aleatório dos dados (bagging) - Cada árvore considera um subconjunto aleatório das features em cada divisão - A previsão final é a média (regressão) ou voto majoritário (classificação) das árvores

Vantagens: - Maior precisão que árvores individuais - Menos propenso a overfitting - Robusto a outliers e ruído - Fornece medidas de importância de features

Desvantagens: - Menos interpretável que uma única árvore - Computacionalmente mais intensivo - Pode ser lento para grandes conjuntos de dados - Tendência a favorecer features com mais níveis

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report, roc_curve, auc
import seaborn as sns

# Carregar o dataset de câncer de mama
data = load_breast_cancer()
X = data.data
y = data.target
feature_names = data.feature_names
target_names = data.target_names

# Dividir em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Criar e treinar o modelo
model = RandomForestClassifier(n_estimators=100, max_depth=5, random_state=42)
model.fit(X_train, y_train)

# Fazer previsões
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:, 1]

# Avaliar o modelo
accuracy = accuracy_score(y_test, y_pred)
print(f"Acurácia: {accuracy:.4f}")

print("\nMatriz de Confusão:")
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=target_names, yticklabels=target_names)
plt.xlabel('Previsão')
plt.ylabel('Valor Real')
plt.title('Matriz de Confusão')
plt.show()

print("\nRelatório de Classificação:")
print(classification_report(y_test, y_pred, target_names=target_names))

# Curva ROC
fpr, tpr, _ = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
```

```
plt.xlabel('Taxa de Falsos Positivos')
plt.ylabel('Taxa de Verdadeiros Positivos')
plt.title('Curva ROC')
plt.legend(loc="lower right")
plt.grid(True)
plt.show()

# Importância das features
importances = model.feature_importances_
indices = np.argsort(importances)[::-1]

plt.figure(figsize=(12, 8))
plt.bar(range(X.shape[1]), importances[indices])
plt.xticks(range(X.shape[1]), [feature_names[i] for i in indices], rotation=90)
plt.xlabel('Feature')
plt.ylabel('Importância')
plt.title('Importância das Features no Random Forest')
plt.grid(True, axis='y')
plt.tight_layout()
plt.show()

# Testar diferentes números de árvores
n_estimators = [10, 50, 100, 200, 500]
train_scores = []
test_scores = []

for n in n_estimators:
    rf = RandomForestClassifier(n_estimators=n, max_depth=5, random_state=42)
    rf.fit(X_train, y_train)
    train_scores.append(rf.score(X_train, y_train))
    test_scores.append(rf.score(X_test, y_test))

# Plotar os resultados
plt.figure(figsize=(10, 6))
plt.plot(n_estimators, train_scores, 'o-', label='Treino')
plt.plot(n_estimators, test_scores, 'o-', label='Teste')
plt.xlabel('Número de Árvores')
plt.ylabel('Acurácia')
plt.title('Acurácia vs. Número de Árvores')
plt.legend()
plt.grid(True)
plt.xscale('log')
plt.show()
```

5. Support Vector Machines (SVM)

SVM é um algoritmo que busca encontrar um hiperplano que melhor separa as classes, maximizando a margem entre elas.

Características: - Busca o hiperplano de margem máxima - Usa vetores de suporte (pontos próximos à fronteira) - Pode usar kernels para mapear dados para espaços de maior dimensão

Vantagens: - Eficaz em espaços de alta dimensionalidade - Versátil através de diferentes funções de kernel - Robusto contra overfitting em espaços de alta dimensão - Bom para dados com classes separáveis

Desvantagens: - Não escala bem para grandes conjuntos de dados - Sensível à escolha de parâmetros e kernel - Difícil interpretação - Não fornece probabilidades diretamente

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import seaborn as sns

# Carregar o dataset de câncer de mama
data = load_breast_cancer()
X = data.data
y = data.target
feature_names = data.feature_names
target_names = data.target_names

# Dividir em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Padronizar os dados
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Criar e treinar o modelo
model = SVC(kernel='rbf', C=1.0, gamma='scale', probability=True, random_state=42)
model.fit(X_train_scaled, y_train)

# Fazer previsões
y_pred = model.predict(X_test_scaled)
y_prob = model.predict_proba(X_test_scaled)[:, 1]

# Avaliar o modelo
accuracy = accuracy_score(y_test, y_pred)
print(f"Acurácia: {accuracy:.4f}")

print("\nMatriz de Confusão:")
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=target_names, yticklabels=target_names)
plt.xlabel('Previsão')
plt.ylabel('Valor Real')
plt.title('Matriz de Confusão')
plt.show()

print("\nRelatório de Classificação:")
print(classification_report(y_test, y_pred, target_names=target_names))
```

```
# Curva ROC
from sklearn.metrics import roc_curve, auc
fpr, tpr, _ = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('Taxa de Falsos Positivos')
plt.ylabel('Taxa de Verdadeiros Positivos')
plt.title('Curva ROC')
plt.legend(loc="lower right")
plt.grid(True)
plt.show()

# Visualizar a fronteira de decisão (para 2 features)
def plot_decision_boundary_svm(X, y, model, scaler, feature_indices=[0, 1]):
    # Selecionar duas features para visualização
    X_subset = X[:, feature_indices]

    # Criar um modelo SVM apenas com essas duas features
    svm_2d = SVC(kernel='rbf', C=1.0, gamma='scale', probability=True, random_state=42)
    X_subset_train, X_subset_test, y_train_2d, y_test_2d = train_test_split(
        X_subset, y, test_size=0.3, random_state=42)

    # Padronizar
    scaler_2d = StandardScaler()
    X_subset_train_scaled = scaler_2d.fit_transform(X_subset_train)
    X_subset_test_scaled = scaler_2d.transform(X_subset_test)

    # Treinar
    svm_2d.fit(X_subset_train_scaled, y_train_2d)

    # Criar uma malha para visualizar as fronteiras de decisão
    h = 0.02  # Tamanho do passo da malha
    x_min, x_max = X_subset[:, 0].min() - 1, X_subset[:, 0].max() + 1
    y_min, y_max = X_subset[:, 1].min() - 1, X_subset[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    # Padronizar os pontos da malha
    mesh_points = np.c_[xx.ravel(), yy.ravel()]
    mesh_points_scaled = scaler_2d.transform(mesh_points)

    # Fazer previsões para cada ponto da malha
    Z = svm_2d.predict(mesh_points_scaled)
    Z = Z.reshape(xx.shape)

    # Plotar as fronteiras de decisão
    plt.figure(figsize=(10, 8))
    plt.contourf(xx, yy, Z, alpha=0.3, cmap='viridis')
```

```
    # Plotar os pontos de dados
    for i, target in enumerate(target_names):
        plt.scatter(X_subset[y == i, 0], X_subset[y == i, 1], label=target)

    plt.xlabel(feature_names[feature_indices[0]])
    plt.ylabel(feature_names[feature_indices[1]])
    plt.title('Fronteira de Decisão do SVM (RBF Kernel)')
    plt.legend()
    plt.grid(True)
    plt.show()

# Visualizar a fronteira de decisão para as duas primeiras features
plot_decision_boundary_svm(X, y, model, scaler, [0, 1])

# Testar diferentes valores de C e gamma
C_values = [0.1, 1, 10, 100]
gamma_values = [0.01, 0.1, 1, 10]

plt.figure(figsize=(12, 10))
for i, C in enumerate(C_values):
    for j, gamma in enumerate(gamma_values):
        svm = SVC(kernel='rbf', C=C, gamma=gamma, random_state=42)
        svm.fit(X_train_scaled, y_train)
        train_score = svm.score(X_train_scaled, y_train)
        test_score = svm.score(X_test_scaled, y_test)

        plt.subplot(4, 4, i*4 + j + 1)
        plt.title(f'C={C}, gamma={gamma}\nTreino={train_score:.2f}, Teste={test_score:.2f}')

        # Criar uma malha para visualizar as fronteiras de decisão (simplificado)
        if X.shape[1] > 2:
            # Usar PCA para reduzir para 2D se necessário
            from sklearn.decomposition import PCA
            pca = PCA(n_components=2)
            X_pca = pca.fit_transform(X)

            # Dividir em treino e teste
            X_pca_train, X_pca_test, y_train_pca, y_test_pca = train_test_split(
                X_pca, y, test_size=0.3, random_state=42)

            # Treinar um SVM 2D
            svm_2d = SVC(kernel='rbf', C=C, gamma=gamma, random_state=42)
            svm_2d.fit(X_pca_train, y_train_pca)

            # Criar uma malha
            h = 0.02
            x_min, x_max = X_pca[:, 0].min() - 1, X_pca[:, 0].max() + 1
            y_min, y_max = X_pca[:, 1].min() - 1, X_pca[:, 1].max() + 1
            xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

            # Fazer previsões
            Z = svm_2d.predict(np.c_[xx.ravel(), yy.ravel()])
            Z = Z.reshape(xx.shape)
```

```
            # Plotar
            plt.contourf(xx, yy, Z, alpha=0.3, cmap='viridis')
            for k, target in enumerate(target_names):
                plt.scatter(X_pca[y == k, 0], X_pca[y == k, 1], s=20, alpha=0.5, label=target)

            if i == 0 and j == 0:
                plt.legend()

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

6. Naive Bayes

Naive Bayes é um classificador probabilístico baseado no teorema de Bayes, com a "ingênua" suposição de independência entre as features.

Características: - Baseado no teorema de Bayes - Assume independência condicional entre features - Calcula a probabilidade de cada classe dado um conjunto de features

Vantagens: - Simples e rápido - Funciona bem com poucos dados de treinamento - Lida bem com features categóricas - Bom para classificação de texto e spam

Desvantagens: - Suposição de independência raramente é verdadeira - Sensível a features irrelevantes - Pode ser superado por outros algoritmos em tarefas complexas

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import seaborn as sns

# Carregar o dataset de vinhos
data = load_wine()
X = data.data
y = data.target
feature_names = data.feature_names
target_names = data.target_names

# Dividir em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Criar e treinar o modelo
model = GaussianNB()
model.fit(X_train, y_train)

# Fazer previsões
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)

# Avaliar o modelo
accuracy = accuracy_score(y_test, y_pred)
```

```
print(f"Acurácia: {accuracy:.4f}")

print("\nMatriz de Confusão:")
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=target_names, yticklabels=target_names)
plt.xlabel('Previsão')
plt.ylabel('Valor Real')
plt.title('Matriz de Confusão')
plt.show()

print("\nRelatório de Classificação:")
print(classification_report(y_test, y_pred, target_names=target_names))

# Visualizar as probabilidades para algumas amostras
n_samples = 5
plt.figure(figsize=(12, 6))
for i in range(n_samples):
    plt.subplot(1, n_samples, i+1)
    plt.bar(range(len(target_names)), y_prob[i])
    plt.xticks(range(len(target_names)), target_names, rotation=90)
    plt.title(f'Amostra {i+1}\nClasse Real: {target_names[y_test[i]]}\nPrevisão: {target_names[y_pred[i]
    plt.ylim(0, 1)
    plt.ylabel('Probabilidade')
plt.tight_layout()
plt.show()

# Visualizar a distribuição das features por classe
plt.figure(figsize=(15, 10))
for i, feature in enumerate(feature_names[:6]):  # Primeiras 6 features
    plt.subplot(2, 3, i+1)
    for target in range(len(target_names)):
        sns.kdeplot(X[y == target, i], label=target_names[target])
    plt.title(feature)
    plt.xlabel('Valor')
    plt.ylabel('Densidade')
    plt.legend()
plt.tight_layout()
plt.show()
```

Algoritmos de Regressão

1. Regressão Linear

A regressão linear é um dos algoritmos mais simples e amplamente utilizados para modelar a relação entre uma variável dependente e uma ou mais variáveis independentes.

Características: - Modelo linear que assume relação linear entre features e target - Minimiza a soma dos quadrados dos resíduos - Fácil interpretação através dos coeficientes

Vantagens: - Simples e interpretável - Rápido para treinar - Funciona bem quando a relação é aproximadamente linear - Base para muitos outros algoritmos

Desvantagens: - Assume linearidade - Sensível a outliers - Não captura relações não lineares - Assume

independência entre features

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
import seaborn as sns

# Carregar o dataset Boston Housing
data = load_boston()
X = data.data
y = data.target
feature_names = data.feature_names

# Criar um DataFrame para facilitar a visualização
df = pd.DataFrame(X, columns=feature_names)
df['PRICE'] = y

# Dividir em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Criar e treinar o modelo
model = LinearRegression()
model.fit(X_train, y_train)

# Fazer previsões
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)

# Avaliar o modelo
train_mse = mean_squared_error(y_train, y_train_pred)
test_mse = mean_squared_error(y_test, y_test_pred)
train_rmse = np.sqrt(train_mse)
test_rmse = np.sqrt(test_mse)
train_r2 = r2_score(y_train, y_train_pred)
test_r2 = r2_score(y_test, y_test_pred)

print("Avaliação do Modelo:")
print(f"RMSE (Treino): {train_rmse:.4f}")
print(f"RMSE (Teste): {test_rmse:.4f}")
print(f"R² (Treino): {train_r2:.4f}")
print(f"R² (Teste): {test_r2:.4f}")

# Visualizar os coeficientes
coef = pd.Series(model.coef_, index=feature_names)
coef = coef.sort_values(ascending=False)

plt.figure(figsize=(10, 6))
coef.plot(kind='bar')
plt.title('Coeficientes da Regressão Linear')
```

```
plt.xlabel('Feature')
plt.ylabel('Coeficiente')
plt.grid(True, axis='y')
plt.tight_layout()
plt.show()

# Visualizar previsões vs. valores reais
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_test_pred, alpha=0.7)
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--')
plt.xlabel('Valores Reais')
plt.ylabel('Previsões')
plt.title('Valores Reais vs. Previsões')
plt.grid(True)
plt.tight_layout()
plt.show()

# Visualizar os resíduos
residuos = y_test - y_test_pred

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
sns.histplot(residuos, kde=True)
plt.title('Distribuição dos Resíduos')
plt.xlabel('Resíduo')
plt.ylabel('Frequência')
plt.grid(True)

plt.subplot(1, 2, 2)
plt.scatter(y_test_pred, residuos, alpha=0.7)
plt.axhline(y=0, color='r', linestyle='--')
plt.title('Resíduos vs. Valores Previstos')
plt.xlabel('Valores Previstos')
plt.ylabel('Resíduos')
plt.grid(True)

plt.tight_layout()
plt.show()

# Regressão linear para uma única feature
feature_index = 5  # RM (número médio de quartos)
X_single = X[:, feature_index].reshape(-1, 1)
X_train_single, X_test_single, y_train_single, y_test_single = train_test_split(
    X_single, y, test_size=0.3, random_state=42)

model_single = LinearRegression()
model_single.fit(X_train_single, y_train_single)
y_pred_single = model_single.predict(X_test_single)

plt.figure(figsize=(10, 6))
plt.scatter(X_train_single, y_train_single, alpha=0.7, label='Treino')
plt.scatter(X_test_single, y_test_single, alpha=0.7, label='Teste')
plt.plot(X_test_single, y_pred_single, color='r', linewidth=2)
```

```
plt.xlabel(feature_names[feature_index])
plt.ylabel('Preço')
plt.title(f'Regressão Linear: {feature_names[feature_index]} vs. Preço')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

2. Regressão Polinomial

A regressão polinomial estende a regressão linear para capturar relações não lineares, adicionando termos polinomiais às features.

Características: - Extensão da regressão linear com termos polinomiais - Pode capturar relações não lineares - O grau do polinômio controla a complexidade do modelo

Vantagens: - Mais flexível que a regressão linear - Pode modelar curvas e padrões não lineares - Ainda relativamente interpretável

Desvantagens: - Propenso a overfitting com graus altos - Sensível a outliers - Pode ser instável em extrapolações

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score

# Gerar dados sintéticos não lineares
np.random.seed(42)
X = np.sort(5 * np.random.rand(100, 1), axis=0)
y = np.sin(X).ravel() + np.random.normal(0, 0.1, X.shape[0])

# Dividir em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Criar modelos com diferentes graus polinomiais
degrees = [1, 2, 3, 5, 10]
plt.figure(figsize=(14, 10))

for i, degree in enumerate(degrees):
    ax = plt.subplot(2, 3, i + 1)

    # Criar um modelo de regressão polinomial
    model = Pipeline([
        ('poly', PolynomialFeatures(degree=degree)),
        ('linear', LinearRegression())
    ])

    # Treinar o modelo
    model.fit(X_train, y_train)
```

```
    # Fazer previsões
    y_train_pred = model.predict(X_train)
    y_test_pred = model.predict(X_test)

    # Calcular o erro
    train_rmse = np.sqrt(mean_squared_error(y_train, y_train_pred))
    test_rmse = np.sqrt(mean_squared_error(y_test, y_test_pred))
    train_r2 = r2_score(y_train, y_train_pred)
    test_r2 = r2_score(y_test, y_test_pred)

    # Plotar os dados e o modelo
    X_plot = np.linspace(0, 5, 100).reshape(-1, 1)
    y_plot = model.predict(X_plot)

    plt.scatter(X_train, y_train, color='blue', s=30, alpha=0.5, label='Treino')
    plt.scatter(X_test, y_test, color='green', s=30, alpha=0.5, label='Teste')
    plt.plot(X_plot, y_plot, color='red', label='Modelo')
    plt.title(f'Polinômio de Grau {degree}\nTreino RMSE: {train_rmse:.4f}, R²: {train_r2:.4f}\nTeste RM
    plt.xlabel('x')
    plt.ylabel('y')
    plt.ylim(-1.5, 1.5)
    plt.legend()
    plt.grid(True)

plt.tight_layout()
plt.show()

# Comparar o desempenho dos modelos
train_rmse = []
test_rmse = []
train_r2 = []
test_r2 = []

for degree in range(1, 21):
    model = Pipeline([
        ('poly', PolynomialFeatures(degree=degree)),
        ('linear', LinearRegression())
    ])

    model.fit(X_train, y_train)

    y_train_pred = model.predict(X_train)
    y_test_pred = model.predict(X_test)

    train_rmse.append(np.sqrt(mean_squared_error(y_train, y_train_pred)))
    test_rmse.append(np.sqrt(mean_squared_error(y_test, y_test_pred)))
    train_r2.append(r2_score(y_train, y_train_pred))
    test_r2.append(r2_score(y_test, y_test_pred))

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(range(1, 21), train_rmse, 'o-', label='Treino')
plt.plot(range(1, 21), test_rmse, 'o-', label='Teste')
```

```
plt.xlabel('Grau do Polinômio')
plt.ylabel('RMSE')
plt.title('RMSE vs. Grau do Polinômio')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(range(1, 21), train_r2, 'o-', label='Treino')
plt.plot(range(1, 21), test_r2, 'o-', label='Teste')
plt.xlabel('Grau do Polinômio')
plt.ylabel('R²')
plt.title('R² vs. Grau do Polinômio')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

# Encontrar o melhor grau
best_degree = np.argmin(test_rmse) + 1
print(f"Melhor grau: {best_degree}")
print(f"Melhor RMSE de teste: {min(test_rmse):.4f}")
print(f"Melhor R² de teste: {test_r2[best_degree-1]:.4f}")
```

3. Regressão Ridge e Lasso (Regularização)

Ridge e Lasso são extensões da regressão linear que adicionam termos de regularização para evitar overfitting.

Características: - Ridge: Adiciona penalidade L2 (soma dos quadrados dos coeficientes) - Lasso: Adiciona penalidade L1 (soma dos valores absolutos dos coeficientes) - ElasticNet: Combina penalidades L1 e L2

Vantagens: - Reduz overfitting - Ridge: Reduz a magnitude dos coeficientes - Lasso: Pode zerar coeficientes (seleção de features) - ElasticNet: Combina os benefícios de Ridge e Lasso

Desvantagens: - Requer ajuste do parâmetro de regularização - Ridge não realiza seleção de features - Lasso pode ser instável quando as features são correlacionadas

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression, Ridge, Lasso, ElasticNet
from sklearn.metrics import mean_squared_error, r2_score
import seaborn as sns

# Carregar o dataset Boston Housing
data = load_boston()
X = data.data
y = data.target
feature_names = data.feature_names

# Dividir em treino e teste
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Padronizar os dados
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Criar e treinar os modelos
models = {
    'Linear Regression': LinearRegression(),
    'Ridge (alpha=1.0)': Ridge(alpha=1.0, random_state=42),
    'Lasso (alpha=0.1)': Lasso(alpha=0.1, random_state=42),
    'ElasticNet (alpha=0.1, l1_ratio=0.5)': ElasticNet(alpha=0.1, l1_ratio=0.5, random_state=42)
}

results = {}

for name, model in models.items():
    # Treinar o modelo
    model.fit(X_train_scaled, y_train)

    # Fazer previsões
    y_train_pred = model.predict(X_train_scaled)
    y_test_pred = model.predict(X_test_scaled)

    # Calcular métricas
    train_rmse = np.sqrt(mean_squared_error(y_train, y_train_pred))
    test_rmse = np.sqrt(mean_squared_error(y_test, y_test_pred))
    train_r2 = r2_score(y_train, y_train_pred)
    test_r2 = r2_score(y_test, y_test_pred)

    # Armazenar resultados
    results[name] = {
        'train_rmse': train_rmse,
        'test_rmse': test_rmse,
        'train_r2': train_r2,
        'test_r2': test_r2,
        'coef': model.coef_
    }

    print(f"\n{name}:")
    print(f"RMSE (Treino): {train_rmse:.4f}")
    print(f"RMSE (Teste): {test_rmse:.4f}")
    print(f"R² (Treino): {train_r2:.4f}")
    print(f"R² (Teste): {test_r2:.4f}")

# Comparar os coeficientes dos diferentes modelos
coef_df = pd.DataFrame({name: results[name]['coef'] for name in models.keys()},
                       index=feature_names)

plt.figure(figsize=(12, 10))
sns.heatmap(coef_df, annot=True, cmap='coolwarm', center=0, fmt='.2f')
plt.title('Comparação dos Coeficientes entre Modelos')
```

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

# Visualizar os coeficientes em um gráfico de barras
plt.figure(figsize=(12, 8))
for i, (name, result) in enumerate(results.items()):
    plt.subplot(2, 2, i+1)
    plt.bar(feature_names, result['coef'])
    plt.title(name)
    plt.xticks(rotation=90)
    plt.ylabel('Coeficiente')
    plt.grid(True, axis='y')
plt.tight_layout()
plt.show()

# Comparar o desempenho dos modelos
metrics = ['train_rmse', 'test_rmse', 'train_r2', 'test_r2']
labels = ['RMSE (Treino)', 'RMSE (Teste)', 'R² (Treino)', 'R² (Teste)']

plt.figure(figsize=(14, 10))
for i, (metric, label) in enumerate(zip(metrics, labels)):
    plt.subplot(2, 2, i+1)
    values = [results[name][metric] for name in models.keys()]
    plt.bar(models.keys(), values)
    plt.title(label)
    plt.xticks(rotation=45)
    plt.grid(True, axis='y')
plt.tight_layout()
plt.show()

# Testar diferentes valores de alpha para Ridge
alphas = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
ridge_results = {}

for alpha in alphas:
    model = Ridge(alpha=alpha, random_state=42)
    model.fit(X_train_scaled, y_train)

    y_train_pred = model.predict(X_train_scaled)
    y_test_pred = model.predict(X_test_scaled)

    train_rmse = np.sqrt(mean_squared_error(y_train, y_train_pred))
    test_rmse = np.sqrt(mean_squared_error(y_test, y_test_pred))

    ridge_results[alpha] = {
        'train_rmse': train_rmse,
        'test_rmse': test_rmse,
        'coef': model.coef_
    }

# Visualizar o efeito do alpha no RMSE
plt.figure(figsize=(10, 6))
plt.plot(alphas, [ridge_results[a]['train_rmse'] for a in alphas], 'o-', label='Treino')
```

```
plt.plot(alphas, [ridge_results[a]['test_rmse'] for a in alphas], 'o-', label='Teste')
plt.xscale('log')
plt.xlabel('Alpha (escala log)')
plt.ylabel('RMSE')
plt.title('RMSE vs. Alpha (Ridge)')
plt.legend()
plt.grid(True)
plt.show()

# Visualizar o efeito do alpha nos coeficientes
plt.figure(figsize=(12, 8))
for i, alpha in enumerate(alphas):
    plt.subplot(2, 3, i+1)
    plt.bar(feature_names, ridge_results[alpha]['coef'])
    plt.title(f'Alpha = {alpha}')
    plt.xticks(rotation=90)
    plt.ylabel('Coeficiente')
    plt.grid(True, axis='y')
plt.tight_layout()
plt.show()
```

4. Árvores de Decisão para Regressão

As árvores de decisão também podem ser usadas para problemas de regressão, dividindo o espaço de features em regiões e atribuindo um valor constante a cada região.

Características: - Estrutura hierárquica de decisões - Divisões baseadas em regras simples - Previsão baseada na média dos valores alvo em cada folha

Vantagens: - Captura relações não lineares - Não requer normalização dos dados - Fácil interpretação - Lida bem com features categóricas e numéricas

Desvantagens: - Tendência a overfitting - Instabilidade - Previsões em degraus (não suaves)

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor, plot_tree
from sklearn.metrics import mean_squared_error, r2_score
import seaborn as sns

# Carregar o dataset Boston Housing
data = load_boston()
X = data.data
y = data.target
feature_names = data.feature_names

# Dividir em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Criar e treinar o modelo
model = DecisionTreeRegressor(max_depth=3, random_state=42)
```

```
model.fit(X_train, y_train)

# Fazer previsões
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)

# Avaliar o modelo
train_rmse = np.sqrt(mean_squared_error(y_train, y_train_pred))
test_rmse = np.sqrt(mean_squared_error(y_test, y_test_pred))
train_r2 = r2_score(y_train, y_train_pred)
test_r2 = r2_score(y_test, y_test_pred)

print("Avaliação do Modelo:")
print(f"RMSE (Treino): {train_rmse:.4f}")
print(f"RMSE (Teste): {test_rmse:.4f}")
print(f"R² (Treino): {train_r2:.4f}")
print(f"R² (Teste): {test_r2:.4f}")

# Visualizar a árvore de decisão
plt.figure(figsize=(20, 10))
plot_tree(model, filled=True, feature_names=feature_names, rounded=True)
plt.title('Árvore de Decisão para Regressão')
plt.show()

# Importância das features
importances = model.feature_importances_
indices = np.argsort(importances)[::-1]

plt.figure(figsize=(10, 6))
plt.bar(range(X.shape[1]), importances[indices])
plt.xticks(range(X.shape[1]), [feature_names[i] for i in indices], rotation=90)
plt.xlabel('Feature')
plt.ylabel('Importância')
plt.title('Importância das Features na Árvore de Decisão')
plt.grid(True, axis='y')
plt.tight_layout()
plt.show()

# Visualizar previsões vs. valores reais
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_test_pred, alpha=0.7)
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--')
plt.xlabel('Valores Reais')
plt.ylabel('Previsões')
plt.title('Valores Reais vs. Previsões')
plt.grid(True)
plt.tight_layout()
plt.show()

# Visualizar os resíduos
residuos = y_test - y_test_pred

plt.figure(figsize=(12, 5))
```

```
plt.subplot(1, 2, 1)
sns.histplot(residuos, kde=True)
plt.title('Distribuição dos Resíduos')
plt.xlabel('Resíduo')
plt.ylabel('Frequência')
plt.grid(True)

plt.subplot(1, 2, 2)
plt.scatter(y_test_pred, residuos, alpha=0.7)
plt.axhline(y=0, color='r', linestyle='--')
plt.title('Resíduos vs. Valores Previstos')
plt.xlabel('Valores Previstos')
plt.ylabel('Resíduos')
plt.grid(True)

plt.tight_layout()
plt.show()

# Testar diferentes profundidades da árvore
max_depths = range(1, 21)
train_rmse_list = []
test_rmse_list = []
train_r2_list = []
test_r2_list = []

for depth in max_depths:
    model = DecisionTreeRegressor(max_depth=depth, random_state=42)
    model.fit(X_train, y_train)

    y_train_pred = model.predict(X_train)
    y_test_pred = model.predict(X_test)

    train_rmse_list.append(np.sqrt(mean_squared_error(y_train, y_train_pred)))
    test_rmse_list.append(np.sqrt(mean_squared_error(y_test, y_test_pred)))
    train_r2_list.append(r2_score(y_train, y_train_pred))
    test_r2_list.append(r2_score(y_test, y_test_pred))

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(max_depths, train_rmse_list, 'o-', label='Treino')
plt.plot(max_depths, test_rmse_list, 'o-', label='Teste')
plt.xlabel('Profundidade Máxima')
plt.ylabel('RMSE')
plt.title('RMSE vs. Profundidade da Árvore')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(max_depths, train_r2_list, 'o-', label='Treino')
plt.plot(max_depths, test_r2_list, 'o-', label='Teste')
plt.xlabel('Profundidade Máxima')
plt.ylabel('R²')
plt.title('R² vs. Profundidade da Árvore')
```

```
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

# Encontrar a melhor profundidade
best_depth = max_depths[np.argmax(test_r2_list)]
print(f"Melhor profundidade: {best_depth}")
print(f"Melhor RMSE de teste: {test_rmse_list[best_depth-1]:.4f}")
print(f"Melhor R² de teste: {test_r2_list[best_depth-1]:.4f}")
```

5. Random Forest para Regressão

Random Forest também pode ser usado para problemas de regressão, combinando múltiplas árvores de decisão.

Características: - Ensemble de árvores de decisão - Cada árvore é treinada em um subconjunto aleatório dos dados - A previsão final é a média das previsões das árvores individuais

Vantagens: - Maior precisão que árvores individuais - Menos propenso a overfitting - Robusto a outliers e ruído - Captura relações não lineares

Desvantagens: - Menos interpretável que uma única árvore - Computacionalmente mais intensivo - Previsões menos suaves que outros métodos

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score
import seaborn as sns

# Carregar o dataset Boston Housing
data = load_boston()
X = data.data
y = data.target
feature_names = data.feature_names

# Dividir em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Criar e treinar o modelo
model = RandomForestRegressor(n_estimators=100, max_depth=5, random_state=42)
model.fit(X_train, y_train)

# Fazer previsões
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)

# Avaliar o modelo
train_rmse = np.sqrt(mean_squared_error(y_train, y_train_pred))
```

```
test_rmse = np.sqrt(mean_squared_error(y_test, y_test_pred))
train_r2 = r2_score(y_train, y_train_pred)
test_r2 = r2_score(y_test, y_test_pred)

print("Avaliação do Modelo:")
print(f"RMSE (Treino): {train_rmse:.4f}")
print(f"RMSE (Teste): {test_rmse:.4f}")
print(f"R² (Treino): {train_r2:.4f}")
print(f"R² (Teste): {test_r2:.4f}")

# Importância das features
importances = model.feature_importances_
indices = np.argsort(importances)[::-1]

plt.figure(figsize=(10, 6))
plt.bar(range(X.shape[1]), importances[indices])
plt.xticks(range(X.shape[1]), [feature_names[i] for i in indices], rotation=90)
plt.xlabel('Feature')
plt.ylabel('Importância')
plt.title('Importância das Features no Random Forest')
plt.grid(True, axis='y')
plt.tight_layout()
plt.show()

# Visualizar previsões vs. valores reais
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_test_pred, alpha=0.7)
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--')
plt.xlabel('Valores Reais')
plt.ylabel('Previsões')
plt.title('Valores Reais vs. Previsões')
plt.grid(True)
plt.tight_layout()
plt.show()

# Visualizar os resíduos
residuos = y_test - y_test_pred

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
sns.histplot(residuos, kde=True)
plt.title('Distribuição dos Resíduos')
plt.xlabel('Resíduo')
plt.ylabel('Frequência')
plt.grid(True)

plt.subplot(1, 2, 2)
plt.scatter(y_test_pred, residuos, alpha=0.7)
plt.axhline(y=0, color='r', linestyle='--')
plt.title('Resíduos vs. Valores Previstos')
plt.xlabel('Valores Previstos')
plt.ylabel('Resíduos')
plt.grid(True)
```

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

# Testar diferentes números de árvores e profundidades
n_estimators_list = [10, 50, 100, 200]
max_depths_list = [None, 3, 5, 10, 15]

results = np.zeros((len(n_estimators_list), len(max_depths_list)))

for i, n_estimators in enumerate(n_estimators_list):
    for j, max_depth in enumerate(max_depths_list):
        model = RandomForestRegressor(n_estimators=n_estimators, max_depth=max_depth, random_state=42)
        model.fit(X_train, y_train)

        y_test_pred = model.predict(X_test)
        test_r2 = r2_score(y_test, y_test_pred)

        results[i, j] = test_r2

# Visualizar os resultados
plt.figure(figsize=(10, 6))
sns.heatmap(results, annot=True, cmap='viridis', fmt='.4f',
            xticklabels=max_depths_list, yticklabels=n_estimators_list)
plt.xlabel('Profundidade Máxima')
plt.ylabel('Número de Árvores')
plt.title('R² para Diferentes Configurações do Random Forest')
plt.tight_layout()
plt.show()

# Encontrar a melhor configuração
best_i, best_j = np.unravel_index(np.argmax(results), results.shape)
best_n_estimators = n_estimators_list[best_i]
best_max_depth = max_depths_list[best_j]
best_r2 = results[best_i, best_j]

print(f"Melhor configuração: n_estimators={best_n_estimators}, max_depth={best_max_depth}")
print(f"Melhor R² de teste: {best_r2:.4f}")
```

Comparação de Algoritmos

Ao escolher um algoritmo para um problema específico, é importante considerar vários fatores:

1. **Natureza dos dados**: Tamanho, dimensionalidade, tipos de features, presença de outliers, etc.
2. **Complexidade do problema**: Linear vs. não linear, número de classes, balanceamento, etc.
3. **Interpretabilidade**: Necessidade de entender como o modelo toma decisões.
4. **Desempenho**: Precisão, velocidade de treinamento e inferência.
5. **Recursos computacionais**: Memória, CPU, GPU disponíveis.

Vamos comparar os algoritmos que discutimos:

Algoritmo	Tipo	Interpretabilidade	Velocidade de Treinamento	Velocidade de Inferência	Captura Relações Não Lineares	Robusto a Outliers	Robusto a Features Irrelevantes	Escalabilidade
Regressão Logística	Classificação	Alta	Rápida	Rápida	Não	Não	Não	Boa
k-NN	Classificação/Regressão	Média	Instantânea	Lenta	Sim	Não	Não	Ruim
Árvore de Decisão	Classificação/Regressão	Alta	Média	Rápida	Sim	Média	Média	Média
Random Forest	Classificação/Regressão	Baixa	Lenta	Média	Sim	Alta	Alta	Média
SVM	Classificação/Regressão	Baixa	Lenta	Média	Sim (com kernels)	Média	Média	Ruim
Naive Bayes	Classificação	Alta	Rápida	Rápida	Não	Não	Não	Boa
Regressão Linear	Regressão	Alta	Rápida	Rápida	Não	Não	Não	Boa
Regressão Polinomial	Regressão	Média	Média	Rápida	Sim	Não	Não	Média
Ridge/Lasso	Regressão	Alta	Rápida	Rápida	Não	Média	Alta (Lasso)	Boa

Exemplo de Comparação de Algoritmos

Vamos implementar um exemplo que compara vários algoritmos de classificação em um mesmo dataset:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score
import seaborn as sns

# Carregar o dataset de câncer de mama
data = load_breast_cancer()
X = data.data
y = data.target
feature_names = data.feature_names
target_names = data.target_names

# Dividir em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Padronizar os dados
scaler = StandardScaler()
```

```
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Criar os modelos
models = {
    'Logistic Regression': LogisticRegression(max_iter=1000, random_state=42),
    'k-NN': KNeighborsClassifier(n_neighbors=5),
    'Decision Tree': DecisionTreeClassifier(max_depth=5, random_state=42),
    'Random Forest': RandomForestClassifier(n_estimators=100, max_depth=5, random_state=42),
    'SVM': SVC(kernel='rbf', C=1.0, gamma='scale', probability=True, random_state=42),
    'Naive Bayes': GaussianNB()
}

# Treinar e avaliar cada modelo
results = {}

for name, model in models.items():
    # Treinar o modelo
    model.fit(X_train_scaled, y_train)

    # Fazer previsões
    y_pred = model.predict(X_test_scaled)

    # Calcular probabilidades (se disponível)
    if hasattr(model, "predict_proba"):
        y_prob = model.predict_proba(X_test_scaled)[:, 1]
    else:
        y_prob = y_pred  # Para modelos que não fornecem probabilidades

    # Calcular métricas
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
    roc_auc = roc_auc_score(y_test, y_prob)

    # Validação cruzada
    cv_scores = cross_val_score(model, X_train_scaled, y_train, cv=5)

    # Armazenar resultados
    results[name] = {
        'accuracy': accuracy,
        'precision': precision,
        'recall': recall,
        'f1': f1,
        'roc_auc': roc_auc,
        'cv_mean': cv_scores.mean(),
        'cv_std': cv_scores.std(),
        'y_pred': y_pred,
        'y_prob': y_prob
    }

    print(f"\n{name}:")
```

```
    print(f"Acurácia: {accuracy:.4f}")
    print(f"Precisão: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1-Score: {f1:.4f}")
    print(f"ROC AUC: {roc_auc:.4f}")
    print(f"Validação Cruzada (5-fold): {cv_scores.mean():.4f} ± {cv_scores.std():.4f}")

# Comparar os modelos
metrics = ['accuracy', 'precision', 'recall', 'f1', 'roc_auc']
labels = ['Acurácia', 'Precisão', 'Recall', 'F1-Score', 'ROC AUC']

plt.figure(figsize=(15, 10))
for i, (metric, label) in enumerate(zip(metrics, labels)):
    plt.subplot(2, 3, i+1)
    values = [results[name][metric] for name in models.keys()]
    plt.bar(models.keys(), values)
    plt.title(label)
    plt.xticks(rotation=45)
    plt.ylim(0.8, 1.0)  # Ajustar conforme necessário
    plt.grid(True, axis='y')

plt.subplot(2, 3, 6)
cv_means = [results[name]['cv_mean'] for name in models.keys()]
cv_stds = [results[name]['cv_std'] for name in models.keys()]
plt.bar(models.keys(), cv_means, yerr=cv_stds)
plt.title('Validação Cruzada (5-fold)')
plt.xticks(rotation=45)
plt.ylim(0.8, 1.0)  # Ajustar conforme necessário
plt.grid(True, axis='y')

plt.tight_layout()
plt.show()

# Curvas ROC para cada modelo
plt.figure(figsize=(10, 8))
for name, result in results.items():
    fpr, tpr, _ = roc_curve(y_test, result['y_prob'])
    plt.plot(fpr, tpr, label=f"{name} (AUC = {result['roc_auc']:.4f})")

plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('Taxa de Falsos Positivos')
plt.ylabel('Taxa de Verdadeiros Positivos')
plt.title('Curvas ROC')
plt.legend()
plt.grid(True)
plt.show()

# Matriz de confusão para o melhor modelo
best_model_name = max(results, key=lambda x: results[x]['f1'])
best_model_pred = results[best_model_name]['y_pred']

plt.figure(figsize=(8, 6))
cm = confusion_matrix(y_test, best_model_pred)
```

```
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=target_names, yticklabels=target_names)
plt.xlabel('Previsão')
plt.ylabel('Valor Real')
plt.title(f'Matriz de Confusão - {best_model_name}')
plt.tight_layout()
plt.show()

print(f"\nMelhor modelo (F1-Score): {best_model_name}")
print(f"F1-Score: {results[best_model_name]['f1']:.4f}")
```

Conclusão

Neste tópico, exploramos os principais algoritmos de aprendizado supervisionado para classificação e regressão. Cada algoritmo tem suas próprias características, vantagens e desvantagens, e a escolha do algoritmo adequado depende do problema específico e dos dados disponíveis.

Alguns pontos importantes a lembrar:

1. **Não existe um algoritmo perfeito para todos os problemas** (No Free Lunch Theorem).
2. **Experimentação é fundamental**: Teste diferentes algoritmos e configurações.
3. **Pré-processamento adequado dos dados** pode melhorar significativamente o desempenho.
4. **Validação cruzada** é essencial para avaliar a capacidade de generalização do modelo.
5. **Interpretabilidade vs. Desempenho**: Às vezes, é necessário sacrificar um pelo outro.

No próximo tópico, exploraremos algoritmos de aprendizado não supervisionado, que são usados quando não temos rótulos para os dados e queremos descobrir padrões ou estruturas ocultas.

Módulo 4: Introdução à Ciência de Dados e Machine Learning

Algoritmos de Aprendizado Não Supervisionado

O aprendizado não supervisionado é um paradigma de machine learning onde o algoritmo aprende a partir de dados não rotulados, ou seja, exemplos onde não há uma resposta correta ou saída desejada. O objetivo é descobrir padrões, estruturas ou relações ocultas nos dados. Neste tópico, vamos explorar os principais algoritmos de aprendizado não supervisionado, suas características, vantagens, desvantagens e aplicações.

Tipos de Problemas de Aprendizado Não Supervisionado

Os problemas de aprendizado não supervisionado podem ser divididos em várias categorias:

1. **Clustering (Agrupamento)**: Agrupar dados similares em clusters.
2. **Redução de Dimensionalidade**: Reduzir o número de variáveis mantendo a informação essencial.
3. **Detecção de Anomalias**: Identificar observações que diferem significativamente do padrão.
4. **Regras de Associação**: Descobrir relações entre variáveis em grandes conjuntos de dados.
5. **Geração de Dados**: Criar novos dados semelhantes aos dados de treinamento.

Vamos explorar os algoritmos mais comuns para cada tipo de problema.

Algoritmos de Clustering

1. K-Means

K-Means é um dos algoritmos de clustering mais simples e populares, que divide os dados em K clusters, onde cada observação pertence ao cluster com a média mais próxima.

Características: - Divide os dados em K clusters predefinidos - Iterativamente atribui pontos ao centroide mais próximo e recalcula os centroides - Minimiza a soma dos quadrados das distâncias dentro de cada cluster

Vantagens: - Simples e fácil de implementar - Escalável para grandes conjuntos de dados - Funciona bem com clusters esféricos e de tamanhos similares

Desvantagens: - Requer especificar o número de clusters (K) antecipadamente - Sensível à inicialização dos centroides - Não lida bem com clusters de formas não esféricas - Sensível a outliers

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
```

```
import seaborn as sns

# Gerar dados sintéticos
n_samples = 1000
n_features = 2
n_clusters = 3
random_state = 42

X, y_true = make_blobs(n_samples=n_samples, n_features=n_features, centers=n_clusters,
                       cluster_std=0.7, random_state=random_state)

# Visualizar os dados
plt.figure(figsize=(10, 6))
plt.scatter(X[:, 0], X[:, 1], s=50, alpha=0.7)
plt.title('Dados Sintéticos')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)
plt.show()

# Aplicar K-Means
kmeans = KMeans(n_clusters=n_clusters, random_state=random_state)
y_pred = kmeans.fit_predict(X)
centroids = kmeans.cluster_centers_

# Visualizar os clusters
plt.figure(figsize=(10, 6))
plt.scatter(X[:, 0], X[:, 1], c=y_pred, s=50, alpha=0.7, cmap='viridis')
plt.scatter(centroids[:, 0], centroids[:, 1], c='red', s=200, marker='X')
plt.title('Clusters K-Means')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)
plt.show()

# Avaliar o clustering usando o coeficiente de silhueta
silhouette_avg = silhouette_score(X, y_pred)
print(f"Coeficiente de Silhueta: {silhouette_avg:.4f}")

# Encontrar o número ideal de clusters usando o método do cotovelo
inertia = []
silhouette_scores = []
k_range = range(2, 11)

for k in k_range:
    kmeans = KMeans(n_clusters=k, random_state=random_state)
    y_pred = kmeans.fit_predict(X)
    inertia.append(kmeans.inertia_)
    silhouette_scores.append(silhouette_score(X, y_pred))

# Plotar o método do cotovelo
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
```

```
plt.plot(k_range, inertia, 'o-')
plt.xlabel('Número de Clusters (k)')
plt.ylabel('Inércia')
plt.title('Método do Cotovelo')
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(k_range, silhouette_scores, 'o-')
plt.xlabel('Número de Clusters (k)')
plt.ylabel('Coeficiente de Silhueta')
plt.title('Coeficiente de Silhueta vs. Número de Clusters')
plt.grid(True)

plt.tight_layout()
plt.show()

# Visualizar os clusters para diferentes valores de k
plt.figure(figsize=(15, 10))
for i, k in enumerate([2, 3, 4, 5]):
    plt.subplot(2, 2, i+1)
    kmeans = KMeans(n_clusters=k, random_state=random_state)
    y_pred = kmeans.fit_predict(X)
    centroids = kmeans.cluster_centers_

    plt.scatter(X[:, 0], X[:, 1], c=y_pred, s=50, alpha=0.7, cmap='viridis')
    plt.scatter(centroids[:, 0], centroids[:, 1], c='red', s=200, marker='X')
    plt.title(f'K-Means com k={k}')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.grid(True)

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

2. DBSCAN (Density-Based Spatial Clustering of Applications with Noise)

DBSCAN é um algoritmo de clustering baseado em densidade que agrupa pontos que estão próximos uns dos outros e marca pontos em regiões de baixa densidade como outliers.

Características: - Baseado em densidade (regiões densas de pontos formam clusters) - Não requer especificar o número de clusters antecipadamente - Identifica outliers (ruído) automaticamente

Vantagens: - Pode descobrir clusters de formas arbitrárias - Robusto a outliers - Não requer especificar o número de clusters - Funciona bem com clusters de densidades similares

Desvantagens: - Sensível aos parâmetros eps (raio de vizinhança) e min_samples - Dificuldade com clusters de densidades muito diferentes - Não funciona bem em espaços de alta dimensionalidade

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons, make_circles
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.metrics import silhouette_score
import seaborn as sns

# Gerar dados sintéticos não esféricos
n_samples = 1000
random_state = 42

# Conjunto 1: Duas luas
X_moons, _ = make_moons(n_samples=n_samples, noise=0.1, random_state=random_state)

# Conjunto 2: Círculos concêntricos
X_circles, _ = make_circles(n_samples=n_samples, noise=0.1, factor=0.5, random_state=random_state)

# Visualizar os dados
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.scatter(X_moons[:, 0], X_moons[:, 1], s=50, alpha=0.7)
plt.title('Dados em Forma de Lua')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)

plt.subplot(1, 2, 2)
plt.scatter(X_circles[:, 0], X_circles[:, 1], s=50, alpha=0.7)
plt.title('Dados em Forma de Círculo')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)

plt.tight_layout()
plt.show()

# Aplicar DBSCAN aos dados em forma de lua
dbscan_moons = DBSCAN(eps=0.3, min_samples=5)
y_pred_moons = dbscan_moons.fit_predict(X_moons)

# Aplicar DBSCAN aos dados em forma de círculo
dbscan_circles = DBSCAN(eps=0.2, min_samples=5)
y_pred_circles = dbscan_circles.fit_predict(X_circles)

# Visualizar os clusters
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.scatter(X_moons[:, 0], X_moons[:, 1], c=y_pred_moons, s=50, alpha=0.7, cmap='viridis')
plt.title('Clusters DBSCAN (Luas)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)

plt.subplot(1, 2, 2)
plt.scatter(X_circles[:, 0], X_circles[:, 1], c=y_pred_circles, s=50, alpha=0.7, cmap='viridis')
plt.title('Clusters DBSCAN (Círculos)')
plt.xlabel('Feature 1')
```

```
plt.ylabel('Feature 2')
plt.grid(True)

plt.tight_layout()
plt.show()

# Contar o número de clusters e outliers
n_clusters_moons = len(set(y_pred_moons)) - (1 if -1 in y_pred_moons else 0)
n_outliers_moons = list(y_pred_moons).count(-1)

n_clusters_circles = len(set(y_pred_circles)) - (1 if -1 in y_pred_circles else 0)
n_outliers_circles = list(y_pred_circles).count(-1)

print(f"Dados em Forma de Lua:")
print(f"Número de clusters: {n_clusters_moons}")
print(f"Número de outliers: {n_outliers_moons}")

print(f"\nDados em Forma de Círculo:")
print(f"Número de clusters: {n_clusters_circles}")
print(f"Número de outliers: {n_outliers_circles}")

# Testar diferentes valores de eps e min_samples
eps_values = [0.1, 0.2, 0.3, 0.4, 0.5]
min_samples_values = [3, 5, 10, 20]

plt.figure(figsize=(15, 10))
for i, eps in enumerate(eps_values):
    for j, min_samples in enumerate(min_samples_values):
        plt.subplot(len(eps_values), len(min_samples_values), i*len(min_samples_values) + j + 1)

        dbscan = DBSCAN(eps=eps, min_samples=min_samples)
        y_pred = dbscan.fit_predict(X_moons)

        plt.scatter(X_moons[:, 0], X_moons[:, 1], c=y_pred, s=30, alpha=0.7, cmap='viridis')
        plt.title(f'eps={eps}, min_samples={min_samples}')
        plt.xticks([])
        plt.yticks([])

        n_clusters = len(set(y_pred)) - (1 if -1 in y_pred else 0)
        n_outliers = list(y_pred).count(-1)
        plt.xlabel(f'Clusters: {n_clusters}, Outliers: {n_outliers}')

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

3. Hierarchical Clustering (Agglomerative)

O clustering hierárquico aglomerativo constrói uma hierarquia de clusters de baixo para cima, começando com cada ponto como um cluster separado e, em seguida, mesclando os clusters mais próximos.

Características: - Constrói uma hierarquia de clusters (dendrograma) - Não requer especificar o número de clusters antecipadamente - Diferentes métricas de distância e critérios de ligação

Vantagens: - Fornece uma visão hierárquica dos dados - Não requer especificar o número de clusters a priori

- Flexível com diferentes métricas de distância - Pode capturar clusters de formas não esféricas

Desvantagens: - Computacionalmente intensivo para grandes conjuntos de dados - Sensível a outliers - Dificuldade em escolher o ponto de corte no dendrograma

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.cluster import AgglomerativeClustering
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.metrics import silhouette_score
import seaborn as sns

# Gerar dados sintéticos
n_samples = 150
n_features = 2
n_clusters = 3
random_state = 42

X, y_true = make_blobs(n_samples=n_samples, n_features=n_features, centers=n_clusters,
                       cluster_std=0.7, random_state=random_state)

# Visualizar os dados
plt.figure(figsize=(10, 6))
plt.scatter(X[:, 0], X[:, 1], s=50, alpha=0.7)
plt.title('Dados Sintéticos')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)
plt.show()

# Aplicar Hierarchical Clustering
agg_clustering = AgglomerativeClustering(n_clusters=n_clusters)
y_pred = agg_clustering.fit_predict(X)

# Visualizar os clusters
plt.figure(figsize=(10, 6))
plt.scatter(X[:, 0], X[:, 1], c=y_pred, s=50, alpha=0.7, cmap='viridis')
plt.title('Clusters Hierárquicos')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)
plt.show()

# Criar o dendrograma
plt.figure(figsize=(12, 8))
Z = linkage(X, method='ward')
dendrogram(Z, truncate_mode='level', p=5)
plt.title('Dendrograma do Clustering Hierárquico')
plt.xlabel('Amostras')
plt.ylabel('Distância')
plt.axhline(y=6, color='r', linestyle='--')  # Linha de corte para 3 clusters
```

```
plt.show()

# Testar diferentes critérios de ligação
linkage_methods = ['ward', 'complete', 'average', 'single']

plt.figure(figsize=(15, 10))
for i, method in enumerate(linkage_methods):
    plt.subplot(2, 2, i+1)

    agg_clustering = AgglomerativeClustering(n_clusters=n_clusters, linkage=method)
    y_pred = agg_clustering.fit_predict(X)

    plt.scatter(X[:, 0], X[:, 1], c=y_pred, s=50, alpha=0.7, cmap='viridis')
    plt.title(f'Linkage: {method}')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.grid(True)

    silhouette_avg = silhouette_score(X, y_pred)
    plt.text(0.05, 0.95, f'Silhouette: {silhouette_avg:.4f}',
             transform=plt.gca().transAxes, fontsize=12,
             verticalalignment='top')

plt.tight_layout()
plt.show()

# Testar diferentes números de clusters
plt.figure(figsize=(15, 10))
silhouette_scores = []
for i, n in enumerate(range(2, 6)):
    plt.subplot(2, 2, i+1)

    agg_clustering = AgglomerativeClustering(n_clusters=n)
    y_pred = agg_clustering.fit_predict(X)

    plt.scatter(X[:, 0], X[:, 1], c=y_pred, s=50, alpha=0.7, cmap='viridis')
    plt.title(f'Número de Clusters: {n}')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.grid(True)

    silhouette_avg = silhouette_score(X, y_pred)
    silhouette_scores.append(silhouette_avg)
    plt.text(0.05, 0.95, f'Silhouette: {silhouette_avg:.4f}',
             transform=plt.gca().transAxes, fontsize=12,
             verticalalignment='top')

plt.tight_layout()
plt.show()

# Plotar o coeficiente de silhueta para diferentes números de clusters
plt.figure(figsize=(10, 6))
plt.plot(range(2, 6), silhouette_scores, 'o-')
```

```
plt.xlabel('Número de Clusters')
plt.ylabel('Coeficiente de Silhueta')
plt.title('Coeficiente de Silhueta vs. Número de Clusters')
plt.grid(True)
plt.show()
```

4. Gaussian Mixture Models (GMM)

GMM é um modelo probabilístico que assume que os dados são gerados a partir de uma mistura de várias distribuições gaussianas.

Características: - Modelo probabilístico baseado em distribuições gaussianas - Cada cluster é representado por uma distribuição gaussiana - Usa o algoritmo EM (Expectation-Maximization) para estimar os parâmetros

Vantagens: - Fornece probabilidades de pertencimento a cada cluster - Flexível para modelar clusters de diferentes formas e tamanhos - Pode capturar correlações entre features

Desvantagens: - Sensível à inicialização - Requer especificar o número de componentes (clusters) - Assume que os clusters têm forma gaussiana - Pode convergir para máximos locais

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.mixture import GaussianMixture
from matplotlib.patches import Ellipse
import seaborn as sns

# Gerar dados sintéticos
n_samples = 1000
n_features = 2
n_clusters = 3
random_state = 42

X, y_true = make_blobs(n_samples=n_samples, n_features=n_features, centers=n_clusters,
                       cluster_std=[1.0, 2.0, 0.5], random_state=random_state)

# Adicionar rotação para tornar os clusters mais elípticos
transformation = [[0.6, -0.6], [-0.4, 0.8]]
X = np.dot(X, transformation)

# Visualizar os dados
plt.figure(figsize=(10, 6))
plt.scatter(X[:, 0], X[:, 1], s=50, alpha=0.7)
plt.title('Dados Sintéticos')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)
plt.show()

# Aplicar GMM
gmm = GaussianMixture(n_components=n_clusters, random_state=random_state)
gmm.fit(X)
```

```
y_pred = gmm.predict(X)
probs = gmm.predict_proba(X)

# Visualizar os clusters
plt.figure(figsize=(10, 6))
plt.scatter(X[:, 0], X[:, 1], c=y_pred, s=50, alpha=0.7, cmap='viridis')
plt.title('Clusters GMM')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)

# Plotar as elipses que representam as gaussianas
def draw_ellipse(position, covariance, ax=None, **kwargs):
    """Desenha uma elipse com base na posição e covariância."""
    ax = ax or plt.gca()

    # Converter covariância para ângulo e largura/altura
    v, w = np.linalg.eigh(covariance)
    u = w[0] / np.linalg.norm(w[0])
    angle = np.arctan2(u[1], u[0])
    angle = 180 * angle / np.pi  # Converter para graus

    # Largura e altura são "sigma" vezes 2
    v = 2. * np.sqrt(2.) * np.sqrt(v)
    ellipse = Ellipse(position, v[0], v[1], 180 + angle, **kwargs)

    return ax.add_patch(ellipse)

for pos, covar, alpha in zip(gmm.means_, gmm.covariances_, gmm.weights_):
    draw_ellipse(pos, covar, alpha=0.5, color='black', fill=False, linewidth=2)

plt.show()

# Visualizar as probabilidades de pertencimento
plt.figure(figsize=(12, 4))
for i in range(n_clusters):
    plt.subplot(1, n_clusters, i+1)
    plt.scatter(X[:, 0], X[:, 1], c=probs[:, i], s=50, alpha=0.7, cmap='viridis')
    plt.title(f'Probabilidade de Pertencer ao Cluster {i+1}')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.colorbar()
    plt.grid(True)

plt.tight_layout()
plt.show()

# Testar diferentes números de componentes
bic_scores = []
aic_scores = []
silhouette_scores = []
n_components_range = range(1, 10)
```

```
for n_components in n_components_range:
    gmm = GaussianMixture(n_components=n_components, random_state=random_state)
    gmm.fit(X)
    y_pred = gmm.predict(X)

    bic_scores.append(gmm.bic(X))
    aic_scores.append(gmm.aic(X))

    if n_components > 1:  # Silhouette score requer pelo menos 2 clusters
        silhouette_scores.append(silhouette_score(X, y_pred))
    else:
        silhouette_scores.append(0)

# Plotar os scores
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(n_components_range, bic_scores, 'o-', label='BIC')
plt.plot(n_components_range, aic_scores, 'o-', label='AIC')
plt.xlabel('Número de Componentes')
plt.ylabel('Score')
plt.title('BIC e AIC vs. Número de Componentes')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(n_components_range[1:], silhouette_scores[1:], 'o-')
plt.xlabel('Número de Componentes')
plt.ylabel('Coeficiente de Silhueta')
plt.title('Coeficiente de Silhueta vs. Número de Componentes')
plt.grid(True)

plt.tight_layout()
plt.show()

# Visualizar os clusters para diferentes números de componentes
plt.figure(figsize=(15, 10))
for i, n in enumerate([2, 3, 4, 5]):
    plt.subplot(2, 2, i+1)

    gmm = GaussianMixture(n_components=n, random_state=random_state)
    gmm.fit(X)
    y_pred = gmm.predict(X)

    plt.scatter(X[:, 0], X[:, 1], c=y_pred, s=50, alpha=0.7, cmap='viridis')
    plt.title(f'Número de Componentes: {n}')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.grid(True)

    # Plotar as elipses
    for pos, covar, alpha in zip(gmm.means_, gmm.covariances_, gmm.weights_):
        draw_ellipse(pos, covar, alpha=0.5, color='black', fill=False, linewidth=2)
```

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

Algoritmos de Redução de Dimensionalidade

1. Principal Component Analysis (PCA)

PCA é uma técnica de redução de dimensionalidade que transforma os dados para um novo sistema de coordenadas, onde as novas variáveis (componentes principais) são combinações lineares das variáveis originais e são ordenadas pela quantidade de variância que explicam.

Características: - Transforma os dados para um novo sistema de coordenadas - Componentes principais são ortogonais entre si - Maximiza a variância ao longo de cada componente

Vantagens: - Reduz a dimensionalidade dos dados - Remove correlações entre variáveis - Útil para visualização de dados de alta dimensionalidade - Pode ajudar a reduzir o ruído

Desvantagens: - Assume relações lineares entre variáveis - Sensível a outliers - Difícil interpretação dos componentes principais - Pode perder informações importantes se os dados não forem bem representados por relações lineares

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import seaborn as sns

# Carregar o dataset Iris
iris = load_iris()
X = iris.data
y = iris.target
feature_names = iris.feature_names
target_names = iris.target_names

# Padronizar os dados
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Aplicar PCA
pca = PCA()
X_pca = pca.fit_transform(X_scaled)

# Visualizar a variância explicada
explained_variance_ratio = pca.explained_variance_ratio_
cumulative_variance_ratio = np.cumsum(explained_variance_ratio)

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.bar(range(1, len(explained_variance_ratio) + 1), explained_variance_ratio)
plt.xlabel('Componente Principal')
plt.ylabel('Variância Explicada')
plt.title('Variância Explicada por Componente')
```

```
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(range(1, len(cumulative_variance_ratio) + 1), cumulative_variance_ratio, 'o-')
plt.axhline(y=0.95, color='r', linestyle='--', label='95% de Variância')
plt.xlabel('Número de Componentes')
plt.ylabel('Variância Explicada Acumulada')
plt.title('Variância Explicada Acumulada')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

print("Variância explicada por componente:")
for i, var in enumerate(explained_variance_ratio):
    print(f"PC{i+1}: {var:.4f} ({cumulative_variance_ratio[i]:.4f} acumulada)")

# Visualizar os dados em 2D usando os dois primeiros componentes principais
plt.figure(figsize=(10, 8))
for i, target in enumerate(target_names):
    plt.scatter(X_pca[y == i, 0], X_pca[y == i, 1], label=target, s=50, alpha=0.7)
plt.xlabel('Primeiro Componente Principal')
plt.ylabel('Segundo Componente Principal')
plt.title('PCA do Dataset Iris')
plt.legend()
plt.grid(True)
plt.show()

# Visualizar os loadings (contribuição de cada feature para os componentes principais)
loadings = pca.components_
plt.figure(figsize=(12, 8))
plt.subplot(2, 1, 1)
plt.bar(feature_names, loadings[0])
plt.title('Loadings do Primeiro Componente Principal')
plt.grid(True)

plt.subplot(2, 1, 2)
plt.bar(feature_names, loadings[1])
plt.title('Loadings do Segundo Componente Principal')
plt.grid(True)

plt.tight_layout()
plt.show()

# Visualizar os loadings como um heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(loadings, annot=True, cmap='coolwarm', xticklabels=feature_names,
            yticklabels=[f'PC{i+1}' for i in range(len(loadings))])
plt.title('Loadings dos Componentes Principais')
plt.tight_layout()
plt.show()
```

```
# Reconstruir os dados originais a partir dos componentes principais
n_components = 2
pca = PCA(n_components=n_components)
X_pca = pca.fit_transform(X_scaled)
X_reconstructed = pca.inverse_transform(X_pca)

# Calcular o erro de reconstrução
reconstruction_error = np.mean((X_scaled - X_reconstructed) ** 2)
print(f"Erro de reconstrução com {n_components} componentes: {reconstruction_error:.4f}")

# Visualizar os dados originais vs. reconstruídos para uma feature
feature_idx = 0  # Primeira feature
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.scatter(range(len(X_scaled)), X_scaled[:, feature_idx], label='Original', alpha=0.7)
plt.scatter(range(len(X_reconstructed)), X_reconstructed[:, feature_idx], label='Reconstruído', alpha=0
plt.xlabel('Amostra')
plt.ylabel(f'Valor Padronizado de {feature_names[feature_idx]}')
plt.title(f'Valores Originais vs. Reconstruídos para {feature_names[feature_idx]}')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.scatter(X_scaled[:, feature_idx], X_reconstructed[:, feature_idx], alpha=0.7)
plt.plot([-3, 3], [-3, 3], 'r--')
plt.xlabel(f'Valor Original de {feature_names[feature_idx]}')
plt.ylabel(f'Valor Reconstruído de {feature_names[feature_idx]}')
plt.title(f'Original vs. Reconstruído para {feature_names[feature_idx]}')
plt.grid(True)

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

2. t-SNE (t-Distributed Stochastic Neighbor Embedding)

t-SNE é uma técnica de redução de dimensionalidade não linear que é particularmente adequada para visualização de dados de alta dimensionalidade.

Características: - Preserva a estrutura local dos dados - Mapeia similaridades de alta dimensão para distâncias em baixa dimensão - Não linear, pode capturar relações complexas

Vantagens: - Excelente para visualização de dados de alta dimensionalidade - Preserva clusters e estruturas locais - Pode revelar padrões que métodos lineares como PCA não conseguem

Desvantagens: - Computacionalmente intensivo - Sensível a hiperparâmetros (perplexidade, taxa de aprendizado) - Não preserva distâncias globais ou densidades - Resultados podem variar entre execuções

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
import seaborn as sns
```

```
import time

# Carregar o dataset de dígitos
digits = load_digits()
X = digits.data
y = digits.target

# Visualizar alguns exemplos de dígitos
plt.figure(figsize=(10, 5))
for i in range(10):
    plt.subplot(2, 5, i + 1)
    plt.imshow(digits.images[i], cmap='binary')
    plt.title(f'Dígito: {digits.target[i]}')
    plt.axis('off')
plt.tight_layout()
plt.show()

# Aplicar t-SNE
start_time = time.time()
tsne = TSNE(n_components=2, random_state=42, perplexity=30, n_iter=1000)
X_tsne = tsne.fit_transform(X)
end_time = time.time()
print(f"Tempo de execução do t-SNE: {end_time - start_time:.2f} segundos")

# Aplicar PCA para comparação
start_time = time.time()
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
end_time = time.time()
print(f"Tempo de execução do PCA: {end_time - start_time:.2f} segundos")

# Visualizar os resultados do t-SNE
plt.figure(figsize=(12, 10))
plt.subplot(2, 1, 1)
scatter = plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=y, cmap='tab10', s=50, alpha=0.7)
plt.colorbar(scatter)
plt.title('Visualização t-SNE dos Dígitos')
plt.xlabel('t-SNE 1')
plt.ylabel('t-SNE 2')
plt.grid(True)

# Visualizar os resultados do PCA
plt.subplot(2, 1, 2)
scatter = plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='tab10', s=50, alpha=0.7)
plt.colorbar(scatter)
plt.title('Visualização PCA dos Dígitos')
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.grid(True)

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

```
# Testar diferentes valores de perplexidade
perplexities = [5, 30, 50, 100]
plt.figure(figsize=(15, 10))

for i, perplexity in enumerate(perplexities):
    plt.subplot(2, 2, i+1)

    tsne = TSNE(n_components=2, random_state=42, perplexity=perplexity, n_iter=1000)
    X_tsne = tsne.fit_transform(X)

    scatter = plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=y, cmap='tab10', s=30, alpha=0.7)
    plt.title(f't-SNE com Perplexidade = {perplexity}')
    plt.xlabel('t-SNE 1')
    plt.ylabel('t-SNE 2')
    plt.grid(True)

    if i == 0:
        plt.colorbar(scatter)

plt.tight_layout()
plt.show()

# Visualizar t-SNE em 3D
tsne_3d = TSNE(n_components=3, random_state=42, perplexity=30, n_iter=1000)
X_tsne_3d = tsne_3d.fit_transform(X)

fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
scatter = ax.scatter(X_tsne_3d[:, 0], X_tsne_3d[:, 1], X_tsne_3d[:, 2], c=y, cmap='tab10', s=30, alpha=
plt.colorbar(scatter)
plt.title('Visualização t-SNE 3D dos Dígitos')
ax.set_xlabel('t-SNE 1')
ax.set_ylabel('t-SNE 2')
ax.set_zlabel('t-SNE 3')
plt.tight_layout()
plt.show()
```

3. UMAP (Uniform Manifold Approximation and Projection)

UMAP é uma técnica de redução de dimensionalidade que é similar ao t-SNE, mas geralmente mais rápida e melhor em preservar a estrutura global dos dados.

Características: - Baseado em teoria de variedades e topologia - Preserva tanto a estrutura local quanto a global - Mais rápido que t-SNE para grandes conjuntos de dados

Vantagens: - Mais rápido que t-SNE - Melhor preservação da estrutura global - Escalável para grandes conjuntos de dados - Pode ser usado para redução de dimensionalidade supervisionada

Desvantagens: - Ainda relativamente novo e menos estabelecido - Sensível a hiperparâmetros - Requer instalação adicional (`pip install umap-learn`)

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
from sklearn.datasets import load_digits
from sklearn.manifold import TSNE
import umap
import seaborn as sns
import time

# Carregar o dataset de dígitos
digits = load_digits()
X = digits.data
y = digits.target

# Aplicar UMAP
start_time = time.time()
reducer = umap.UMAP(n_neighbors=15, min_dist=0.1, n_components=2, random_state=42)
X_umap = reducer.fit_transform(X)
end_time = time.time()
print(f"Tempo de execução do UMAP: {end_time - start_time:.2f} segundos")

# Aplicar t-SNE para comparação
start_time = time.time()
tsne = TSNE(n_components=2, random_state=42, perplexity=30, n_iter=1000)
X_tsne = tsne.fit_transform(X)
end_time = time.time()
print(f"Tempo de execução do t-SNE: {end_time - start_time:.2f} segundos")

# Visualizar os resultados do UMAP
plt.figure(figsize=(12, 10))
plt.subplot(2, 1, 1)
scatter = plt.scatter(X_umap[:, 0], X_umap[:, 1], c=y, cmap='tab10', s=50, alpha=0.7)
plt.colorbar(scatter)
plt.title('Visualização UMAP dos Dígitos')
plt.xlabel('UMAP 1')
plt.ylabel('UMAP 2')
plt.grid(True)

# Visualizar os resultados do t-SNE
plt.subplot(2, 1, 2)
scatter = plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=y, cmap='tab10', s=50, alpha=0.7)
plt.colorbar(scatter)
plt.title('Visualização t-SNE dos Dígitos')
plt.xlabel('t-SNE 1')
plt.ylabel('t-SNE 2')
plt.grid(True)

plt.tight_layout()
plt.show()

# Testar diferentes valores de n_neighbors e min_dist
n_neighbors_values = [5, 15, 30, 50]
min_dist_values = [0.0, 0.1, 0.5, 0.99]

plt.figure(figsize=(15, 15))
for i, n_neighbors in enumerate(n_neighbors_values):
```

```
    for j, min_dist in enumerate(min_dist_values):
        plt.subplot(4, 4, i*4 + j + 1)

        reducer = umap.UMAP(n_neighbors=n_neighbors, min_dist=min_dist,
                            n_components=2, random_state=42)
        X_umap = reducer.fit_transform(X)

        scatter = plt.scatter(X_umap[:, 0], X_umap[:, 1], c=y, cmap='tab10', s=30, alpha=0.7)
        plt.title(f'n_neighbors={n_neighbors}, min_dist={min_dist}')
        plt.xticks([])
        plt.yticks([])

        if i == 0 and j == 0:
            plt.colorbar(scatter)

plt.tight_layout()
plt.show()

# UMAP supervisionado
reducer_supervised = umap.UMAP(n_neighbors=15, min_dist=0.1, n_components=2,
                               random_state=42, target_metric='categorical')
X_umap_supervised = reducer_supervised.fit_transform(X, y)

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
scatter = plt.scatter(X_umap[:, 0], X_umap[:, 1], c=y, cmap='tab10', s=50, alpha=0.7)
plt.colorbar(scatter)
plt.title('UMAP Não Supervisionado')
plt.xlabel('UMAP 1')
plt.ylabel('UMAP 2')
plt.grid(True)

plt.subplot(1, 2, 2)
scatter = plt.scatter(X_umap_supervised[:, 0], X_umap_supervised[:, 1], c=y, cmap='tab10', s=50, alpha=
plt.colorbar(scatter)
plt.title('UMAP Supervisionado')
plt.xlabel('UMAP 1')
plt.ylabel('UMAP 2')
plt.grid(True)

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

Algoritmos de Detecção de Anomalias

1. Isolation Forest

Isolation Forest é um algoritmo de detecção de anomalias baseado em árvores que isola observações construindo partições aleatórias.

Características: - Baseado em árvores de decisão - Isola anomalias em vez de perfilar dados normais - Eficiente para grandes conjuntos de dados

Vantagens: - Rápido e escalável - Não requer suposições sobre a distribuição dos dados - Funciona bem em espaços de alta dimensionalidade - Não requer estimativa de densidade

Desvantagens: - Sensível a hiperparâmetros - Pode não funcionar bem se as anomalias formarem clusters - Desempenho pode degradar com muitas features irrelevantes

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.ensemble import IsolationForest
from sklearn.preprocessing import StandardScaler
import seaborn as sns

# Gerar dados sintéticos com outliers
n_samples = 300
n_outliers = 15
n_features = 2
random_state = 42

# Gerar dados normais
X, _ = make_blobs(n_samples=n_samples - n_outliers, centers=1,
                  cluster_std=0.5, random_state=random_state)

# Gerar outliers
X_outliers = np.random.uniform(low=-4, high=4, size=(n_outliers, n_features))
X = np.vstack([X, X_outliers])

# Visualizar os dados
plt.figure(figsize=(10, 6))
plt.scatter(X[:, 0], X[:, 1], s=50, alpha=0.7)
plt.title('Dados Sintéticos com Outliers')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)
plt.show()

# Aplicar Isolation Forest
clf = IsolationForest(n_estimators=100, contamination=n_outliers/n_samples,
                      random_state=random_state)
y_pred = clf.fit_predict(X)
scores = clf.decision_function(X)

# Converter para rótulos binários (1: normal, 0: anomalia)
y_pred_binary = np.where(y_pred == 1, 0, 1)  # Converter -1 para 1 (anomalia)

# Visualizar os resultados
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.scatter(X[:, 0], X[:, 1], c=y_pred_binary, cmap='viridis', s=50, alpha=0.7)
plt.colorbar(label='Anomalia (1) / Normal (0)')
plt.title('Detecção de Anomalias com Isolation Forest')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)
```

```
plt.subplot(1, 2, 2)
plt.scatter(X[:, 0], X[:, 1], c=scores, cmap='viridis', s=50, alpha=0.7)
plt.colorbar(label='Score de Anomalia')
plt.title('Scores de Anomalia')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)

plt.tight_layout()
plt.show()

# Testar diferentes valores de contaminação
contamination_values = [0.01, 0.05, 0.1, 0.2]
plt.figure(figsize=(15, 10))

for i, contamination in enumerate(contamination_values):
    plt.subplot(2, 2, i+1)

    clf = IsolationForest(n_estimators=100, contamination=contamination,
                          random_state=random_state)
    y_pred = clf.fit_predict(X)
    y_pred_binary = np.where(y_pred == 1, 0, 1)

    plt.scatter(X[:, 0], X[:, 1], c=y_pred_binary, cmap='viridis', s=50, alpha=0.7)
    plt.colorbar(label='Anomalia (1) / Normal (0)')
    plt.title(f'Contaminação = {contamination}')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.grid(True)

    n_detected = np.sum(y_pred_binary)
    plt.text(0.05, 0.95, f'Anomalias detectadas: {n_detected}',
             transform=plt.gca().transAxes, fontsize=12,
             verticalalignment='top')

plt.tight_layout()
plt.show()

# Aplicar em um dataset real: Boston Housing
from sklearn.datasets import load_boston

boston = load_boston()
X_boston = boston.data
feature_names = boston.feature_names

# Padronizar os dados
scaler = StandardScaler()
X_boston_scaled = scaler.fit_transform(X_boston)

# Aplicar Isolation Forest
clf = IsolationForest(n_estimators=100, contamination=0.05, random_state=random_state)
y_pred = clf.fit_predict(X_boston_scaled)
scores = clf.decision_function(X_boston_scaled)
```

```
# Converter para rótulos binários
y_pred_binary = np.where(y_pred == 1, 0, 1)

# Criar um DataFrame com os dados e os resultados
df_boston = pd.DataFrame(X_boston, columns=feature_names)
df_boston['anomaly'] = y_pred_binary
df_boston['score'] = scores

# Visualizar a distribuição dos scores
plt.figure(figsize=(10, 6))
sns.histplot(scores, kde=True, bins=30)
plt.axvline(x=0, color='r', linestyle='--')
plt.title('Distribuição dos Scores de Anomalia')
plt.xlabel('Score')
plt.ylabel('Frequência')
plt.grid(True)
plt.show()

# Visualizar as anomalias em um scatter plot de duas features
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.scatter(df_boston['RM'], df_boston['LSTAT'], c=y_pred_binary, cmap='viridis', s=50, alpha=0.7)
plt.colorbar(label='Anomalia (1) / Normal (0)')
plt.title('Detecção de Anomalias')
plt.xlabel('RM (Número médio de quartos)')
plt.ylabel('LSTAT (% de status baixo da população)')
plt.grid(True)

plt.subplot(1, 2, 2)
plt.scatter(df_boston['RM'], df_boston['LSTAT'], c=scores, cmap='viridis', s=50, alpha=0.7)
plt.colorbar(label='Score de Anomalia')
plt.title('Scores de Anomalia')
plt.xlabel('RM (Número médio de quartos)')
plt.ylabel('LSTAT (% de status baixo da população)')
plt.grid(True)

plt.tight_layout()
plt.show()

# Visualizar as características das anomalias
anomalies = df_boston[df_boston['anomaly'] == 1]
normal = df_boston[df_boston['anomaly'] == 0]

print(f"Número de anomalias detectadas: {len(anomalies)}")
print("\nEstatísticas das anomalias:")
print(anomalies.describe())
print("\nEstatísticas dos dados normais:")
print(normal.describe())

# Comparar a distribuição de algumas features entre dados normais e anômalos
features_to_plot = ['RM', 'LSTAT', 'DIS', 'NOX']
plt.figure(figsize=(15, 10))
```

```
for i, feature in enumerate(features_to_plot):
    plt.subplot(2, 2, i+1)
    sns.kdeplot(normal[feature], label='Normal', fill=True, alpha=0.3)
    sns.kdeplot(anomalies[feature], label='Anomalia', fill=True, alpha=0.3)
    plt.title(f'Distribuição de {feature}')
    plt.xlabel(feature)
    plt.ylabel('Densidade')
    plt.legend()
    plt.grid(True)

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

2. One-Class SVM

One-Class SVM é um algoritmo de detecção de anomalias que aprende uma fronteira de decisão que engloba a maioria dos dados normais, tratando pontos fora dessa fronteira como anomalias.

Características: - Extensão do SVM para detecção de anomalias - Aprende uma fronteira que engloba a maioria dos dados normais - Usa apenas dados normais para treinamento (em teoria)

Vantagens: - Não requer suposições sobre a distribuição dos dados - Pode capturar fronteiras de decisão complexas com kernels - Funciona bem quando as anomalias são bem separadas dos dados normais

Desvantagens: - Sensível a outliers no conjunto de treinamento - Computacionalmente intensivo para grandes conjuntos de dados - Difícil de ajustar os hiperparâmetros - Não escala bem com a dimensionalidade

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.svm import OneClassSVM
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import IsolationForest
import seaborn as sns

# Gerar dados sintéticos em forma de lua com outliers
n_samples = 300
n_outliers = 15
random_state = 42

# Gerar dados normais em forma de lua
X, _ = make_moons(n_samples=n_samples - n_outliers, noise=0.1, random_state=random_state)

# Gerar outliers
X_outliers = np.random.uniform(low=-2, high=2, size=(n_outliers, 2))
X = np.vstack([X, X_outliers])

# Padronizar os dados
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Visualizar os dados
plt.figure(figsize=(10, 6))
```

```
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], s=50, alpha=0.7)
plt.title('Dados Sintéticos em Forma de Lua com Outliers')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)
plt.show()

# Aplicar One-Class SVM
svm = OneClassSVM(kernel='rbf', gamma='scale', nu=0.05)
y_pred_svm = svm.fit_predict(X_scaled)
scores_svm = svm.decision_function(X_scaled)

# Converter para rótulos binários (1: anomalia, 0: normal)
y_pred_binary_svm = np.where(y_pred_svm == 1, 0, 1)

# Aplicar Isolation Forest para comparação
clf = IsolationForest(n_estimators=100, contamination=0.05, random_state=random_state)
y_pred_if = clf.fit_predict(X_scaled)
scores_if = clf.decision_function(X_scaled)

# Converter para rótulos binários
y_pred_binary_if = np.where(y_pred_if == 1, 0, 1)

# Visualizar os resultados
plt.figure(figsize=(12, 10))
plt.subplot(2, 2, 1)
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=y_pred_binary_svm, cmap='viridis', s=50, alpha=0.7)
plt.colorbar(label='Anomalia (1) / Normal (0)')
plt.title('Detecção de Anomalias com One-Class SVM')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)

plt.subplot(2, 2, 2)
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=scores_svm, cmap='viridis', s=50, alpha=0.7)
plt.colorbar(label='Score de Anomalia')
plt.title('Scores de Anomalia (One-Class SVM)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)

plt.subplot(2, 2, 3)
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=y_pred_binary_if, cmap='viridis', s=50, alpha=0.7)
plt.colorbar(label='Anomalia (1) / Normal (0)')
plt.title('Detecção de Anomalias com Isolation Forest')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)

plt.subplot(2, 2, 4)
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=scores_if, cmap='viridis', s=50, alpha=0.7)
plt.colorbar(label='Score de Anomalia')
plt.title('Scores de Anomalia (Isolation Forest)')
```

```
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)

plt.tight_layout()
plt.show()

# Testar diferentes valores de nu e gamma
nu_values = [0.01, 0.05, 0.1, 0.2]
gamma_values = ['scale', 0.1, 0.5, 1.0]

plt.figure(figsize=(15, 15))
for i, nu in enumerate(nu_values):
    for j, gamma in enumerate(gamma_values):
        plt.subplot(4, 4, i*4 + j + 1)

        svm = OneClassSVM(kernel='rbf', gamma=gamma, nu=nu)
        y_pred = svm.fit_predict(X_scaled)
        y_pred_binary = np.where(y_pred == 1, 0, 1)

        plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=y_pred_binary, cmap='viridis', s=30, alpha=0.7)
        plt.title(f'nu={nu}, gamma={gamma}')
        plt.xticks([])
        plt.yticks([])

        n_detected = np.sum(y_pred_binary)
        plt.text(0.05, 0.95, f'Anomalias: {n_detected}',
                 transform=plt.gca().transAxes, fontsize=10,
                 verticalalignment='top')

plt.tight_layout()
plt.show()

# Visualizar a fronteira de decisão
def plot_decision_boundary(X, model, ax=None, title=""):
    """Plotar a fronteira de decisão de um modelo de detecção de anomalias."""
    if ax is None:
        ax = plt.gca()

    # Criar uma malha para visualizar a fronteira de decisão
    h = 0.02  # Tamanho do passo da malha
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    # Fazer previsões para cada ponto da malha
    Z = model.decision_function(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plotar a fronteira de decisão
    contour = ax.contourf(xx, yy, Z, levels=np.linspace(Z.min(), 0, 7), cmap='Blues_r', alpha=0.5)
    ax.contour(xx, yy, Z, levels=[0], linewidths=2, colors='red')
```

```
    # Plotar os pontos de dados
    y_pred = model.predict(X)
    y_pred_binary = np.where(y_pred == 1, 0, 1)
    scatter = ax.scatter(X[:, 0], X[:, 1], c=y_pred_binary, cmap='viridis', s=30, alpha=0.7)

    ax.set_title(title)
    ax.set_xlabel('Feature 1')
    ax.set_ylabel('Feature 2')
    ax.grid(True)

    return contour, scatter

plt.figure(figsize=(12, 5))
ax1 = plt.subplot(1, 2, 1)
svm = OneClassSVM(kernel='rbf', gamma='scale', nu=0.05)
svm.fit(X_scaled)
plot_decision_boundary(X_scaled, svm, ax=ax1, title='Fronteira de Decisão (One-Class SVM)')

ax2 = plt.subplot(1, 2, 2)
clf = IsolationForest(n_estimators=100, contamination=0.05, random_state=random_state)
clf.fit(X_scaled)
plot_decision_boundary(X_scaled, clf, ax=ax2, title='Fronteira de Decisão (Isolation Forest)')

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

Algoritmos de Regras de Associação

1. Apriori

Apriori é um algoritmo para descobrir regras de associação em conjuntos de dados, frequentemente usado em análise de cesta de compras.

Características: - Identifica conjuntos de itens frequentes e regras de associação - Usa o princípio de que todos os subconjuntos de um conjunto frequente também devem ser frequentes - Baseado em suporte e confiança

Vantagens: - Fácil de entender e implementar - Produz regras interpretáveis - Útil para análise de cesta de compras e recomendações

Desvantagens: - Computacionalmente intensivo para grandes conjuntos de dados - Requer múltiplas passagens pelos dados - Pode gerar um grande número de regras

Exemplo de Implementação:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mlxtend.frequent_patterns import apriori, association_rules
from mlxtend.preprocessing import TransactionEncoder
import seaborn as sns

# Criar um conjunto de dados de transações
transactions = [
    ['pão', 'leite'],
    ['pão', 'fralda', 'cerveja', 'ovos'],
```

```
    ['leite', 'fralda', 'cerveja', 'refrigerante'],
    ['pão', 'leite', 'fralda', 'cerveja'],
    ['pão', 'leite', 'fralda', 'refrigerante']
]

# Converter as transações para um formato adequado para o algoritmo Apriori
te = TransactionEncoder()
te_ary = te.fit(transactions).transform(transactions)
df = pd.DataFrame(te_ary, columns=te.columns_)

print("Dados de transações:")
print(df)

# Aplicar o algoritmo Apriori para encontrar conjuntos de itens frequentes
frequent_itemsets = apriori(df, min_support=0.2, use_colnames=True)
print("\nConjuntos de itens frequentes:")
print(frequent_itemsets)

# Gerar regras de associação
rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=0.7)
print("\nRegras de associação:")
print(rules)

# Visualizar as regras de associação
plt.figure(figsize=(10, 6))
plt.scatter(rules['support'], rules['confidence'], alpha=0.5)
plt.xlabel('Suporte')
plt.ylabel('Confiança')
plt.title('Suporte vs. Confiança')
plt.grid(True)
plt.show()

# Visualizar as regras de associação com lift
plt.figure(figsize=(10, 6))
plt.scatter(rules['support'], rules['lift'], alpha=0.5)
plt.xlabel('Suporte')
plt.ylabel('Lift')
plt.title('Suporte vs. Lift')
plt.grid(True)
plt.show()

# Visualizar as regras como um heatmap
pivot = rules.pivot(index='antecedents', columns='consequents', values='lift')
plt.figure(figsize=(10, 8))
sns.heatmap(pivot, annot=True, cmap='viridis')
plt.title('Heatmap das Regras de Associação (Lift)')
plt.tight_layout()
plt.show()

# Criar um conjunto de dados maior e mais realista
import random

# Lista de produtos
```

```
products = ['pão', 'leite', 'ovos', 'manteiga', 'queijo', 'iogurte', 'carne', 'frango',
            'peixe', 'arroz', 'feijão', 'macarrão', 'molho de tomate', 'azeite', 'sal',
            'açúcar', 'café', 'chá', 'refrigerante', 'suco', 'água', 'cerveja', 'vinho',
            'biscoito', 'chocolate', 'sorvete', 'frutas', 'legumes', 'verduras', 'detergente',
            'sabão em pó', 'amaciante', 'papel higiênico', 'shampoo', 'condicionador',
            'sabonete', 'pasta de dente', 'escova de dente', 'desodorante', 'fralda']

# Definir algumas associações comuns
common_associations = [
    ['pão', 'leite', 'manteiga'],
    ['café', 'açúcar', 'leite'],
    ['arroz', 'feijão'],
    ['macarrão', 'molho de tomate'],
    ['cerveja', 'salgadinho'],
    ['vinho', 'queijo'],
    ['shampoo', 'condicionador'],
    ['escova de dente', 'pasta de dente'],
    ['fralda', 'sabonete']
]

# Gerar transações aleatórias
n_transactions = 1000
random.seed(42)
transactions = []

for _ in range(n_transactions):
    # Adicionar uma associação comum com probabilidade 0.7
    if random.random() < 0.7:
        transaction = list(random.choice(common_associations))
    else:
        transaction = []

    # Adicionar produtos aleatórios
    n_additional = random.randint(1, 5)
    additional_products = random.sample(products, n_additional)

    transaction.extend(additional_products)
    transactions.append(list(set(transaction)))  # Remover duplicatas

# Converter as transações para um formato adequado para o algoritmo Apriori
te = TransactionEncoder()
te_ary = te.fit(transactions).transform(transactions)
df = pd.DataFrame(te_ary, columns=te.columns_)

# Aplicar o algoritmo Apriori para encontrar conjuntos de itens frequentes
frequent_itemsets = apriori(df, min_support=0.05, use_colnames=True)
print("\nNúmero de conjuntos de itens frequentes:", len(frequent_itemsets))

# Gerar regras de associação
rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=0.5)
print("\nNúmero de regras de associação:", len(rules))

# Visualizar as top 10 regras por lift
```

```
top_rules = rules.sort_values('lift', ascending=False).head(10)
print("\nTop 10 regras por lift:")
print(top_rules[['antecedents', 'consequents', 'support', 'confidence', 'lift']])

# Visualizar as top 10 regras
plt.figure(figsize=(12, 8))
plt.barh(range(len(top_rules)), top_rules['lift'])
plt.yticks(range(len(top_rules)), [str(a) + ' -> ' + str(c) for a, c in zip(top_rules['antecedents'], to
plt.xlabel('Lift')
plt.title('Top 10 Regras de Associação por Lift')
plt.grid(True, axis='x')
plt.tight_layout()
plt.show()

# Visualizar a distribuição de suporte, confiança e lift
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.hist(rules['support'], bins=20)
plt.xlabel('Suporte')
plt.ylabel('Frequência')
plt.title('Distribuição de Suporte')
plt.grid(True)

plt.subplot(1, 3, 2)
plt.hist(rules['confidence'], bins=20)
plt.xlabel('Confiança')
plt.ylabel('Frequência')
plt.title('Distribuição de Confiança')
plt.grid(True)

plt.subplot(1, 3, 3)
plt.hist(rules['lift'], bins=20)
plt.xlabel('Lift')
plt.ylabel('Frequência')
plt.title('Distribuição de Lift')
plt.grid(True)

plt.tight_layout()
plt.show()

# Visualizar a relação entre suporte, confiança e lift
plt.figure(figsize=(10, 8))
scatter = plt.scatter(rules['support'], rules['confidence'], c=rules['lift'],
                      s=rules['lift']*20, alpha=0.5, cmap='viridis')
plt.colorbar(scatter, label='Lift')
plt.xlabel('Suporte')
plt.ylabel('Confiança')
plt.title('Suporte vs. Confiança vs. Lift')
plt.grid(True)
plt.tight_layout()
plt.show()
```

Comparação de Algoritmos

Ao escolher um algoritmo de aprendizado não supervisionado para um problema específico, é importante considerar vários fatores:

1. **Natureza dos dados**: Tamanho, dimensionalidade, tipos de features, presença de outliers, etc.
2. **Objetivo da análise**: Clustering, redução de dimensionalidade, detecção de anomalias, etc.
3. **Interpretabilidade**: Necessidade de entender os resultados.
4. **Desempenho**: Velocidade de treinamento e inferência.
5. **Recursos computacionais**: Memória, CPU, GPU disponíveis.

Vamos comparar os algoritmos que discutimos:

Algoritmo	Tipo	Interpretabilidade	Velocidade	Escalabilidade	Sensibilidade a Outliers	Sensibilidade a Hiperparâmetros	Formas de Clusters
K-Means	Clustering	Alta	Rápido	Boa	Alta	Média (K)	Esféricos
DBSCAN	Clustering	Média	Médio	Média	Baixa	Alta (eps, min_samples)	Arbitrários
Hierarchical Clustering	Clustering	Alta	Lento	Ruim	Alta	Média	Arbitrários
Gaussian Mixture Models	Clustering	Média	Médio	Média	Alta	Alta	Elípticos
PCA	Redução de Dimensionalidade	Média	Rápido	Boa	Alta	Baixa	Linear
t-SNE	Redução de Dimensionalidade	Baixa	Lento	Ruim	Média	Alta (perplexidade)	Não Linear
UMAP	Redução de Dimensionalidade	Baixa	Médio	Média	Média	Alta (n_neighbors, min_dist)	Não Linear
Isolation Forest	Detecção de Anomalias	Média	Rápido	Boa	Baixa	Média	N/A
One-Class SVM	Detecção de Anomalias	Baixa	Lento	Ruim	Alta	Alta (nu, gamma)	N/A
Apriori	Regras de Associação	Alta	Lento	Ruim	N/A	Média (suporte, confiança)	N/A

Exemplo de Comparação de Algoritmos de Clustering

Vamos implementar um exemplo que compara vários algoritmos de clustering em um mesmo dataset:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs, make_moons, make_circles
from sklearn.cluster import KMeans, DBSCAN, AgglomerativeClustering
from sklearn.mixture import GaussianMixture
from sklearn.metrics import silhouette_score, adjusted_rand_score
```

```
import seaborn as sns
import time

# Criar diferentes conjuntos de dados sintéticos
n_samples = 1000
random_state = 42

# Conjunto 1: Blobs (clusters esféricos)
X_blobs, y_blobs = make_blobs(n_samples=n_samples, centers=3,
                              cluster_std=0.7, random_state=random_state)

# Conjunto 2: Luas (clusters não esféricos)
X_moons, y_moons = make_moons(n_samples=n_samples, noise=0.1, random_state=random_state)

# Conjunto 3: Círculos (clusters concêntricos)
X_circles, y_circles = make_circles(n_samples=n_samples, noise=0.1, factor=0.5, random_state=random_sta

# Visualizar os conjuntos de dados
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.scatter(X_blobs[:, 0], X_blobs[:, 1], c=y_blobs, cmap='viridis', s=30, alpha=0.7)
plt.title('Blobs')
plt.grid(True)

plt.subplot(1, 3, 2)
plt.scatter(X_moons[:, 0], X_moons[:, 1], c=y_moons, cmap='viridis', s=30, alpha=0.7)
plt.title('Luas')
plt.grid(True)

plt.subplot(1, 3, 3)
plt.scatter(X_circles[:, 0], X_circles[:, 1], c=y_circles, cmap='viridis', s=30, alpha=0.7)
plt.title('Círculos')
plt.grid(True)

plt.tight_layout()
plt.show()

# Definir os algoritmos de clustering
clustering_algorithms = {
    'K-Means': KMeans(n_clusters=3, random_state=random_state),
    'DBSCAN': DBSCAN(eps=0.3, min_samples=5),
    'Agglomerative': AgglomerativeClustering(n_clusters=3),
    'Gaussian Mixture': GaussianMixture(n_components=3, random_state=random_state)
}

# Definir os conjuntos de dados
datasets = {
    'Blobs': (X_blobs, y_blobs),
    'Luas': (X_moons, y_moons),
    'Círculos': (X_circles, y_circles)
}

# Aplicar os algoritmos aos conjuntos de dados
```

```
results = {}

for dataset_name, (X, y_true) in datasets.items():
    results[dataset_name] = {}

    for algo_name, algorithm in clustering_algorithms.items():
        start_time = time.time()

        # Ajustar parâmetros específicos para cada dataset
        if dataset_name == 'Luas':
            if algo_name == 'DBSCAN':
                algorithm = DBSCAN(eps=0.3, min_samples=5)
            elif algo_name == 'K-Means':
                algorithm = KMeans(n_clusters=2, random_state=random_state)
            elif algo_name == 'Agglomerative':
                algorithm = AgglomerativeClustering(n_clusters=2)
            elif algo_name == 'Gaussian Mixture':
                algorithm = GaussianMixture(n_components=2, random_state=random_state)

        elif dataset_name == 'Círculos':
            if algo_name == 'DBSCAN':
                algorithm = DBSCAN(eps=0.2, min_samples=5)
            elif algo_name == 'K-Means':
                algorithm = KMeans(n_clusters=2, random_state=random_state)
            elif algo_name == 'Agglomerative':
                algorithm = AgglomerativeClustering(n_clusters=2)
            elif algo_name == 'Gaussian Mixture':
                algorithm = GaussianMixture(n_components=2, random_state=random_state)

        # Treinar o algoritmo
        if hasattr(algorithm, 'fit_predict'):
            y_pred = algorithm.fit_predict(X)
        else:
            algorithm.fit(X)
            y_pred = algorithm.predict(X)

        end_time = time.time()

        # Calcular métricas
        if len(np.unique(y_pred)) > 1:  # Verificar se há mais de um cluster
            silhouette = silhouette_score(X, y_pred)
            ari = adjusted_rand_score(y_true, y_pred)
        else:
            silhouette = 0
            ari = 0

        # Armazenar resultados
        results[dataset_name][algo_name] = {
            'y_pred': y_pred,
            'silhouette': silhouette,
            'ari': ari,
            'time': end_time - start_time
        }
```

```
# Visualizar os resultados
plt.figure(figsize=(15, 15))
row = 0

for dataset_name, (X, _) in datasets.items():
    for algo_name, algorithm in clustering_algorithms.items():
        row += 1
        plt.subplot(len(datasets), len(clustering_algorithms), row)

        y_pred = results[dataset_name][algo_name]['y_pred']
        silhouette = results[dataset_name][algo_name]['silhouette']
        ari = results[dataset_name][algo_name]['ari']
        exec_time = results[dataset_name][algo_name]['time']

        plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap='viridis', s=30, alpha=0.7)
        plt.title(f'{dataset_name} - {algo_name}\nSilhouette: {silhouette:.2f}, ARI: {ari:.2f}\nTempo:
        plt.xticks([])
        plt.yticks([])
        plt.grid(True)

plt.tight_layout()
plt.show()

# Comparar as métricas
metrics = ['silhouette', 'ari', 'time']
labels = ['Coeficiente de Silhueta', 'Adjusted Rand Index', 'Tempo de Execução (s)']

for i, (metric, label) in enumerate(zip(metrics, labels)):
    plt.figure(figsize=(15, 5))

    for j, dataset_name in enumerate(datasets.keys()):
        plt.subplot(1, len(datasets), j+1)

        values = [results[dataset_name][algo_name][metric] for algo_name in clustering_algorithms.keys()
        plt.bar(clustering_algorithms.keys(), values)
        plt.title(f'{label} - {dataset_name}')
        plt.xticks(rotation=45)
        plt.grid(True, axis='y')

        if metric == 'time':
            plt.yscale('log')

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

Conclusão

Neste tópico, exploramos os principais algoritmos de aprendizado não supervisionado para clustering, redução de dimensionalidade, detecção de anomalias e regras de associação. Cada algoritmo tem suas próprias características, vantagens e desvantagens, e a escolha do algoritmo adequado depende do problema específico e dos dados disponíveis.

Alguns pontos importantes a lembrar:

1. **Não existe um algoritmo perfeito para todos os problemas** (No Free Lunch Theorem).
2. **Experimentação é fundamental**: Teste diferentes algoritmos e configurações.
3. **Pré-processamento adequado dos dados** pode melhorar significativamente o desempenho.
4. **Visualização dos resultados** é essencial para entender e interpretar os padrões descobertos.
5. **Avaliação dos resultados** pode ser desafiadora sem rótulos verdadeiros, mas métricas como o coeficiente de silhueta podem ajudar.

O aprendizado não supervisionado é uma área fascinante do machine learning que permite descobrir padrões ocultos nos dados sem a necessidade de rótulos. Essas técnicas são amplamente utilizadas em segmentação de clientes, detecção de fraudes, recomendação de produtos, compressão de imagens, entre muitas outras aplicações.

No próximo tópico, exploraremos redes neurais e deep learning, que são técnicas mais avançadas de machine learning capazes de aprender representações complexas e resolver problemas desafiadores em visão computacional, processamento de linguagem natural e muito mais.

Módulo 5: Redes Neurais e Deep Learning

Módulo 5: Redes Neurais e Deep Learning

Fundamentos de Redes Neurais

As redes neurais artificiais são modelos computacionais inspirados no funcionamento do cérebro humano, capazes de aprender a partir de dados e realizar tarefas complexas como reconhecimento de imagens, processamento de linguagem natural e jogos. Neste tópico, vamos explorar os fundamentos das redes neurais, sua estrutura, funcionamento e como treiná-las.

O que são Redes Neurais?

Uma rede neural artificial é um modelo matemático composto por unidades de processamento simples (neurônios) organizadas em camadas e conectadas entre si. Cada conexão tem um peso associado que é ajustado durante o processo de aprendizado.

As redes neurais são particularmente úteis para: - Reconhecimento de padrões complexos - Aproximação de funções não lineares - Classificação e regressão - Processamento de dados sequenciais - Geração de conteúdo

Neurônio Artificial

O neurônio artificial, também chamado de perceptron, é a unidade básica de processamento de uma rede neural. Ele recebe múltiplas entradas, aplica pesos a essas entradas, soma-as, aplica uma função de ativação e produz uma saída.

A operação de um neurônio pode ser descrita matematicamente como:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

Onde: - x_i são as entradas - w_i são os pesos - b é o viés (bias) - f é a função de ativação - y é a saída

Funções de Ativação

As funções de ativação introduzem não-linearidade no modelo, permitindo que a rede aprenda relações complexas. Algumas funções de ativação comuns incluem:

1. **Sigmoid**: $f(x) = \frac{1}{1+e^{-x}}$
   - Saída entre 0 e 1
   - Útil para problemas de classificação binária
   - Sofre do problema de desvanecimento do gradiente
2. **Tangente Hiperbólica (tanh)**: $f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
   - Saída entre -1 e 1

- Similar à sigmoid, mas com saída centralizada em zero
- Também sofre do problema de desvanecimento do gradiente

3. **ReLU (Rectified Linear Unit)**: $f(x) = \max(0, x)$
   - Saída 0 para entradas negativas, e a própria entrada para valores positivos
   - Computacionalmente eficiente
   - Ajuda a mitigar o problema de desvanecimento do gradiente
   - Pode sofrer do problema de "neurônios mortos"
4. **Leaky ReLU**: $f(x) = \max(\alpha x, x)$, onde α é um valor pequeno (ex: 0.01)
   - Variação do ReLU que permite um pequeno gradiente para entradas negativas
   - Ajuda a evitar neurônios mortos
5. **Softmax**: $f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$
   - Transforma um vetor de valores em uma distribuição de probabilidade
   - Soma das saídas é 1
   - Usada na camada de saída para problemas de classificação multiclasse

Arquitetura de Redes Neurais

As redes neurais são organizadas em camadas:

1. **Camada de Entrada**: Recebe os dados de entrada
2. **Camadas Ocultas**: Processam os dados através de transformações não lineares
3. **Camada de Saída**: Produz o resultado final

Redes Neurais Feedforward

As redes neurais feedforward são o tipo mais simples de rede neural, onde a informação flui em uma única direção, da entrada para a saída, sem ciclos ou loops.

Uma rede neural feedforward com uma camada oculta pode ser representada matematicamente como:

$$\mathbf{h} = f_1(\mathbf{W_1}\mathbf{x} + \mathbf{b_1})$$
$$\mathbf{y} = f_2(\mathbf{W_2}\mathbf{h} + \mathbf{b_2})$$

Onde: - $\mathbf{x}$ é o vetor de entrada - $\mathbf{W_1}$ e $\mathbf{W_2}$ são as matrizes de pesos - $\mathbf{b_1}$ e $\mathbf{b_2}$ são os vetores de viés - f_1 e f_2 são as funções de ativação - $\mathbf{h}$ é a saída da camada oculta - $\mathbf{y}$ é a saída final

Treinamento de Redes Neurais

O treinamento de uma rede neural envolve ajustar os pesos e vieses para minimizar uma função de custo que mede a diferença entre as saídas previstas e os valores reais.

Função de Custo

A função de custo (ou função de perda) quantifica o erro do modelo. Algumas funções de custo comuns incluem:

1. **Erro Quadrático Médio (MSE)**: $L = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
   - Usada para problemas de regressão
2. **Entropia Cruzada Binária**: $L = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$
   - Usada para problemas de classificação binária
3. **Entropia Cruzada Categórica**: $L = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij}\log(\hat{y}_{ij})$
   - Usada para problemas de classificação multiclasse

Backpropagation

O algoritmo de backpropagation (retropropagação) é usado para calcular o gradiente da função de custo em relação aos pesos da rede. Ele funciona em duas fases:

1. **Forward Pass**: Os dados de entrada são propagados pela rede para calcular a saída e o erro.
2. **Backward Pass**: O erro é propagado de volta pela rede para calcular os gradientes.

Otimizadores

Os otimizadores são algoritmos que ajustam os pesos da rede com base nos gradientes calculados. Alguns otimizadores comuns incluem:

1. **Gradient Descent (Descida do Gradiente)**: Atualiza os pesos na direção oposta ao gradiente.
   - $w_{t+1} = w_t - \eta \nabla L(w_t)$
   - Onde η é a taxa de aprendizado
2. **Stochastic Gradient Descent (SGD)**: Versão estocástica do Gradient Descent que usa um subconjunto aleatório dos dados (mini-batch) em cada iteração.
3. **Momentum**: Adiciona um termo de momento para acelerar a convergência e evitar mínimos locais.
   - $v_{t+1} = \gamma v_t + \eta \nabla L(w_t)$
   - $w_{t+1} = w_t - v_{t+1}$
   - Onde γ é o coeficiente de momento
4. **Adam**: Combina as vantagens do Momentum e do RMSProp, adaptando a taxa de aprendizado para cada parâmetro.

Regularização

A regularização é usada para prevenir o overfitting, que ocorre quando o modelo se ajusta demais aos dados de treinamento e não generaliza bem para novos dados.

Técnicas de Regularização

1. **L1 Regularization (Lasso)**: Adiciona o termo $\lambda \sum_i |w_i|$ à função de custo.
   - Tende a produzir pesos esparsos (muitos zeros)
2. **L2 Regularization (Ridge)**: Adiciona o termo $\lambda \sum_i w_i^2$ à função de custo.
   - Penaliza pesos grandes
3. **Dropout**: Durante o treinamento, desativa aleatoriamente uma fração dos neurônios em cada iteração.
   - Força a rede a aprender representações redundantes
   - Simula o treinamento de múltiplas redes diferentes
4. **Batch Normalization**: Normaliza as ativações de cada camada para ter média zero e variância unitária.
   - Acelera o treinamento
   - Reduz a sensibilidade à inicialização dos pesos
   - Atua como uma forma de regularização
5. **Early Stopping**: Interrompe o treinamento quando o desempenho no conjunto de validação começa a piorar.

Implementação de uma Rede Neural Simples

Vamos implementar uma rede neural simples usando NumPy para entender melhor como funciona:

```
import numpy as np
import matplotlib.pyplot as plt
```

```
# Definir a arquitetura da rede
input_size = 2
hidden_size = 4
output_size = 1

# Inicializar os pesos e vieses
def initialize_parameters(input_size, hidden_size, output_size):
    np.random.seed(42)
    W1 = np.random.randn(hidden_size, input_size) * 0.01
    b1 = np.zeros((hidden_size, 1))
    W2 = np.random.randn(output_size, hidden_size) * 0.01
    b2 = np.zeros((output_size, 1))

    parameters = {
        "W1": W1,
        "b1": b1,
        "W2": W2,
        "b2": b2
    }

    return parameters

# Funções de ativação
def sigmoid(Z):
    return 1 / (1 + np.exp(-Z))

def relu(Z):
    return np.maximum(0, Z)

# Forward propagation
def forward_propagation(X, parameters):
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    # Camada oculta com ReLU
    Z1 = np.dot(W1, X) + b1
    A1 = relu(Z1)

    # Camada de saída com sigmoid
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)

    cache = {
        "Z1": Z1,
        "A1": A1,
        "Z2": Z2,
        "A2": A2
    }

    return A2, cache
```

```
# Função de custo (entropia cruzada binária)
def compute_cost(A2, Y):
    m = Y.shape[1]
    cost = -1/m * np.sum(Y * np.log(A2) + (1 - Y) * np.log(1 - A2))
    return cost

# Derivadas das funções de ativação
def sigmoid_derivative(Z):
    s = sigmoid(Z)
    return s * (1 - s)

def relu_derivative(Z):
    return (Z > 0).astype(int)

# Backward propagation
def backward_propagation(X, Y, cache, parameters):
    m = X.shape[1]

    W1 = parameters["W1"]
    W2 = parameters["W2"]

    A1 = cache["A1"]
    A2 = cache["A2"]
    Z1 = cache["Z1"]

    # Gradientes da camada de saída
    dZ2 = A2 - Y
    dW2 = 1/m * np.dot(dZ2, A1.T)
    db2 = 1/m * np.sum(dZ2, axis=1, keepdims=True)

    # Gradientes da camada oculta
    dZ1 = np.dot(W2.T, dZ2) * relu_derivative(Z1)
    dW1 = 1/m * np.dot(dZ1, X.T)
    db1 = 1/m * np.sum(dZ1, axis=1, keepdims=True)

    gradients = {
        "dW1": dW1,
        "db1": db1,
        "dW2": dW2,
        "db2": db2
    }

    return gradients

# Atualizar parâmetros
def update_parameters(parameters, gradients, learning_rate):
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    dW1 = gradients["dW1"]
    db1 = gradients["db1"]
```

```
    dW2 = gradients["dW2"]
    db2 = gradients["db2"]

    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2

    parameters = {
        "W1": W1,
        "b1": b1,
        "W2": W2,
        "b2": b2
    }

    return parameters

# Modelo completo
def neural_network_model(X, Y, hidden_size, num_iterations, learning_rate):
    np.random.seed(42)
    input_size = X.shape[0]
    output_size = Y.shape[0]

    # Inicializar parâmetros
    parameters = initialize_parameters(input_size, hidden_size, output_size)

    # Lista para armazenar o custo
    costs = []

    # Loop de treinamento
    for i in range(num_iterations):
        # Forward propagation
        A2, cache = forward_propagation(X, parameters)

        # Calcular o custo
        cost = compute_cost(A2, Y)

        # Backward propagation
        gradients = backward_propagation(X, Y, cache, parameters)

        # Atualizar parâmetros
        parameters = update_parameters(parameters, gradients, learning_rate)

        # Registrar o custo a cada 100 iterações
        if i % 100 == 0:
            costs.append(cost)
            print(f"Custo após iteração {i}: {cost}")

    return parameters, costs

# Gerar dados para o problema XOR
def generate_xor_data(n_samples):
    np.random.seed(42)
```

```
    X = np.random.rand(2, n_samples) * 2 - 1  # Valores entre -1 e 1
    Y = np.logical_xor(X[0, :] > 0, X[1, :] > 0).astype(int).reshape(1, n_samples)
    return X, Y

# Gerar dados
X, Y = generate_xor_data(400)

# Treinar o modelo
hidden_size = 4
num_iterations = 10000
learning_rate = 0.1

parameters, costs = neural_network_model(X, Y, hidden_size, num_iterations, learning_rate)

# Plotar o custo ao longo do treinamento
plt.figure(figsize=(10, 6))
plt.plot(costs)
plt.xlabel('Iterações (x100)')
plt.ylabel('Custo')
plt.title('Custo ao longo do treinamento')
plt.grid(True)
plt.show()

# Visualizar a fronteira de decisão
def plot_decision_boundary(X, Y, parameters):
    # Definir a malha
    h = 0.01
    x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
    y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    # Fazer previsões para cada ponto da malha
    Z = np.zeros((xx.shape[0], xx.shape[1]))
    for i in range(xx.shape[0]):
        for j in range(xx.shape[1]):
            Z[i, j] = forward_propagation(np.array([[xx[i, j]], [yy[i, j]]]), parameters)[0][0, 0]

    # Plotar a fronteira de decisão
    plt.figure(figsize=(10, 8))
    plt.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.RdBu)

    # Plotar os pontos de dados
    plt.scatter(X[0, :], X[1, :], c=Y.reshape(-1), cmap=plt.cm.RdBu, edgecolors='k')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title('Fronteira de Decisão')
    plt.grid(True)
    plt.show()

# Visualizar a fronteira de decisão
plot_decision_boundary(X, Y, parameters)
```

Implementação com TensorFlow/Keras

Agora, vamos implementar a mesma rede neural usando TensorFlow/Keras, que é uma biblioteca de alto nível para construir e treinar redes neurais:

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

# Gerar dados para o problema XOR
def generate_xor_data(n_samples):
    np.random.seed(42)
    X = np.random.rand(n_samples, 2) * 2 - 1  # Valores entre -1 e 1
    Y = np.logical_xor(X[:, 0] > 0, X[:, 1] > 0).astype(int).reshape(-1, 1)
    return X, Y

# Gerar dados
X, Y = generate_xor_data(400)

# Construir o modelo
model = Sequential([
    Dense(4, input_shape=(2,), activation='relu'),  # Camada oculta com 4 neurônios e ativação ReLU
    Dense(1, activation='sigmoid')  # Camada de saída com ativação sigmoid
])

# Compilar o modelo
model.compile(optimizer=Adam(learning_rate=0.1), loss='binary_crossentropy', metrics=['accuracy'])

# Resumo do modelo
model.summary()

# Treinar o modelo
history = model.fit(X, Y, epochs=100, batch_size=32, verbose=1)

# Plotar o custo e a acurácia ao longo do treinamento
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'])
plt.xlabel('Época')
plt.ylabel('Perda')
plt.title('Perda ao longo do treinamento')
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'])
plt.xlabel('Época')
plt.ylabel('Acurácia')
plt.title('Acurácia ao longo do treinamento')
plt.grid(True)
plt.tight_layout()
plt.show()
```

```python
# Visualizar a fronteira de decisão
def plot_decision_boundary(X, model):
    # Definir a malha
    h = 0.01
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    # Fazer previsões para cada ponto da malha
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plotar a fronteira de decisão
    plt.figure(figsize=(10, 8))
    plt.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.RdBu)

    # Plotar os pontos de dados
    plt.scatter(X[:, 0], X[:, 1], c=Y.reshape(-1), cmap=plt.cm.RdBu, edgecolors='k')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title('Fronteira de Decisão (Keras)')
    plt.grid(True)
    plt.show()

# Visualizar a fronteira de decisão
plot_decision_boundary(X, model)
```

Desafios no Treinamento de Redes Neurais

O treinamento de redes neurais apresenta vários desafios:

1. **Vanishing Gradient (Desvanecimento do Gradiente)**: Quando os gradientes se tornam muito pequenos durante a backpropagation, dificultando o treinamento de camadas mais profundas.
   - Solução: Usar funções de ativação como ReLU, inicialização de pesos adequada, batch normalization.
2. **Exploding Gradient (Explosão do Gradiente)**: Quando os gradientes se tornam muito grandes, causando instabilidade no treinamento.
   - Solução: Gradient clipping, inicialização de pesos adequada.
3. **Overfitting**: Quando o modelo se ajusta demais aos dados de treinamento e não generaliza bem.
   - Solução: Regularização (L1, L2, dropout), early stopping, aumentar o conjunto de dados.
4. **Underfitting**: Quando o modelo é muito simples para capturar a complexidade dos dados.
   - Solução: Aumentar a complexidade do modelo, treinar por mais tempo, ajustar hiperparâmetros.
5. **Escolha de Hiperparâmetros**: Taxa de aprendizado, número de camadas, número de neurônios, etc.
   - Solução: Grid search, random search, validação cruzada.

Conclusão

Neste tópico, exploramos os fundamentos das redes neurais, incluindo a estrutura do neurônio artificial, funções de ativação, arquitetura de redes, treinamento com backpropagation, otimizadores, regularização e implementação prática.

As redes neurais são ferramentas poderosas para resolver problemas complexos, mas requerem uma compreensão sólida dos conceitos fundamentais e das técnicas de treinamento para serem usadas efetivamente.

Nos próximos tópicos, exploraremos arquiteturas mais avançadas de redes neurais, como redes neurais convolucionais (CNNs) para processamento de imagens e redes neurais recorrentes (RNNs) para processamento de sequências.

Módulo 5: Redes Neurais e Deep Learning

Redes Neurais Convolucionais (CNNs)

As Redes Neurais Convolucionais (CNNs ou ConvNets) são um tipo especializado de rede neural projetada principalmente para processar dados com estrutura de grade, como imagens. Elas revolucionaram o campo da visão computacional e são fundamentais para aplicações como reconhecimento de imagens, detecção de objetos, segmentação semântica e muito mais. Neste tópico, vamos explorar a arquitetura, os componentes e as aplicações das CNNs.

Por que Redes Convolucionais?

As redes neurais tradicionais (fully connected) não são eficientes para processar imagens por vários motivos:

1. **Número excessivo de parâmetros**: Uma imagem colorida de 224x224 pixels tem $224 \times 224 \times 3 = 150.528$ pixels. Uma única camada totalmente conectada com 1000 neurônios teria mais de 150 milhões de parâmetros.
2. **Não preservam a estrutura espacial**: As redes totalmente conectadas tratam cada pixel independentemente, ignorando a estrutura espacial da imagem.
3. **Não são invariantes a translações**: Se um objeto se move na imagem, a representação muda completamente.

As CNNs resolvem esses problemas através de três ideias principais: - **Campos receptivos locais**: Cada neurônio se conecta apenas a uma região local da entrada - **Compartilhamento de parâmetros**: Os mesmos filtros são aplicados em diferentes posições da imagem - **Subamostragem (pooling)**: Redução da dimensionalidade para tornar a representação mais compacta e invariante a pequenas translações

Arquitetura de uma CNN

Uma CNN típica consiste em uma sequência de camadas:

1. **Camada Convolucional**: Aplica filtros (kernels) à imagem de entrada para extrair características
2. **Função de Ativação**: Geralmente ReLU, aplicada após cada convolução
3. **Camada de Pooling**: Reduz a dimensionalidade espacial
4. **Camadas Totalmente Conectadas**: Usadas no final da rede para classificação

Camada Convolucional

A operação de convolução consiste em deslizar um filtro (kernel) sobre a imagem de entrada e calcular o produto escalar em cada posição. Matematicamente, para uma imagem 2D:

$$S(i,j) = \sum_m \sum_n K(m,n) \cdot I(i+m, j+n)$$

Onde: - $S(i,j)$ é o valor de saída na posição (i,j) - K é o kernel (filtro) - I é a imagem de entrada

Hiperparâmetros da camada convolucional: - **Tamanho do filtro**: Dimensões do kernel (ex: 3×3, 5×5) - **Número de filtros**: Quantidade de kernels diferentes (cada um aprende a detectar um tipo de característica) - **Stride (passo)**: Distância entre aplicações sucessivas do filtro - **Padding (preenchimento)**: Adição de zeros ao redor da imagem para controlar o tamanho da saída

Camada de Pooling

A camada de pooling reduz a dimensionalidade espacial, mantendo as informações mais importantes. Os tipos mais comuns são:

1. **Max Pooling**: Seleciona o valor máximo em cada região
2. **Average Pooling**: Calcula a média dos valores em cada região

Hiperparâmetros da camada de pooling: - **Tamanho da janela**: Dimensões da região de pooling (ex: 2×2) - **Stride**: Distância entre aplicações sucessivas da janela

Arquiteturas Comuns de CNNs

Algumas arquiteturas famosas de CNNs incluem:

1. **LeNet-5** (1998): Uma das primeiras CNNs, usada para reconhecimento de dígitos manuscritos
2. **AlexNet** (2012): Vencedora do ImageNet Challenge 2012, marcou o início da revolução do deep learning em visão computacional
3. **VGG** (2014): Arquitetura simples e profunda com filtros 3×3
4. **GoogLeNet/Inception** (2014): Introduziu o módulo Inception com convoluções paralelas de diferentes tamanhos
5. **ResNet** (2015): Introduziu conexões residuais para treinar redes muito profundas
6. **MobileNet** (2017): Projetada para dispositivos móveis com recursos limitados
7. **EfficientNet** (2019): Usa escalonamento composto para equilibrar profundidade, largura e resolução

Implementação de uma CNN com TensorFlow/Keras

Vamos implementar uma CNN simples para classificação de imagens usando o dataset CIFAR-10:

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns

# Carregar e preparar o dataset CIFAR-10
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

# Normalizar os dados
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
```

```
# Converter rótulos para one-hot encoding
y_train_one_hot = tf.keras.utils.to_categorical(y_train, 10)
y_test_one_hot = tf.keras.utils.to_categorical(y_test, 10)

# Nomes das classes
class_names = ['avião', 'automóvel', 'pássaro', 'gato', 'cervo',
               'cachorro', 'sapo', 'cavalo', 'navio', 'caminhão']

# Visualizar algumas imagens do dataset
plt.figure(figsize=(10, 10))
for i in range(25):
    plt.subplot(5, 5, i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(X_train[i])
    plt.xlabel(class_names[y_train[i][0]])
plt.tight_layout()
plt.show()

# Construir o modelo CNN
model = Sequential([
    # Primeira camada convolucional
    Conv2D(32, (3, 3), padding='same', activation='relu', input_shape=(32, 32, 3)),
    BatchNormalization(),
    Conv2D(32, (3, 3), padding='same', activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),

    # Segunda camada convolucional
    Conv2D(64, (3, 3), padding='same', activation='relu'),
    BatchNormalization(),
    Conv2D(64, (3, 3), padding='same', activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),

    # Terceira camada convolucional
    Conv2D(128, (3, 3), padding='same', activation='relu'),
    BatchNormalization(),
    Conv2D(128, (3, 3), padding='same', activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),

    # Camadas totalmente conectadas
    Flatten(),
    Dense(512, activation='relu'),
    BatchNormalization(),
    Dropout(0.5),
    Dense(10, activation='softmax')
])
```

```
# Compilar o modelo
model.compile(optimizer=Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Resumo do modelo
model.summary()

# Data augmentation para melhorar a generalização
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    zoom_range=0.1
)
datagen.fit(X_train)

# Callbacks para melhorar o treinamento
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=1e-6)

# Treinar o modelo
batch_size = 64
epochs = 50

history = model.fit(
    datagen.flow(X_train, y_train_one_hot, batch_size=batch_size),
    steps_per_epoch=len(X_train) // batch_size,
    epochs=epochs,
    validation_data=(X_test, y_test_one_hot),
    callbacks=[early_stopping, reduce_lr]
)

# Avaliar o modelo
test_loss, test_acc = model.evaluate(X_test, y_test_one_hot)
print(f'Acurácia no conjunto de teste: {test_acc:.4f}')

# Plotar o histórico de treinamento
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Treino')
plt.plot(history.history['val_accuracy'], label='Validação')
plt.xlabel('Época')
plt.ylabel('Acurácia')
plt.title('Acurácia ao longo do treinamento')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Treino')
plt.plot(history.history['val_loss'], label='Validação')
plt.xlabel('Época')
```

```
plt.ylabel('Perda')
plt.title('Perda ao longo do treinamento')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Fazer previsões
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true_classes = np.squeeze(y_test)

# Matriz de confusão
plt.figure(figsize=(12, 10))
cm = confusion_matrix(y_true_classes, y_pred_classes)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, yticklabels=class_names)
plt.xlabel('Previsão')
plt.ylabel('Valor Real')
plt.title('Matriz de Confusão')
plt.tight_layout()
plt.show()

# Relatório de classificação
print('Relatório de Classificação:')
print(classification_report(y_true_classes, y_pred_classes, target_names=class_names))

# Visualizar algumas previsões
plt.figure(figsize=(12, 12))
for i in range(25):
    plt.subplot(5, 5, i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(X_test[i])
    predicted_label = class_names[y_pred_classes[i]]
    true_label = class_names[y_true_classes[i]]
    color = 'green' if predicted_label == true_label else 'red'
    plt.xlabel(f'{predicted_label} ({true_label})', color=color)
plt.tight_layout()
plt.show()

# Visualizar os filtros da primeira camada convolucional
filters = model.layers[0].get_weights()[0]
filters = np.transpose(filters, (3, 0, 1, 2))

plt.figure(figsize=(12, 8))
for i in range(32):
    plt.subplot(4, 8, i+1)
    plt.imshow(filters[i, :, :, :], cmap='viridis')
    plt.axis('off')
plt.suptitle('Filtros da Primeira Camada Convolucional')
plt.tight_layout()
plt.subplots_adjust(top=0.9)
```

```
plt.show()

# Visualizar as ativações para uma imagem de exemplo
img_index = 0
img = X_test[img_index]
img = np.expand_dims(img, axis=0)

# Criar um modelo para visualizar as ativações da primeira camada convolucional
activation_model = tf.keras.models.Model(
    inputs=model.input,
    outputs=model.layers[0].output
)

activations = activation_model.predict(img)
activations = np.squeeze(activations)

plt.figure(figsize=(12, 8))
for i in range(32):
    plt.subplot(4, 8, i+1)
    plt.imshow(activations[:, :, i], cmap='viridis')
    plt.axis('off')
plt.suptitle(f'Ativações da Primeira Camada Convolucional para {class_names[y_test[img_index][0]]}')
plt.tight_layout()
plt.subplots_adjust(top=0.9)
plt.show()
```

Transfer Learning

O transfer learning é uma técnica poderosa que permite aproveitar modelos pré-treinados em grandes conjuntos de dados para resolver problemas com conjuntos de dados menores. Isso é particularmente útil em visão computacional, onde modelos como VGG16, ResNet e MobileNet foram treinados em milhões de imagens.

Vamos implementar um exemplo de transfer learning usando o modelo VGG16 pré-treinado no ImageNet:

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.applications import VGG16
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

# Definir parâmetros
img_width, img_height = 224, 224
batch_size = 32
epochs = 20
num_classes = 5  # Exemplo com 5 classes

# Diretórios de dados (substitua pelos seus diretórios)
train_dir = 'data/train'
validation_dir = 'data/validation'
```

```
# Data augmentation para o conjunto de treinamento
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

# Apenas rescale para o conjunto de validação
validation_datagen = ImageDataGenerator(rescale=1./255)

# Carregar os dados
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    class_mode='categorical'
)

validation_generator = validation_datagen.flow_from_directory(
    validation_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    class_mode='categorical'
)

# Carregar o modelo VGG16 pré-treinado
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(img_width, img_height, 3))

# Congelar as camadas do modelo base
for layer in base_model.layers:
    layer.trainable = False

# Adicionar novas camadas no topo
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(512, activation='relu')(x)
x = Dropout(0.5)(x)
predictions = Dense(num_classes, activation='softmax')(x)

# Criar o modelo final
model = Model(inputs=base_model.input, outputs=predictions)

# Compilar o modelo
model.compile(optimizer=Adam(learning_rate=0.0001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Callbacks
```

```
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=1e-6)

# Treinar o modelo
history = model.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // batch_size,
    epochs=epochs,
    validation_data=validation_generator,
    validation_steps=validation_generator.samples // batch_size,
    callbacks=[early_stopping, reduce_lr]
)

# Plotar o histórico de treinamento
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Treino')
plt.plot(history.history['val_accuracy'], label='Validação')
plt.xlabel('Época')
plt.ylabel('Acurácia')
plt.title('Acurácia ao longo do treinamento')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Treino')
plt.plot(history.history['val_loss'], label='Validação')
plt.xlabel('Época')
plt.ylabel('Perda')
plt.title('Perda ao longo do treinamento')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Fine-tuning: descongelar algumas camadas do modelo base
for layer in base_model.layers[-4:]:
    layer.trainable = True

# Recompilar o modelo com uma taxa de aprendizado menor
model.compile(optimizer=Adam(learning_rate=1e-5),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Continuar o treinamento com fine-tuning
history_ft = model.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // batch_size,
    epochs=10,
    validation_data=validation_generator,
    validation_steps=validation_generator.samples // batch_size,
    callbacks=[early_stopping, reduce_lr]
)
```

```python
# Avaliar o modelo
validation_loss, validation_accuracy = model.evaluate(validation_generator)
print(f'Acurácia no conjunto de validação: {validation_accuracy:.4f}')

# Salvar o modelo
model.save('transfer_learning_model.h5')
```

Aplicações de CNNs

As CNNs são usadas em uma ampla variedade de aplicações:

1. **Classificação de Imagens**: Identificar a categoria principal de uma imagem
2. **Detecção de Objetos**: Localizar e classificar múltiplos objetos em uma imagem (ex: YOLO, SSD, Faster R-CNN)
3. **Segmentação Semântica**: Classificar cada pixel da imagem (ex: U-Net, DeepLab)
4. **Segmentação de Instâncias**: Identificar e separar instâncias individuais de objetos
5. **Transferência de Estilo**: Aplicar o estilo de uma imagem a outra
6. **Geração de Imagens**: Criar novas imagens (ex: GANs)
7. **Super-resolução**: Aumentar a resolução de imagens
8. **Reconhecimento Facial**: Identificar e verificar faces
9. **Análise Médica**: Detectar anomalias em imagens médicas

Técnicas Avançadas em CNNs

1. Arquiteturas Residuais (ResNet)

As redes residuais introduzem conexões de atalho (skip connections) que permitem que o gradiente flua mais facilmente através da rede, facilitando o treinamento de redes muito profundas:

```
x -> [Conv] -> [ReLU] -> [Conv] -> + -> [ReLU] -> output
|                                  ^
|----------------------------------|
```

2. Módulos Inception

Os módulos Inception aplicam convoluções de diferentes tamanhos em paralelo e concatenam os resultados, permitindo que a rede aprenda características em diferentes escalas:

```
                    -> [1x1 Conv] ->
                   /
input -> [1x1 Conv] -> [3x3 Conv] -> [Concatenate] -> output
                   \
                    -> [5x5 Conv] ->
                    \
                     -> [3x3 MaxPool] -> [1x1 Conv] ->
```

3. Atenção Espacial

Os mecanismos de atenção permitem que a rede se concentre em regiões específicas da imagem, melhorando o desempenho em tarefas como detecção de objetos e segmentação:

```
input -> [Feature Extractor] -> [Attention Module] -> [Weighted Features] -> output
```

Desafios e Boas Práticas

Desafios

1. **Overfitting**: Especialmente com conjuntos de dados pequenos
2. **Custo Computacional**: Treinamento de CNNs profundas requer GPUs potentes
3. **Quantidade de Dados**: CNNs geralmente precisam de muitos dados rotulados
4. **Interpretabilidade**: Difícil entender o que a rede está aprendendo

Boas Práticas

1. **Data Augmentation**: Aumentar artificialmente o conjunto de dados através de transformações
2. **Batch Normalization**: Normalizar as ativações para acelerar o treinamento
3. **Dropout**: Desativar aleatoriamente neurônios durante o treinamento para evitar overfitting
4. **Transfer Learning**: Usar modelos pré-treinados como ponto de partida
5. **Arquiteturas Eficientes**: Usar arquiteturas como MobileNet para dispositivos com recursos limitados
6. **Monitoramento**: Acompanhar métricas como acurácia, perda, matriz de confusão

Conclusão

As Redes Neurais Convolucionais revolucionaram o campo da visão computacional e continuam sendo a base para muitas aplicações de processamento de imagens. Sua capacidade de aprender hierarquias de características diretamente dos dados as torna extremamente poderosas para uma ampla gama de tarefas.

Neste tópico, exploramos a arquitetura das CNNs, seus componentes principais, implementação prática, transfer learning e aplicações. Nos próximos tópicos, abordaremos outras arquiteturas avançadas de redes neurais, como as Redes Neurais Recorrentes (RNNs) para processamento de sequências e dados temporais.

Módulo 5: Redes Neurais e Deep Learning

Redes Neurais Recorrentes (RNNs)

As Redes Neurais Recorrentes (RNNs) são um tipo especializado de rede neural projetada para processar dados sequenciais, como texto, áudio, séries temporais ou qualquer dado onde a ordem e o contexto são importantes. Diferentemente das redes feedforward tradicionais, as RNNs possuem conexões que formam ciclos, permitindo que a informação persista ao longo do tempo. Neste tópico, vamos explorar a arquitetura, os componentes e as aplicações das RNNs.

Por que Redes Recorrentes?

As redes neurais tradicionais (feedforward) apresentam limitações ao lidar com dados sequenciais:

1. **Não capturam dependências temporais**: Cada entrada é processada independentemente, sem considerar entradas anteriores.
2. **Tamanho fixo de entrada**: Requerem um tamanho fixo de entrada, dificultando o processamento de sequências de comprimento variável.
3. **Não compartilham parâmetros ao longo do tempo**: Precisariam de parâmetros diferentes para cada posição na sequência.

As RNNs resolvem esses problemas através de: - **Estado oculto (hidden state)**: Mantém informações sobre elementos anteriores da sequência - **Compartilhamento de parâmetros**: Os mesmos pesos são usados em cada passo de tempo - **Processamento sequencial**: Processa os dados um elemento por vez, mantendo o contexto

Arquitetura Básica de uma RNN

Uma RNN simples consiste em uma célula recorrente que processa cada elemento da sequência, um por vez, mantendo um estado oculto que é atualizado a cada passo:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$
$$y_t = W_{hy}h_t + b_y$$

Onde: - x_t é a entrada no tempo t - h_t é o estado oculto no tempo t - y_t é a saída no tempo t - W_{hh}, W_{xh}, W_{hy} são matrizes de pesos - b_h, b_y são vetores de viés - tanh é a função de ativação (pode ser outra, como ReLU ou sigmoid)

Visualmente, podemos representar uma RNN de duas formas:

1. **Representação compacta**:

```
x_t          y_t
      RNN
h_{t-1}          h_t
```

2. **Representação desdobrada no tempo**:

```
x_1          y_1   x_2          y_2   x_3          y_3
      RNN         RNN         RNN


        h_0          h_1          h_2

        h_1          h_2          h_3
```

Tipos de RNNs

Dependendo da relação entre a entrada e a saída, existem diferentes tipos de RNNs:

1. **One-to-One**: Uma entrada, uma saída (rede feedforward padrão)
2. **One-to-Many**: Uma entrada, múltiplas saídas (ex: geração de música a partir de um tema)
3. **Many-to-One**: Múltiplas entradas, uma saída (ex: classificação de sentimento)
4. **Many-to-Many (Sincronizado)**: Múltiplas entradas, múltiplas saídas alinhadas (ex: previsão de séries temporais)
5. **Many-to-Many (Assíncrono)**: Múltiplas entradas, múltiplas saídas não alinhadas (ex: tradução de texto)

Problemas das RNNs Simples

As RNNs simples enfrentam dois problemas principais:

1. **Vanishing Gradient (Desvanecimento do Gradiente)**: Durante o backpropagation through time (BPTT), os gradientes podem se tornar extremamente pequenos, dificultando o aprendizado de dependências de longo prazo.
2. **Exploding Gradient (Explosão do Gradiente)**: Os gradientes podem crescer exponencialmente, causando instabilidade no treinamento.

RNNs Avançadas

Para superar os problemas das RNNs simples, foram desenvolvidas arquiteturas mais avançadas:

1. Long Short-Term Memory (LSTM)

A LSTM foi projetada para lidar com o problema do desvanecimento do gradiente, permitindo que a rede aprenda dependências de longo prazo. Ela utiliza um sistema de "portas" (gates) para controlar o fluxo de informação:

- **Forget Gate**: Decide quais informações do estado da célula devem ser descartadas
- **Input Gate**: Decide quais novos valores serão armazenados no estado da célula

- **Output Gate**: Decide quais partes do estado da célula serão produzidas como saída

Matematicamente:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C_t} = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C_t}$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

Onde: - f_t é a forget gate - i_t é a input gate - $\tilde{C_t}$ é o candidato a novo estado da célula - C_t é o estado da célula - o_t é a output gate - h_t é o estado oculto - σ é a função sigmoid - $*$ representa multiplicação elemento a elemento

2. Gated Recurrent Unit (GRU)

A GRU é uma versão simplificada da LSTM, com menos portas e sem um estado de célula separado:

- **Reset Gate**: Decide quanto da informação anterior será esquecida
- **Update Gate**: Decide quanto da informação anterior será mantida

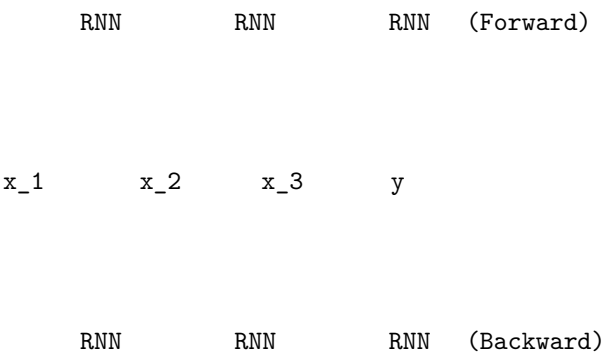
Matematicamente:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$
$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$
$$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t] + b)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Onde: - z_t é a update gate - r_t é a reset gate - $\tilde{h}_t$ é o candidato a novo estado oculto - h_t é o estado oculto

3. Bidirectional RNNs

As RNNs bidirecionais processam a sequência em ambas as direções (do início para o fim e do fim para o início), permitindo que a rede capture informações de contexto passado e futuro:



Implementação de uma RNN Simples com NumPy

Vamos implementar uma RNN simples para entender melhor como funciona:

```
import numpy as np
import matplotlib.pyplot as plt

# Função para gerar dados de exemplo: sequência de senos
def generate_sine_wave(seq_length=100, num_samples=1000):
    x = np.linspace(0, 25, num_samples)
    y = np.sin(x)

    # Criar sequências de entrada-saída
    X, Y = [], []
    for i in range(len(y) - seq_length):
        X.append(y[i:i+seq_length])
        Y.append(y[i+seq_length])

    return np.array(X), np.array(Y)

# Parâmetros
input_size = 1
hidden_size = 16
output_size = 1
seq_length = 20
learning_rate = 0.01
num_epochs = 100

# Gerar dados
X, Y = generate_sine_wave(seq_length, 1000)
X = X.reshape(-1, seq_length, input_size)
Y = Y.reshape(-1, output_size)

# Dividir em treino e teste
train_size = int(0.8 * len(X))
X_train, X_test = X[:train_size], X[train_size:]
Y_train, Y_test = Y[:train_size], Y[train_size:]

# Inicializar pesos
np.random.seed(42)
Wxh = np.random.randn(hidden_size, input_size) * 0.01  # Pesos entrada -> oculta
Whh = np.random.randn(hidden_size, hidden_size) * 0.01  # Pesos oculta -> oculta
Why = np.random.randn(output_size, hidden_size) * 0.01  # Pesos oculta -> saída
bh = np.zeros((hidden_size, 1))  # Viés da camada oculta
by = np.zeros((output_size, 1))  # Viés da camada de saída

# Função de ativação e sua derivada
def tanh(x):
    return np.tanh(x)

def tanh_derivative(x):
    return 1 - np.tanh(x)**2

# Forward pass
def forward(x, h_prev):
```

```
    # Armazenar estados para backpropagation
    inputs, hiddens, outputs = {}, {}, {}
    hiddens[-1] = np.copy(h_prev)

    # Para cada passo de tempo
    for t in range(len(x)):
        inputs[t] = x[t].reshape(-1, 1)
        hiddens[t] = tanh(np.dot(Wxh, inputs[t]) + np.dot(Whh, hiddens[t-1]) + bh)

    # Saída final
    outputs = np.dot(Why, hiddens[len(x)-1]) + by

    return outputs, hiddens, inputs

# Backward pass (Backpropagation Through Time - BPTT)
def backward(y_pred, y_true, hiddens, inputs):
    global Wxh, Whh, Why, bh, by

    # Inicializar gradientes
    dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
    dbh, dby = np.zeros_like(bh), np.zeros_like(by)
    dhnext = np.zeros_like(hiddens[0])

    # Erro na saída
    dy = y_pred - y_true.reshape(-1, 1)
    dWhy += np.dot(dy, hiddens[len(inputs)-1].T)
    dby += dy

    # Backpropagation through time
    for t in reversed(range(len(inputs))):
        # Gradiente do estado oculto
        dh = np.dot(Why.T, dy) + dhnext

        # Gradiente da função de ativação
        dhraw = dh * tanh_derivative(hiddens[t])

        # Gradientes dos pesos e vieses
        dbh += dhraw
        dWxh += np.dot(dhraw, inputs[t].T)
        dWhh += np.dot(dhraw, hiddens[t-1].T)

        # Gradiente para o próximo passo de tempo
        dhnext = np.dot(Whh.T, dhraw)

    # Clip para evitar explosão do gradiente
    for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
        np.clip(dparam, -5, 5, out=dparam)

    # Atualizar pesos
    Wxh -= learning_rate * dWxh
    Whh -= learning_rate * dWhh
    Why -= learning_rate * dWhy
    bh -= learning_rate * dbh
```

```
    by -= learning_rate * dby

    return np.sum(dy**2)

# Treinar o modelo
losses = []
h = np.zeros((hidden_size, 1))  # Estado oculto inicial

for epoch in range(num_epochs):
    total_loss = 0
    h = np.zeros((hidden_size, 1))  # Resetar estado oculto no início de cada época

    for i in range(len(X_train)):
        # Forward pass
        y_pred, hiddens, inputs = forward(X_train[i], h)

        # Backward pass
        loss = backward(y_pred, Y_train[i], hiddens, inputs)
        total_loss += loss

        # Atualizar estado oculto para a próxima sequência
        h = hiddens[len(X_train[i])-1]

    # Registrar perda média
    avg_loss = total_loss / len(X_train)
    losses.append(avg_loss)

    if epoch % 10 == 0:
        print(f'Época {epoch}, Perda: {avg_loss}')

# Plotar a perda ao longo do treinamento
plt.figure(figsize=(10, 6))
plt.plot(losses)
plt.xlabel('Época')
plt.ylabel('Perda')
plt.title('Perda ao longo do treinamento')
plt.grid(True)
plt.show()

# Testar o modelo
def predict(X, h):
    predictions = []

    for i in range(len(X)):
        y_pred, hiddens, _ = forward(X[i], h)
        predictions.append(y_pred[0, 0])
        h = hiddens[len(X[i])-1]

    return np.array(predictions)

# Fazer previsões
h = np.zeros((hidden_size, 1))
train_predictions = predict(X_train, h)
```

```
h = np.zeros((hidden_size, 1))
test_predictions = predict(X_test, h)

# Calcular erro
train_rmse = np.sqrt(np.mean((train_predictions - Y_train.flatten())**2))
test_rmse = np.sqrt(np.mean((test_predictions - Y_test.flatten())**2))
print(f'RMSE (Treino): {train_rmse}')
print(f'RMSE (Teste): {test_rmse}')

# Visualizar resultados
plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
plt.plot(Y_train.flatten(), label='Real')
plt.plot(train_predictions, label='Previsão')
plt.title('Conjunto de Treino')
plt.legend()
plt.grid(True)

plt.subplot(2, 1, 2)
plt.plot(Y_test.flatten(), label='Real')
plt.plot(test_predictions, label='Previsão')
plt.title('Conjunto de Teste')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Gerar previsões futuras
def generate_future(seed_sequence, steps):
    h = np.zeros((hidden_size, 1))
    x = seed_sequence.reshape(-1, 1)

    # Processar a sequência inicial
    for t in range(len(x)):
        h = tanh(np.dot(Wxh, x[t].reshape(-1, 1)) + np.dot(Whh, h) + bh)

    # Gerar previsões futuras
    predictions = []
    current_x = x[-1]

    for _ in range(steps):
        # Prever o próximo valor
        h = tanh(np.dot(Wxh, current_x.reshape(-1, 1)) + np.dot(Whh, h) + bh)
        y_pred = np.dot(Why, h) + by
        predictions.append(y_pred[0, 0])

        # Usar a previsão como entrada para o próximo passo
        current_x = y_pred

    return np.array(predictions)

# Gerar previsões futuras
seed_idx = 0
```

```
seed_sequence = X_test[seed_idx].flatten()
future_steps = 100
future_predictions = generate_future(seed_sequence, future_steps)

# Visualizar previsões futuras
plt.figure(figsize=(12, 6))
plt.plot(np.arange(len(seed_sequence)), seed_sequence, label='Sequência Inicial')
plt.plot(np.arange(len(seed_sequence), len(seed_sequence) + len(future_predictions)),
         future_predictions, label='Previsões Futuras')
plt.xlabel('Passo de Tempo')
plt.ylabel('Valor')
plt.title('Geração de Sequência Futura')
plt.legend()
plt.grid(True)
plt.show()
```

Implementação de LSTM com TensorFlow/Keras

Vamos implementar uma LSTM para previsão de séries temporais usando TensorFlow/Keras:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout, Bidirectional
from tensorflow.keras.optimizers import Adam
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error

# Gerar dados de exemplo: série temporal de seno com ruído
def generate_time_series(n_samples=1000):
    time = np.arange(0, n_samples)
    # Componente de tendência
    trend = 0.001 * time
    # Componente sazonal
    seasonality = 0.5 * np.sin(2 * np.pi * time / 50) + 0.25 * np.sin(2 * np.pi * time / 100)
    # Componente de ruído
    noise = 0.1 * np.random.randn(n_samples)
    # Série temporal final
    series = trend + seasonality + noise
    return series

# Gerar série temporal
np.random.seed(42)
series = generate_time_series(1000)

# Visualizar a série temporal
plt.figure(figsize=(12, 6))
plt.plot(series)
plt.title('Série Temporal Sintética')
plt.xlabel('Tempo')
plt.ylabel('Valor')
plt.grid(True)
```

```
plt.show()

# Normalizar os dados
scaler = MinMaxScaler(feature_range=(0, 1))
series_scaled = scaler.fit_transform(series.reshape(-1, 1))

# Criar sequências de entrada-saída
def create_sequences(data, seq_length):
    X, y = [], []
    for i in range(len(data) - seq_length):
        X.append(data[i:i+seq_length])
        y.append(data[i+seq_length])
    return np.array(X), np.array(y)

# Parâmetros
seq_length = 50
X, y = create_sequences(series_scaled, seq_length)

# Dividir em treino e teste
train_size = int(0.8 * len(X))
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Construir o modelo LSTM
model = Sequential([
    LSTM(50, activation='relu', return_sequences=True, input_shape=(seq_length, 1)),
    Dropout(0.2),
    LSTM(50, activation='relu'),
    Dropout(0.2),
    Dense(1)
])

# Compilar o modelo
model.compile(optimizer=Adam(learning_rate=0.001), loss='mse')

# Resumo do modelo
model.summary()

# Treinar o modelo
history = model.fit(
    X_train, y_train,
    epochs=50,
    batch_size=32,
    validation_data=(X_test, y_test),
    verbose=1
)

# Plotar o histórico de treinamento
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Treino')
plt.plot(history.history['val_loss'], label='Validação')
plt.xlabel('Época')
plt.ylabel('Perda (MSE)')
```

```
plt.title('Perda ao longo do treinamento')
plt.legend()
plt.grid(True)
plt.show()

# Fazer previsões
train_predictions = model.predict(X_train)
test_predictions = model.predict(X_test)

# Inverter a normalização
train_predictions = scaler.inverse_transform(train_predictions)
y_train_inv = scaler.inverse_transform(y_train)
test_predictions = scaler.inverse_transform(test_predictions)
y_test_inv = scaler.inverse_transform(y_test)

# Calcular métricas
train_rmse = np.sqrt(mean_squared_error(y_train_inv, train_predictions))
test_rmse = np.sqrt(mean_squared_error(y_test_inv, test_predictions))
train_mae = mean_absolute_error(y_train_inv, train_predictions)
test_mae = mean_absolute_error(y_test_inv, test_predictions)

print(f'RMSE (Treino): {train_rmse}')
print(f'RMSE (Teste): {test_rmse}')
print(f'MAE (Treino): {train_mae}')
print(f'MAE (Teste): {test_mae}')

# Visualizar resultados
# Preparar índices para plotagem
train_index = np.arange(seq_length, seq_length + len(train_predictions))
test_index = np.arange(seq_length + len(train_predictions), seq_length + len(train_predictions) + len(t

plt.figure(figsize=(12, 6))
plt.plot(series, label='Série Original')
plt.plot(train_index, train_predictions, label='Previsões (Treino)')
plt.plot(test_index, test_predictions, label='Previsões (Teste)')
plt.xlabel('Tempo')
plt.ylabel('Valor')
plt.title('Previsões vs. Valores Reais')
plt.legend()
plt.grid(True)
plt.show()

# Implementar modelo bidirecional
model_bidirectional = Sequential([
    Bidirectional(LSTM(50, activation='relu', return_sequences=True), input_shape=(seq_length, 1)),
    Dropout(0.2),
    Bidirectional(LSTM(50, activation='relu')),
    Dropout(0.2),
    Dense(1)
])

# Compilar o modelo
model_bidirectional.compile(optimizer=Adam(learning_rate=0.001), loss='mse')
```

```
# Resumo do modelo
model_bidirectional.summary()

# Treinar o modelo
history_bidirectional = model_bidirectional.fit(
    X_train, y_train,
    epochs=50,
    batch_size=32,
    validation_data=(X_test, y_test),
    verbose=1
)

# Fazer previsões com o modelo bidirecional
test_predictions_bidirectional = model_bidirectional.predict(X_test)
test_predictions_bidirectional = scaler.inverse_transform(test_predictions_bidirectional)

# Calcular métricas
test_rmse_bidirectional = np.sqrt(mean_squared_error(y_test_inv, test_predictions_bidirectional))
test_mae_bidirectional = mean_absolute_error(y_test_inv, test_predictions_bidirectional)

print(f'RMSE (Teste, Bidirecional): {test_rmse_bidirectional}')
print(f'MAE (Teste, Bidirecional): {test_mae_bidirectional}')

# Comparar modelos
plt.figure(figsize=(12, 6))
plt.plot(test_index, y_test_inv, label='Valores Reais')
plt.plot(test_index, test_predictions, label='LSTM Unidirecional')
plt.plot(test_index, test_predictions_bidirectional, label='LSTM Bidirecional')
plt.xlabel('Tempo')
plt.ylabel('Valor')
plt.title('Comparação de Modelos')
plt.legend()
plt.grid(True)
plt.show()

# Previsão de múltiplos passos à frente
def forecast_future(model, last_sequence, steps, scaler):
    future_predictions = []
    current_sequence = last_sequence.copy()

    for _ in range(steps):
        # Prever o próximo valor
        next_pred = model.predict(current_sequence.reshape(1, seq_length, 1))
        future_predictions.append(next_pred[0, 0])

        # Atualizar a sequência para a próxima previsão
        current_sequence = np.append(current_sequence[1:], next_pred)

    # Inverter a normalização
    future_predictions = scaler.inverse_transform(np.array(future_predictions).reshape(-1, 1))
    return future_predictions

# Obter a última sequência do conjunto de teste
```

```
last_sequence = X_test[-1]

# Prever valores futuros
future_steps = 100
future_predictions = forecast_future(model, last_sequence, future_steps, scaler)

# Visualizar previsões futuras
future_index = np.arange(len(series), len(series) + future_steps)

plt.figure(figsize=(12, 6))
plt.plot(series, label='Série Original')
plt.plot(future_index, future_predictions, label='Previsões Futuras')
plt.xlabel('Tempo')
plt.ylabel('Valor')
plt.title('Previsões Futuras')
plt.legend()
plt.grid(True)
plt.show()
```

Aplicações de RNNs

As RNNs são usadas em uma ampla variedade de aplicações:

1. **Processamento de Linguagem Natural (NLP)**:
   - Tradução automática
   - Geração de texto
   - Análise de sentimento
   - Chatbots e assistentes virtuais
   - Resumo de texto
2. **Séries Temporais**:
   - Previsão de preços de ações
   - Previsão de demanda
   - Previsão meteorológica
   - Análise de dados de sensores
3. **Áudio e Música**:
   - Reconhecimento de fala
   - Geração de música
   - Identificação de locutor
4. **Bioinformática**:
   - Análise de sequências de DNA
   - Previsão de estrutura de proteínas

Exemplo de Aplicação em NLP: Classificação de Sentimento

Vamos implementar um modelo LSTM para classificação de sentimento usando o dataset IMDB:

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout, Bidirectional
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.callbacks import EarlyStopping
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix
import seaborn as sns

# Carregar o dataset IMDB
vocab_size = 10000  # Tamanho do vocabulário
max_length = 200    # Comprimento máximo das sequências
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=vocab_size)

# Padding das sequências
X_train = pad_sequences(X_train, maxlen=max_length, padding='post', truncating='post')
X_test = pad_sequences(X_test, maxlen=max_length, padding='post', truncating='post')

# Verificar as dimensões dos dados
print(f'Forma de X_train: {X_train.shape}')
print(f'Forma de X_test: {X_test.shape}')

# Construir o modelo
embedding_dim = 128  # Dimensão do embedding

model = Sequential([
    Embedding(vocab_size, embedding_dim, input_length=max_length),
    Bidirectional(LSTM(64, return_sequences=True)),
    Dropout(0.3),
    Bidirectional(LSTM(32)),
    Dropout(0.3),
    Dense(16, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Compilar o modelo
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Resumo do modelo
model.summary()

# Early stopping para evitar overfitting
early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)

# Treinar o modelo
history = model.fit(
    X_train, y_train,
    epochs=10,
    batch_size=128,
    validation_split=0.2,
    callbacks=[early_stopping],
    verbose=1
)

# Plotar o histórico de treinamento
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Treino')
plt.plot(history.history['val_accuracy'], label='Validação')
```

```
plt.xlabel('Época')
plt.ylabel('Acurácia')
plt.title('Acurácia ao longo do treinamento')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Treino')
plt.plot(history.history['val_loss'], label='Validação')
plt.xlabel('Época')
plt.ylabel('Perda')
plt.title('Perda ao longo do treinamento')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Avaliar o modelo
y_pred_prob = model.predict(X_test)
y_pred = (y_pred_prob > 0.5).astype(int)

# Calcular métricas
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print(f'Acurácia: {accuracy:.4f}')
print(f'Precisão: {precision:.4f}')
print(f'Recall: {recall:.4f}')
print(f'F1-Score: {f1:.4f}')

# Matriz de confusão
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Negativo', 'Positivo'], yticklabels=['
plt.xlabel('Previsão')
plt.ylabel('Valor Real')
plt.title('Matriz de Confusão')
plt.tight_layout()
plt.show()

# Função para decodificar as críticas
word_index = imdb.get_word_index()
reverse_word_index = {value: key for key, value in word_index.items()}

def decode_review(text):
    return ' '.join([reverse_word_index.get(i - 3, '?') for i in text if i > 3])

# Visualizar algumas previsões
num_examples = 5
indices = np.random.randint(0, len(X_test), num_examples)
```

```python
for i in indices:
    review = decode_review(X_test[i])
    prediction = y_pred[i][0]
    true_sentiment = y_test[i]

    print(f'Crítica: {review[:100]}...')
    print(f'Sentimento real: {"Positivo" if true_sentiment == 1 else "Negativo"}')
    print(f'Previsão: {"Positivo" if prediction == 1 else "Negativo"}')
    print('-' * 80)
```

Arquiteturas Avançadas

1. Encoder-Decoder (Sequence-to-Sequence)

A arquitetura Encoder-Decoder é usada para tarefas onde a entrada e a saída são sequências de comprimentos diferentes, como tradução automática:

Encoder:

```
x_1          x_2          x_3          estado_final
     RNN          RNN          RNN
```

Decoder:

```
<start>          y_1           y_2          y_3
     RNN          RNN          RNN
```

```
    estado_final
```

2. Attention Mechanism

O mecanismo de atenção permite que o modelo se concentre em partes específicas da sequência de entrada ao gerar cada elemento da saída, melhorando significativamente o desempenho em tarefas como tradução:

Encoder:

```
x_1          x_2          x_3          estados_ocultos
     RNN          RNN          RNN
```

```
                              Atenção
```

```
Decoder:


<start>          y_1          y_2          y_3
     RNN         RNN         RNN
```

3. Transformers

Os Transformers substituíram as RNNs em muitas tarefas de NLP, usando apenas mecanismos de atenção (sem recorrência) para processar sequências. Modelos como BERT, GPT e T5 são baseados na arquitetura Transformer.

Desafios e Boas Práticas

Desafios

1. **Treinamento Lento**: RNNs são processadas sequencialmente, dificultando a paralelização
2. **Dependências de Longo Prazo**: Mesmo LSTMs e GRUs podem ter dificuldade com sequências muito longas
3. **Overfitting**: Especialmente com conjuntos de dados pequenos
4. **Explosão/Desvanecimento do Gradiente**: Ainda pode ocorrer, mesmo com LSTMs e GRUs

Boas Práticas

1. **Usar LSTMs ou GRUs** em vez de RNNs simples
2. **Gradient Clipping**: Limitar o valor dos gradientes para evitar explosão
3. **Bidirectional RNNs**: Para capturar contexto em ambas as direções
4. **Dropout**: Para evitar overfitting
5. **Batch Normalization**: Para acelerar o treinamento
6. **Residual Connections**: Para facilitar o fluxo do gradiente
7. **Considerar Transformers**: Para sequências muito longas ou quando a paralelização é importante

Conclusão

As Redes Neurais Recorrentes são ferramentas poderosas para processar dados sequenciais, permitindo que o modelo mantenha informações de contexto ao longo do tempo. Variantes como LSTM e GRU resolvem muitos dos problemas das RNNs simples, tornando-as eficazes para uma ampla gama de aplicações.

Neste tópico, exploramos a arquitetura das RNNs, seus componentes principais, implementação prática e aplicações. Nos próximos tópicos, abordaremos outras arquiteturas avançadas de deep learning, como Transformers e Generative Adversarial Networks (GANs).

Módulo 5: Redes Neurais e Deep Learning

Transformers e Modelos de Linguagem

Os Transformers representam uma revolução na área de processamento de linguagem natural (NLP) e além. Introduzidos no artigo "Attention is All You Need" (2017), eles substituíram as arquiteturas recorrentes (RNNs) em muitas tarefas, oferecendo melhor paralelização, captura de dependências de longo alcance e desempenho superior. Neste tópico, vamos explorar a arquitetura Transformer, seus componentes e as principais implementações como BERT, GPT e T5.

Por que Transformers?

As RNNs e LSTMs, embora poderosas para dados sequenciais, apresentam limitações:

1. **Processamento Sequencial**: Processam os dados um elemento por vez, dificultando a paralelização
2. **Dificuldade com Dependências de Longo Alcance**: Mesmo LSTMs podem perder informações em sequências muito longas
3. **Tempo de Treinamento**: Treinamento lento devido à natureza sequencial

Os Transformers resolvem esses problemas através de: - **Paralelização**: Processam todos os elementos da sequência simultaneamente - **Mecanismo de Atenção**: Capturam diretamente relações entre todos os elementos, independentemente da distância - **Sem Recorrência**: Eliminam a necessidade de estados ocultos recorrentes

Arquitetura do Transformer

A arquitetura Transformer consiste em dois componentes principais: 1. **Encoder**: Processa a sequência de entrada 2. **Decoder**: Gera a sequência de saída

Cada um desses componentes é composto por múltiplas camadas idênticas, cada uma contendo: - **Multi-Head Attention**: Permite que o modelo atenda a diferentes posições e representações - **Feed-Forward Network**: Processa cada posição independentemente - **Layer Normalization e Residual Connections**: Facilitam o treinamento

Mecanismo de Atenção

O coração do Transformer é o mecanismo de atenção, especificamente a "Scaled Dot-Product Attention":

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Onde: - Q (Query): Representação da posição atual - K (Key): Representação de todas as posições para comparação - V (Value): Valores a serem agregados - d_k: Dimensão das chaves (fator de escala)

A Multi-Head Attention executa essa operação em paralelo com diferentes projeções lineares:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$

Onde:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Positional Encoding

Como os Transformers processam todos os elementos simultaneamente, eles perdem a informação sobre a ordem dos elementos. Para resolver isso, são adicionados "Positional Encodings" às embeddings de entrada:

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$
$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

Onde: - pos: Posição na sequência - i: Dimensão - d_{model}: Dimensão do modelo

Principais Modelos Baseados em Transformers

1. BERT (Bidirectional Encoder Representations from Transformers)

BERT, desenvolvido pelo Google, é um modelo de linguagem bidirecional que utiliza apenas o encoder do Transformer. Ele é pré-treinado em duas tarefas: - **Masked Language Modeling (MLM)**: Prever palavras aleatoriamente mascaradas - **Next Sentence Prediction (NSP)**: Prever se duas sentenças são consecutivas

BERT é particularmente eficaz para tarefas de compreensão de linguagem como classificação de texto, reconhecimento de entidades nomeadas e resposta a perguntas.

2. GPT (Generative Pre-trained Transformer)

GPT, desenvolvido pela OpenAI, utiliza apenas o decoder do Transformer e é treinado de forma autorregressiva para prever o próximo token em uma sequência. Ele é particularmente eficaz para tarefas de geração de texto como completamento de texto, tradução e resumo.

A série GPT evoluiu significativamente: - **GPT-1**: Introduziu o conceito de pré-treinamento e fine-tuning - **GPT-2**: Aumentou significativamente o tamanho do modelo e demonstrou capacidades zero-shot - **GPT-3**: Escalou para 175 bilhões de parâmetros, mostrando capacidades few-shot impressionantes - **GPT-4**: Multimodal, com capacidades avançadas de raciocínio e compreensão

3. T5 (Text-to-Text Transfer Transformer)

T5, desenvolvido pelo Google, reformula todas as tarefas de NLP como problemas de texto para texto. Ele utiliza tanto o encoder quanto o decoder do Transformer e é pré-treinado em uma variedade de tarefas, tornando-o altamente versátil.

Implementação de um Transformer Simples com TensorFlow

Vamos implementar um Transformer simples para tradução de inglês para português:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, LayerNormalization, MultiHeadAttention, Dropout, Embe
from tensorflow.keras.models import Model
import matplotlib.pyplot as plt
```

```python
# Definir hiperparâmetros
d_model = 128  # Dimensão do modelo
num_heads = 8  # Número de cabeças de atenção
dff = 512  # Dimensão da feed-forward network
dropout_rate = 0.1
max_seq_length = 50
vocab_size = 10000

# Positional Encoding
def get_positional_encoding(max_seq_length, d_model):
    positional_encoding = np.zeros((max_seq_length, d_model))
    for pos in range(max_seq_length):
        for i in range(0, d_model, 2):
            positional_encoding[pos, i] = np.sin(pos / (10000 ** (i / d_model)))
            if i + 1 < d_model:
                positional_encoding[pos, i + 1] = np.cos(pos / (10000 ** (i / d_model)))

    return tf.cast(positional_encoding[np.newaxis, ...], dtype=tf.float32)

# Camada de Encoder
def encoder_layer(units, d_model, num_heads, dff, dropout_rate=0.1):
    inputs = Input(shape=(None, d_model))

    # Multi-Head Attention
    attention_output = MultiHeadAttention(
        num_heads=num_heads, key_dim=d_model // num_heads)(inputs, inputs, inputs)
    attention_output = Dropout(dropout_rate)(attention_output)

    # Add & Norm
    out1 = LayerNormalization(epsilon=1e-6)(inputs + attention_output)

    # Feed Forward Network
    ffn_output = Dense(dff, activation='relu')(out1)
    ffn_output = Dense(d_model)(ffn_output)
    ffn_output = Dropout(dropout_rate)(ffn_output)

    # Add & Norm
    out2 = LayerNormalization(epsilon=1e-6)(out1 + ffn_output)

    return Model(inputs=inputs, outputs=out2)

# Camada de Decoder
def decoder_layer(units, d_model, num_heads, dff, dropout_rate=0.1):
    inputs = Input(shape=(None, d_model))
    enc_outputs = Input(shape=(None, d_model))

    # Masked Multi-Head Attention
    look_ahead_mask = 1 - tf.linalg.band_part(tf.ones((tf.shape(inputs)[1], tf.shape(inputs)[1])), -1, 0
    masked_attention_output = MultiHeadAttention(
        num_heads=num_heads, key_dim=d_model // num_heads)(
        inputs, inputs, inputs, attention_mask=look_ahead_mask)
    masked_attention_output = Dropout(dropout_rate)(masked_attention_output)
```

```
    # Add & Norm
    out1 = LayerNormalization(epsilon=1e-6)(inputs + masked_attention_output)

    # Multi-Head Attention with encoder outputs
    attention_output = MultiHeadAttention(
        num_heads=num_heads, key_dim=d_model // num_heads)(
        out1, enc_outputs, enc_outputs)
    attention_output = Dropout(dropout_rate)(attention_output)

    # Add & Norm
    out2 = LayerNormalization(epsilon=1e-6)(out1 + attention_output)

    # Feed Forward Network
    ffn_output = Dense(dff, activation='relu')(out2)
    ffn_output = Dense(d_model)(ffn_output)
    ffn_output = Dropout(dropout_rate)(ffn_output)

    # Add & Norm
    out3 = LayerNormalization(epsilon=1e-6)(out2 + ffn_output)

    return Model(inputs=[inputs, enc_outputs], outputs=out3)

# Encoder completo
def encoder(vocab_size, num_layers, units, d_model, num_heads, dff, max_seq_length, dropout_rate=0.1):
    inputs = Input(shape=(None,))

    # Embedding
    embeddings = Embedding(vocab_size, d_model)(inputs)
    embeddings *= tf.math.sqrt(tf.cast(d_model, tf.float32))

    # Positional Encoding
    positional_encoding = get_positional_encoding(max_seq_length, d_model)
    pos_encoding = positional_encoding[:, :tf.shape(embeddings)[1], :]

    x = embeddings + pos_encoding
    x = Dropout(dropout_rate)(x)

    # Encoder layers
    for i in range(num_layers):
        encoder_layer_instance = encoder_layer(units, d_model, num_heads, dff, dropout_rate)
        x = encoder_layer_instance(x)

    return Model(inputs=inputs, outputs=x)

# Decoder completo
def decoder(vocab_size, num_layers, units, d_model, num_heads, dff, max_seq_length, dropout_rate=0.1):
    inputs = Input(shape=(None,))
    enc_outputs = Input(shape=(None, d_model))

    # Embedding
    embeddings = Embedding(vocab_size, d_model)(inputs)
    embeddings *= tf.math.sqrt(tf.cast(d_model, tf.float32))
```

```
    # Positional Encoding
    positional_encoding = get_positional_encoding(max_seq_length, d_model)
    pos_encoding = positional_encoding[:, :tf.shape(embeddings)[1], :]

    x = embeddings + pos_encoding
    x = Dropout(dropout_rate)(x)

    # Decoder layers
    for i in range(num_layers):
        decoder_layer_instance = decoder_layer(units, d_model, num_heads, dff, dropout_rate)
        x = decoder_layer_instance([x, enc_outputs])

    # Final output layer
    outputs = Dense(vocab_size)(x)

    return Model(inputs=[inputs, enc_outputs], outputs=outputs)

# Transformer completo
def transformer(vocab_size, num_layers, units, d_model, num_heads, dff, max_seq_length, dropout_rate=0.
    # Encoder
    encoder_inputs = Input(shape=(None,))
    encoder_model = encoder(vocab_size, num_layers, units, d_model, num_heads, dff, max_seq_length, dro
    enc_outputs = encoder_model(encoder_inputs)

    # Decoder
    decoder_inputs = Input(shape=(None,))
    decoder_model = decoder(vocab_size, num_layers, units, d_model, num_heads, dff, max_seq_length, dro
    decoder_outputs = decoder_model([decoder_inputs, enc_outputs])

    # Final model
    model = Model(inputs=[encoder_inputs, decoder_inputs], outputs=decoder_outputs)

    return model

# Criar o modelo
num_layers = 4
units = 512
model = transformer(vocab_size, num_layers, units, d_model, num_heads, dff, max_seq_length, dropout_rat

# Compilar o modelo
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['accuracy']
)

# Resumo do modelo
model.summary()

# Nota: Para treinar este modelo, precisaríamos de um dataset de tradução inglês-português
# e implementar um gerador de dados personalizado para alimentar o modelo.
```

Implementação de Fine-tuning do BERT com Hugging Face

A biblioteca Hugging Face Transformers facilita muito o uso de modelos pré-treinados. Vamos implementar um exemplo de fine-tuning do BERT para classificação de sentimento:

```
import numpy as np
import pandas as pd
import torch
from torch.utils.data import Dataset, DataLoader
from transformers import BertTokenizer, BertForSequenceClassification, AdamW
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt
import seaborn as sns

# Verificar se GPU está disponível
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Usando dispositivo: {device}")

# Carregar o tokenizador e o modelo pré-treinado
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)
model.to(device)

# Exemplo de dataset (substitua por seu próprio dataset)
# Aqui estamos usando um exemplo simples de críticas de filmes
texts = [
    "Este filme é incrível, adorei cada minuto!",
    "Que desperdício de tempo, não recomendo para ninguém.",
    "Uma obra-prima do cinema moderno.",
    "Atuações medíocres e roteiro previsível.",
    "Fiquei emocionado do início ao fim.",
    # ... adicione mais exemplos
]
labels = [1, 0, 1, 0, 1]  # 1: positivo, 0: negativo

# Dividir em treino e teste
X_train, X_test, y_train, y_test = train_test_split(texts, labels, test_size=0.2, random_state=42)

# Criar dataset personalizado
class SentimentDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_length=128):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        text = self.texts[idx]
        label = self.labels[idx]

        encoding = self.tokenizer(
```

```
            text,
            add_special_tokens=True,
            max_length=self.max_length,
            padding='max_length',
            truncation=True,
            return_tensors='pt'
        )

        return {
            'input_ids': encoding['input_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].flatten(),
            'labels': torch.tensor(label, dtype=torch.long)
        }

# Criar datasets e dataloaders
train_dataset = SentimentDataset(X_train, y_train, tokenizer)
test_dataset = SentimentDataset(X_test, y_test, tokenizer)

batch_size = 16
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size)

# Configurar otimizador
optimizer = AdamW(model.parameters(), lr=2e-5)

# Função de treinamento
def train_epoch(model, dataloader, optimizer, device):
    model.train()
    total_loss = 0

    for batch in dataloader:
        optimizer.zero_grad()

        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        outputs = model(
            input_ids=input_ids,
            attention_mask=attention_mask,
            labels=labels
        )

        loss = outputs.loss
        total_loss += loss.item()

        loss.backward()
        optimizer.step()

    return total_loss / len(dataloader)

# Função de avaliação
def evaluate(model, dataloader, device):
```

```
    model.eval()
    predictions = []
    actual_labels = []

    with torch.no_grad():
        for batch in dataloader:
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['labels'].to(device)

            outputs = model(
                input_ids=input_ids,
                attention_mask=attention_mask
            )

            _, preds = torch.max(outputs.logits, dim=1)

            predictions.extend(preds.cpu().tolist())
            actual_labels.extend(labels.cpu().tolist())

    return accuracy_score(actual_labels, predictions), classification_report(actual_labels, predictions

# Treinar o modelo
epochs = 3
train_losses = []

for epoch in range(epochs):
    print(f"Época {epoch+1}/{epochs}")
    train_loss = train_epoch(model, train_dataloader, optimizer, device)
    train_losses.append(train_loss)
    print(f"Perda de treinamento: {train_loss:.4f}")

    accuracy, report = evaluate(model, test_dataloader, device)
    print(f"Acurácia: {accuracy:.4f}")
    print(report)
    print("-" * 50)

# Plotar a perda de treinamento
plt.figure(figsize=(10, 6))
plt.plot(range(1, epochs+1), train_losses, marker='o')
plt.title('Perda de Treinamento por Época')
plt.xlabel('Época')
plt.ylabel('Perda')
plt.grid(True)
plt.show()

# Salvar o modelo
model.save_pretrained('./bert_sentiment_model')
tokenizer.save_pretrained('./bert_sentiment_model')

# Função para fazer previsões em novos textos
def predict_sentiment(text, model, tokenizer, device):
    model.eval()
```

```
    encoding = tokenizer(
        text,
        add_special_tokens=True,
        max_length=128,
        padding='max_length',
        truncation=True,
        return_tensors='pt'
    )

    input_ids = encoding['input_ids'].to(device)
    attention_mask = encoding['attention_mask'].to(device)

    with torch.no_grad():
        outputs = model(
            input_ids=input_ids,
            attention_mask=attention_mask
        )
        _, preds = torch.max(outputs.logits, dim=1)

    return "Positivo" if preds.item() == 1 else "Negativo"

# Testar em algumas frases
test_sentences = [
    "Este é o melhor filme que já vi!",
    "Não gostei nada da história, muito chata.",
    "Uma experiência cinematográfica única e emocionante.",
    "Atuações fracas e efeitos especiais terríveis."
]

for sentence in test_sentences:
    sentiment = predict_sentiment(sentence, model, tokenizer, device)
    print(f"Texto: {sentence}")
    print(f"Sentimento: {sentiment}")
    print("-" * 50)
```

Aplicações de Transformers

Os Transformers revolucionaram o campo de NLP e estão sendo aplicados em uma ampla variedade de tarefas:

1. **Processamento de Linguagem Natural**:
   - Tradução automática
   - Resumo de texto
   - Geração de texto
   - Resposta a perguntas
   - Análise de sentimento
   - Reconhecimento de entidades nomeadas
   - Preenchimento de texto mascarado
2. **Visão Computacional**:
   - Classificação de imagens
   - Detecção de objetos
   - Segmentação semântica
   - Geração de legendas para imagens
3. **Áudio e Fala**:

- Reconhecimento de fala
- Síntese de fala
- Classificação de áudio

4. **Multimodal**:
   - Compreensão de imagem e texto
   - Geração de imagens a partir de texto
   - Descrição de vídeos

Modelos de Linguagem de Grande Escala

Os modelos de linguagem de grande escala (Large Language Models - LLMs) são Transformers com bilhões de parâmetros, treinados em vastos corpora de texto. Eles demonstram capacidades impressionantes:

1. **Few-shot Learning**: Aprender novas tarefas com poucos exemplos
2. **Zero-shot Learning**: Realizar tarefas sem exemplos específicos
3. **In-context Learning**: Aprender a partir do contexto fornecido
4. **Emergent Abilities**: Habilidades que surgem apenas em modelos de grande escala

Exemplos de LLMs: - **GPT-3/4 (OpenAI)**: 175B+ parâmetros - **PaLM (Google)**: 540B parâmetros - **LLaMA (Meta)**: 7B a 65B parâmetros - **Claude (Anthropic)**: Tamanho não divulgado - **Gemini (Google)**: Multimodal, tamanho não divulgado

Desafios e Considerações Éticas

Os Transformers e LLMs apresentam desafios significativos:

1. **Custo Computacional**: Treinamento e inferência requerem recursos computacionais substanciais
2. **Viés e Fairness**: Podem perpetuar ou amplificar vieses presentes nos dados de treinamento
3. **Toxicidade**: Podem gerar conteúdo tóxico ou prejudicial
4. **Privacidade**: Podem memorizar informações sensíveis dos dados de treinamento
5. **Desinformação**: Podem gerar informações falsas ou enganosas de forma convincente
6. **Impacto Ambiental**: O treinamento de modelos grandes tem uma pegada de carbono significativa

Técnicas Avançadas

1. Prompt Engineering

A engenharia de prompts é a arte de projetar entradas (prompts) que orientam o modelo a produzir saídas desejadas:

- **Zero-shot Prompting**: "Traduza o seguinte texto para o francês: 'Hello, how are you?' "
- **Few-shot Prompting**: Fornecer alguns exemplos antes da tarefa principal
- **Chain-of-Thought**: Solicitar raciocínio passo a passo
- **ReAct**: Combinar raciocínio e ações

2. RLHF (Reinforcement Learning from Human Feedback)

RLHF é uma técnica para alinhar modelos de linguagem com preferências humanas: 1. **Pré-treinamento**: Treinar um modelo de linguagem em um corpus grande 2. **Coleta de Feedback**: Coletar avaliações humanas de diferentes respostas 3. **Treinamento de Modelo de Recompensa**: Treinar um modelo para prever preferências humanas 4. **Fine-tuning com RL**: Otimizar o modelo usando o modelo de recompensa

3. Modelos Multimodais

Modelos que podem processar e gerar múltiplas modalidades (texto, imagem, áudio): - **CLIP (Contrastive Language-Image Pre-training)**: Conecta texto e imagens - **DALL-E, Midjourney, Stable Diffusion**: Geram imagens a partir de descrições textuais - **GPT-4V, Gemini**: Processam texto e imagens para várias tarefas

Conclusão

Os Transformers revolucionaram o campo de NLP e estão expandindo para outras áreas como visão computacional e processamento de áudio. Sua capacidade de processar sequências em paralelo e capturar dependências de longo alcance os torna extremamente poderosos para uma ampla gama de tarefas.

Os modelos de linguagem de grande escala baseados em Transformers demonstram capacidades impressionantes de compreensão e geração de linguagem, mas também apresentam desafios significativos em termos de recursos computacionais, viés, toxicidade e desinformação.

À medida que a tecnologia continua a evoluir, é provável que vejamos aplicações ainda mais impressionantes e impactantes dos Transformers em diversos domínios, bem como avanços para abordar os desafios atuais.

Módulo 5: Redes Neurais e Deep Learning

Generative Adversarial Networks (GANs) e Modelos Generativos

As Generative Adversarial Networks (GANs) representam uma classe poderosa de modelos generativos que revolucionaram a capacidade das máquinas de criar conteúdo novo e realista. Introduzidas por Ian Goodfellow e colegas em 2014, as GANs consistem em dois modelos neurais que competem entre si: um gerador que cria amostras e um discriminador que tenta distinguir amostras reais de amostras geradas. Neste tópico, vamos explorar a arquitetura das GANs, suas variantes, aplicações e implementações práticas.

Fundamentos das GANs

Arquitetura Básica

Uma GAN consiste em dois componentes principais:

1. **Gerador (G)**: Uma rede neural que aprende a mapear de um espaço latente (geralmente um vetor de ruído aleatório) para o espaço de dados desejado (ex: imagens). O objetivo do gerador é criar amostras tão realistas que o discriminador não consiga distingui-las das amostras reais.
2. **Discriminador (D)**: Uma rede neural que recebe amostras (reais ou geradas) e tenta classificá-las como reais ou falsas. O objetivo do discriminador é maximizar sua capacidade de distinguir entre amostras reais e geradas.

O treinamento de uma GAN é um jogo de soma zero (minimax) entre esses dois componentes: - O gerador tenta minimizar a probabilidade do discriminador classificar corretamente as amostras geradas - O discriminador tenta maximizar sua precisão na classificação de amostras reais e geradas

Matematicamente, o objetivo é encontrar um equilíbrio de Nash no seguinte jogo de soma zero:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

Onde: - $p_{data}(x)$ é a distribuição dos dados reais - $p_z(z)$ é a distribuição do espaço latente (geralmente uma distribuição normal) - $G(z)$ é a saída do gerador para a entrada z - $D(x)$ é a probabilidade do discriminador classificar x como real

Processo de Treinamento

O treinamento de uma GAN geralmente segue estes passos:

1. **Amostragem**: Obter um mini-batch de amostras reais e gerar um mini-batch de amostras falsas usando o gerador
2. **Atualizar o Discriminador**: Treinar o discriminador para maximizar a probabilidade de classificar corretamente amostras reais e falsas

3. **Amostragem de Ruído**: Gerar um novo mini-batch de ruído
4. **Atualizar o Gerador**: Treinar o gerador para minimizar a probabilidade do discriminador classificar corretamente as amostras geradas
5. **Repetir**: Continuar este processo até atingir um equilíbrio

Desafios no Treinamento de GANs

O treinamento de GANs apresenta vários desafios:

1. **Instabilidade**: O equilíbrio entre gerador e discriminador é difícil de alcançar e manter
2. **Modo Collapse**: O gerador pode aprender a produzir apenas um subconjunto limitado de amostras
3. **Não-convergência**: As GANs podem não convergir para um equilíbrio estável
4. **Avaliação**: É difícil quantificar objetivamente o desempenho de modelos generativos

Variantes Importantes de GANs

1. DCGAN (Deep Convolutional GAN)

DCGANs incorporam camadas convolucionais no gerador e discriminador, tornando-as mais adequadas para tarefas de geração de imagens. Elas introduzem várias diretrizes arquitetônicas: - Usar strided convolutions em vez de pooling - Usar BatchNormalization em ambas as redes - Remover camadas totalmente conectadas - Usar ReLU no gerador e LeakyReLU no discriminador

2. Conditional GAN (cGAN)

As cGANs permitem controlar o processo de geração condicionando tanto o gerador quanto o discriminador em informações adicionais, como rótulos de classe. Isso permite gerar amostras de uma classe específica.

3. CycleGAN

CycleGANs permitem a tradução de imagem para imagem sem pares de treinamento alinhados. Elas usam duas GANs e uma restrição de consistência cíclica para aprender mapeamentos entre domínios.

4. StyleGAN

StyleGANs introduzem um mecanismo de estilo adaptativo que permite controle fino sobre as características geradas e produz imagens de alta qualidade com variação natural.

5. Pix2Pix

Pix2Pix é uma GAN condicional para tradução de imagem para imagem com pares de treinamento alinhados, como mapas de segmentação para fotos realistas.

Implementação de uma DCGAN com TensorFlow

Vamos implementar uma DCGAN para gerar imagens de dígitos manuscritos usando o dataset MNIST:

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Reshape, Flatten, Dropout, LeakyReLU, Conv2D, Conv2DT
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.datasets import mnist

# Carregar e preparar o dataset MNIST
(X_train, _), (_, _) = mnist.load_data()
```

```
X_train = X_train / 127.5 - 1.0  # Normalizar para [-1, 1]
X_train = np.expand_dims(X_train, axis=-1)  # Adicionar canal

# Parâmetros
img_rows = 28
img_cols = 28
channels = 1
img_shape = (img_rows, img_cols, channels)
latent_dim = 100

# Construir o gerador
def build_generator():
    model = Sequential()

    # Camada densa inicial
    model.add(Dense(7*7*256, use_bias=False, input_shape=(latent_dim,)))
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha=0.2))

    # Reshape para começar o processo de upsampling
    model.add(Reshape((7, 7, 256)))

    # Upsampling para 14x14
    model.add(Conv2DTranspose(128, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha=0.2))

    # Upsampling para 28x28
    model.add(Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha=0.2))

    # Camada de saída
    model.add(Conv2DTranspose(1, (5, 5), strides=(1, 1), padding='same', use_bias=False, activation='ta

    # Resumo do modelo
    model.summary()

    # Entrada de ruído e saída de imagem gerada
    noise = Input(shape=(latent_dim,))
    img = model(noise)

    return Model(noise, img)

# Construir o discriminador
def build_discriminator():
    model = Sequential()

    # Primeira camada convolucional
    model.add(Conv2D(64, (5, 5), strides=(2, 2), padding='same', input_shape=img_shape))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.3))
```

```
    # Segunda camada convolucional
    model.add(Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.3))

    # Achatar e camada de saída
    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))

    # Resumo do modelo
    model.summary()

    # Entrada de imagem e saída de classificação
    img = Input(shape=img_shape)
    validity = model(img)

    return Model(img, validity)

# Construir e compilar o discriminador
discriminator = build_discriminator()
discriminator.compile(loss='binary_crossentropy',
                      optimizer=Adam(learning_rate=0.0002, beta_1=0.5),
                      metrics=['accuracy'])

# Construir o gerador
generator = build_generator()

# Para o modelo combinado, não atualizamos os pesos do discriminador
discriminator.trainable = False

# A entrada do modelo combinado é ruído, a saída é a validade da imagem gerada
z = Input(shape=(latent_dim,))
img = generator(z)
validity = discriminator(img)

# O modelo combinado (gerador + discriminador)
combined = Model(z, validity)
combined.compile(loss='binary_crossentropy',
                 optimizer=Adam(learning_rate=0.0002, beta_1=0.5))

# Função para salvar imagens geradas
def save_imgs(epoch, generator, examples=25, dim=(5, 5), figsize=(10, 10)):
    noise = np.random.normal(0, 1, (examples, latent_dim))
    gen_imgs = generator.predict(noise)

    # Rescale images 0 - 1
    gen_imgs = 0.5 * gen_imgs + 0.5

    plt.figure(figsize=figsize)
    for i in range(examples):
        plt.subplot(dim[0], dim[1], i+1)
        plt.imshow(gen_imgs[i, :, :, 0], cmap='gray')
        plt.axis('off')
```

```
    plt.tight_layout()
    plt.savefig(f"gan_images/mnist_epoch_{epoch}.png")
    plt.close()

# Criar diretório para salvar imagens
import os
os.makedirs("gan_images", exist_ok=True)

# Parâmetros de treinamento
epochs = 30000
batch_size = 128
save_interval = 1000

# Rótulos para dados reais e falsos
valid = np.ones((batch_size, 1))
fake = np.zeros((batch_size, 1))

# Histórico de perdas
d_losses = []
g_losses = []

# Loop de treinamento
for epoch in range(epochs):
    # Treinar o discriminador

    # Selecionar um batch aleatório de imagens
    idx = np.random.randint(0, X_train.shape[0], batch_size)
    imgs = X_train[idx]

    # Gerar um batch de novas imagens
    noise = np.random.normal(0, 1, (batch_size, latent_dim))
    gen_imgs = generator.predict(noise)

    # Treinar o discriminador
    d_loss_real = discriminator.train_on_batch(imgs, valid)
    d_loss_fake = discriminator.train_on_batch(gen_imgs, fake)
    d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

    # Treinar o gerador
    noise = np.random.normal(0, 1, (batch_size, latent_dim))
    g_loss = combined.train_on_batch(noise, valid)

    # Registrar histórico
    d_losses.append(d_loss[0])
    g_losses.append(g_loss)

    # Imprimir progresso
    if epoch % 100 == 0:
        print(f"{epoch} [D loss: {d_loss[0]:.4f}, acc.: {100*d_loss[1]:.2f}%] [G loss: {g_loss:.4f}]")

    # Salvar imagens geradas
    if epoch % save_interval == 0:
        save_imgs(epoch, generator)
```

```
# Plotar histórico de perdas
plt.figure(figsize=(10, 5))
plt.plot(d_losses, label='Discriminator')
plt.plot(g_losses, label='Generator')
plt.title('Loss over training')
plt.legend()
plt.savefig("gan_images/loss_history.png")
plt.close()

# Gerar e salvar um conjunto final de imagens
save_imgs(epochs, generator, examples=25, dim=(5, 5), figsize=(10, 10))
```

Implementação de uma Conditional GAN (cGAN)

Vamos implementar uma cGAN para gerar dígitos MNIST de uma classe específica:

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Reshape, Flatten, Dropout, LeakyReLU, Conv2D, Conv2DT
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.datasets import mnist

# Carregar e preparar o dataset MNIST
(X_train, y_train), (_, _) = mnist.load_data()
X_train = X_train / 127.5 - 1.0  # Normalizar para [-1, 1]
X_train = np.expand_dims(X_train, axis=-1)  # Adicionar canal

# Parâmetros
img_rows = 28
img_cols = 28
channels = 1
img_shape = (img_rows, img_cols, channels)
latent_dim = 100
num_classes = 10

# Construir o gerador
def build_generator():
    # Entrada de ruído
    noise = Input(shape=(latent_dim,))

    # Entrada de rótulo
    label = Input(shape=(1,), dtype='int32')

    # Embedding do rótulo
    label_embedding = Embedding(num_classes, 50)(label)
    label_embedding = Flatten()(label_embedding)

    # Concatenar ruído e embedding do rótulo
    model_input = Concatenate()([noise, label_embedding])

    # Camada densa inicial
```

```
    x = Dense(7*7*256, use_bias=False)(model_input)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Reshape((7, 7, 256))(x)

    # Upsampling para 14x14
    x = Conv2DTranspose(128, (5, 5), strides=(2, 2), padding='same', use_bias=False)(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)

    # Upsampling para 28x28
    x = Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False)(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)

    # Camada de saída
    img = Conv2DTranspose(channels, (5, 5), strides=(1, 1), padding='same', use_bias=False, activation=

    return Model([noise, label], img)

# Construir o discriminador
def build_discriminator():
    # Entrada de imagem
    img = Input(shape=img_shape)

    # Entrada de rótulo
    label = Input(shape=(1,), dtype='int32')

    # Embedding do rótulo
    label_embedding = Embedding(num_classes, np.prod(img_shape))(label)
    label_embedding = Flatten()(label_embedding)
    label_embedding = Reshape(img_shape)(label_embedding)

    # Concatenar imagem e embedding do rótulo
    combined_input = Concatenate(axis=-1)([img, label_embedding])

    # Primeira camada convolucional
    x = Conv2D(64, (5, 5), strides=(2, 2), padding='same')(combined_input)
    x = LeakyReLU(alpha=0.2)(x)
    x = Dropout(0.3)(x)

    # Segunda camada convolucional
    x = Conv2D(128, (5, 5), strides=(2, 2), padding='same')(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Dropout(0.3)(x)

    # Achatar e camada de saída
    x = Flatten()(x)
    validity = Dense(1, activation='sigmoid')(x)

    return Model([img, label], validity)

# Construir e compilar o discriminador
```

```
discriminator = build_discriminator()
discriminator.compile(loss='binary_crossentropy',
                      optimizer=Adam(learning_rate=0.0002, beta_1=0.5),
                      metrics=['accuracy'])

# Construir o gerador
generator = build_generator()

# Para o modelo combinado, não atualizamos os pesos do discriminador
discriminator.trainable = False

# A entrada do modelo combinado é ruído e rótulo, a saída é a validade da imagem gerada
noise = Input(shape=(latent_dim,))
label = Input(shape=(1,))
img = generator([noise, label])
validity = discriminator([img, label])

# O modelo combinado (gerador + discriminador)
combined = Model([noise, label], validity)
combined.compile(loss='binary_crossentropy',
                 optimizer=Adam(learning_rate=0.0002, beta_1=0.5))

# Função para salvar imagens geradas
def save_imgs(epoch, generator, examples=10, dim=(2, 5), figsize=(10, 4)):
    # Gerar dígitos de 0 a 9
    noise = np.random.normal(0, 1, (examples, latent_dim))
    sampled_labels = np.array([i for i in range(examples)])

    gen_imgs = generator.predict([noise, sampled_labels])

    # Rescale images 0 - 1
    gen_imgs = 0.5 * gen_imgs + 0.5

    plt.figure(figsize=figsize)
    for i in range(examples):
        plt.subplot(dim[0], dim[1], i+1)
        plt.imshow(gen_imgs[i, :, :, 0], cmap='gray')
        plt.title(f"Digit: {sampled_labels[i]}")
        plt.axis('off')
    plt.tight_layout()
    plt.savefig(f"cgan_images/mnist_epoch_{epoch}.png")
    plt.close()

# Criar diretório para salvar imagens
import os
os.makedirs("cgan_images", exist_ok=True)

# Parâmetros de treinamento
epochs = 30000
batch_size = 128
save_interval = 1000

# Rótulos para dados reais e falsos
```

```
valid = np.ones((batch_size, 1))
fake = np.zeros((batch_size, 1))

# Histórico de perdas
d_losses = []
g_losses = []

# Loop de treinamento
for epoch in range(epochs):
    # Treinar o discriminador

    # Selecionar um batch aleatório de imagens e rótulos
    idx = np.random.randint(0, X_train.shape[0], batch_size)
    imgs = X_train[idx]
    labels = y_train[idx].reshape(-1, 1)

    # Gerar um batch de novas imagens
    noise = np.random.normal(0, 1, (batch_size, latent_dim))
    sampled_labels = np.random.randint(0, num_classes, (batch_size, 1))
    gen_imgs = generator.predict([noise, sampled_labels])

    # Treinar o discriminador
    d_loss_real = discriminator.train_on_batch([imgs, labels], valid)
    d_loss_fake = discriminator.train_on_batch([gen_imgs, sampled_labels], fake)
    d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

    # Treinar o gerador
    noise = np.random.normal(0, 1, (batch_size, latent_dim))
    sampled_labels = np.random.randint(0, num_classes, (batch_size, 1))
    g_loss = combined.train_on_batch([noise, sampled_labels], valid)

    # Registrar histórico
    d_losses.append(d_loss[0])
    g_losses.append(g_loss)

    # Imprimir progresso
    if epoch % 100 == 0:
        print(f"{epoch} [D loss: {d_loss[0]:.4f}, acc.: {100*d_loss[1]:.2f}%] [G loss: {g_loss:.4f}]")

    # Salvar imagens geradas
    if epoch % save_interval == 0:
        save_imgs(epoch, generator)

# Plotar histórico de perdas
plt.figure(figsize=(10, 5))
plt.plot(d_losses, label='Discriminator')
plt.plot(g_losses, label='Generator')
plt.title('Loss over training')
plt.legend()
plt.savefig("cgan_images/loss_history.png")
plt.close()

# Gerar e salvar um conjunto final de imagens
```

```
save_imgs(epochs, generator, examples=10, dim=(2, 5), figsize=(10, 4))
```

Aplicações de GANs

As GANs têm uma ampla gama de aplicações:

1. **Geração de Imagens**:
   - Geração de rostos realistas
   - Síntese de imagens de alta resolução
   - Geração de arte
   - Design de moda e produtos
2. **Tradução de Imagem para Imagem**:
   - Conversão de esboços para fotos realistas
   - Colorização de imagens em preto e branco
   - Restauração de imagens antigas
   - Transferência de estilo
3. **Síntese de Dados**:
   - Geração de dados sintéticos para treinamento
   - Aumento de dados para conjuntos de dados limitados
   - Preservação de privacidade através de dados sintéticos
4. **Edição de Imagens**:
   - Manipulação facial (envelhecimento, rejuvenescimento)
   - Edição semântica (mudar atributos específicos)
   - Preenchimento de imagens (inpainting)
   - Super-resolução
5. **Outras Modalidades**:
   - Geração de música
   - Síntese de voz
   - Geração de texto
   - Criação de modelos 3D

Outros Modelos Generativos

Além das GANs, existem outros modelos generativos importantes:

1. Variational Autoencoders (VAEs)

Os VAEs são modelos generativos que aprendem uma representação latente dos dados e podem gerar novas amostras. Eles consistem em: - Um encoder que mapeia dados para uma distribuição no espaço latente - Um decoder que reconstrói os dados a partir de amostras do espaço latente

Diferentemente das GANs, os VAEs têm uma função de perda bem definida e treinamento mais estável, mas geralmente produzem amostras menos nítidas.

2. Diffusion Models

Os modelos de difusão são uma classe recente de modelos generativos que aprendem a reverter um processo de difusão que gradualmente adiciona ruído aos dados. Eles têm mostrado resultados impressionantes em geração de imagens (DALL-E 2, Stable Diffusion) e áudio.

3. Autoregressive Models

Modelos autorregressivos geram dados sequencialmente, um elemento por vez, condicionando cada elemento nos elementos anteriores. Exemplos incluem PixelCNN para imagens e GPT para texto.

4. Flow-based Models

Modelos baseados em fluxo aprendem transformações invertíveis entre uma distribuição simples (como uma gaussiana) e a distribuição dos dados. Eles permitem cálculo exato da verossimilhança e geração eficiente de amostras.

Implementação de um Variational Autoencoder (VAE)

Vamos implementar um VAE simples para o dataset MNIST:

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Reshape, Flatten, Conv2D, Conv2DTranspose, BatchNorma
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.datasets import mnist
import tensorflow.keras.backend as K

# Carregar e preparar o dataset MNIST
(X_train, _), (X_test, _) = mnist.load_data()
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
X_train = np.expand_dims(X_train, axis=-1)
X_test = np.expand_dims(X_test, axis=-1)

# Parâmetros
img_rows = 28
img_cols = 28
channels = 1
img_shape = (img_rows, img_cols, channels)
latent_dim = 2  # Dimensão do espaço latente (2 para visualização)
batch_size = 128
epochs = 50

# Encoder
def build_encoder():
    inputs = Input(shape=img_shape)

    x = Conv2D(32, 3, strides=2, padding='same')(inputs)
    x = LeakyReLU(0.2)(x)

    x = Conv2D(64, 3, strides=2, padding='same')(x)
    x = LeakyReLU(0.2)(x)

    x = Conv2D(128, 3, strides=2, padding='same')(x)
    x = LeakyReLU(0.2)(x)

    x = Flatten()(x)
    x = Dense(256)(x)
    x = LeakyReLU(0.2)(x)

    # Média e log variância do espaço latente
    z_mean = Dense(latent_dim)(x)
    z_log_var = Dense(latent_dim)(x)
```

```
    # Função de amostragem
    def sampling(args):
        z_mean, z_log_var = args
        batch = K.shape(z_mean)[0]
        dim = K.int_shape(z_mean)[1]
        epsilon = K.random_normal(shape=(batch, dim))
        return z_mean + K.exp(0.5 * z_log_var) * epsilon

    z = Lambda(sampling)([z_mean, z_log_var])

    encoder = Model(inputs, [z_mean, z_log_var, z], name='encoder')
    return encoder

# Decoder
def build_decoder():
    latent_inputs = Input(shape=(latent_dim,))

    x = Dense(7 * 7 * 128)(latent_inputs)
    x = LeakyReLU(0.2)(x)
    x = Reshape((7, 7, 128))(x)

    x = Conv2DTranspose(128, 3, strides=2, padding='same')(x)
    x = LeakyReLU(0.2)(x)

    x = Conv2DTranspose(64, 3, strides=2, padding='same')(x)
    x = LeakyReLU(0.2)(x)

    outputs = Conv2DTranspose(1, 3, activation='sigmoid', padding='same')(x)

    decoder = Model(latent_inputs, outputs, name='decoder')
    return decoder

# Construir o VAE
encoder = build_encoder()
decoder = build_decoder()

inputs = Input(shape=img_shape)
z_mean, z_log_var, z = encoder(inputs)
outputs = decoder(z)

# Função de perda do VAE
def vae_loss(inputs, outputs):
    # Reconstrução
    reconstruction_loss = tf.keras.losses.binary_crossentropy(
        K.flatten(inputs), K.flatten(outputs)) * img_rows * img_cols

    # KL divergence
    kl_loss = -0.5 * K.sum(1 + z_log_var - K.square(z_mean) - K.exp(z_log_var), axis=-1)

    return K.mean(reconstruction_loss + kl_loss)

# Modelo VAE completo
vae = Model(inputs, outputs, name='vae')
```

```
vae.compile(optimizer=Adam(learning_rate=0.0001), loss=vae_loss)

# Treinar o VAE
vae.fit(X_train, X_train,
        epochs=epochs,
        batch_size=batch_size,
        validation_data=(X_test, X_test))

# Visualizar o espaço latente
def plot_latent_space(encoder, decoder):
    # Exibir uma grade 30x30 de dígitos no espaço latente
    n = 30
    digit_size = 28
    figure = np.zeros((digit_size * n, digit_size * n))

    # Criar uma grade de valores z
    grid_x = np.linspace(-3, 3, n)
    grid_y = np.linspace(-3, 3, n)[::-1]

    for i, yi in enumerate(grid_y):
        for j, xi in enumerate(grid_x):
            z_sample = np.array([[xi, yi]])
            x_decoded = decoder.predict(z_sample)
            digit = x_decoded[0].reshape(digit_size, digit_size)
            figure[i * digit_size: (i + 1) * digit_size,
                   j * digit_size: (j + 1) * digit_size] = digit

    plt.figure(figsize=(10, 10))
    start_range = digit_size // 2
    end_range = n * digit_size + start_range
    pixel_range = np.arange(start_range, end_range, digit_size)
    sample_range_x = np.round(grid_x, 1)
    sample_range_y = np.round(grid_y, 1)
    plt.xticks(pixel_range, sample_range_x)
    plt.yticks(pixel_range, sample_range_y)
    plt.xlabel("z[0]")
    plt.ylabel("z[1]")
    plt.imshow(figure, cmap='Greys_r')
    plt.savefig('vae_latent_space.png')
    plt.close()

# Visualizar reconstruções
def plot_reconstructions(encoder, decoder, n=10):
    # Exibir dígitos originais e reconstruídos
    X_test_sample = X_test[:n]

    z_mean, _, _ = encoder.predict(X_test_sample)
    X_test_reconstructed = decoder.predict(z_mean)

    plt.figure(figsize=(20, 4))
    for i in range(n):
        # Original
        ax = plt.subplot(2, n, i + 1)
```

```
        plt.imshow(X_test_sample[i].reshape(28, 28), cmap='Greys_r')
        plt.title("Original")
        plt.axis('off')

        # Reconstrução
        ax = plt.subplot(2, n, i + 1 + n)
        plt.imshow(X_test_reconstructed[i].reshape(28, 28), cmap='Greys_r')
        plt.title("Reconstructed")
        plt.axis('off')

    plt.savefig('vae_reconstructions.png')
    plt.close()

# Visualizar o espaço latente e reconstruções
plot_latent_space(encoder, decoder)
plot_reconstructions(encoder, decoder)
```

Desafios e Avanços Recentes

Desafios

1. **Estabilidade de Treinamento**: As GANs são notoriamente difíceis de treinar e podem sofrer de problemas como modo collapse e não-convergência.
2. **Avaliação**: É difícil avaliar objetivamente a qualidade e diversidade das amostras geradas.
3. **Controle Semântico**: Controlar atributos específicos nas amostras geradas pode ser desafiador.
4. **Escalabilidade**: Treinar GANs para gerar imagens de alta resolução requer recursos computacionais significativos.

Avanços Recentes

1. **StyleGAN3**: Melhora a qualidade e controle da geração de imagens, com foco em invariância a transformações.
2. **Diffusion Models**: Modelos como DALL-E 2, Midjourney e Stable Diffusion têm superado as GANs em muitas tarefas de geração de imagens.
3. **GANs 3D**: Extensão das GANs para geração de modelos 3D e cenas.
4. **Modelos Multimodais**: GANs que podem trabalhar com múltiplas modalidades, como texto e imagem.

Conclusão

As Generative Adversarial Networks representam uma abordagem poderosa para modelagem generativa, permitindo a criação de conteúdo novo e realista em diversas modalidades. Apesar dos desafios de treinamento, as GANs e suas variantes têm aplicações impressionantes em geração de imagens, tradução de imagem para imagem, síntese de dados e muito mais.

Outros modelos generativos como VAEs, modelos de difusão e modelos autorregressivos oferecem abordagens alternativas com suas próprias vantagens e desvantagens. A escolha do modelo depende da aplicação específica, dos recursos disponíveis e dos requisitos de qualidade e controle.

À medida que a pesquisa avança, podemos esperar modelos generativos ainda mais poderosos, capazes de criar conteúdo cada vez mais realista e útil para uma ampla gama de aplicações.

Módulo 5: Redes Neurais e Deep Learning

Aplicações Práticas e Projetos

Neste tópico final sobre redes neurais e deep learning, vamos explorar aplicações práticas e projetos que demonstram como utilizar as técnicas e arquiteturas que estudamos nos tópicos anteriores. Abordaremos projetos em diferentes domínios, desde visão computacional até processamento de linguagem natural e séries temporais, fornecendo exemplos concretos que você pode adaptar e expandir para seus próprios projetos.

Projeto 1: Sistema de Reconhecimento de Imagens

Neste projeto, vamos criar um sistema de classificação de imagens usando transfer learning com uma rede neural convolucional pré-treinada.

Objetivo

Desenvolver um classificador de imagens capaz de identificar diferentes categorias de objetos ou cenas.

Implementação

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau, ModelCheckpoint
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
import os

# Configurações
img_width, img_height = 224, 224
batch_size = 32
epochs = 20
num_classes = 5  # Ajuste para o número de classes no seu dataset

# Diretórios de dados
train_dir = 'data/train'
```

```
validation_dir = 'data/validation'
test_dir = 'data/test'

# Verificar se os diretórios existem
for directory in [train_dir, validation_dir, test_dir]:
    if not os.path.exists(directory):
        print(f"Aviso: O diretório {directory} não existe. Crie-o e adicione suas imagens.")

# Data augmentation para o conjunto de treinamento
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

# Apenas rescale para validação e teste
validation_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

# Carregar os dados
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    class_mode='categorical'
)

validation_generator = validation_datagen.flow_from_directory(
    validation_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    class_mode='categorical'
)

test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False
)

# Obter mapeamento de classes
class_indices = train_generator.class_indices
class_names = list(class_indices.keys())
print("Classes:", class_names)

# Carregar o modelo ResNet50 pré-treinado
```

```
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(img_width, img_height, 3))

# Congelar as camadas do modelo base
for layer in base_model.layers:
    layer.trainable = False

# Adicionar novas camadas no topo
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(512, activation='relu')(x)
x = Dropout(0.5)(x)
predictions = Dense(num_classes, activation='softmax')(x)

# Criar o modelo final
model = Model(inputs=base_model.input, outputs=predictions)

# Compilar o modelo
model.compile(optimizer=Adam(learning_rate=0.0001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Resumo do modelo
model.summary()

# Callbacks
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=1e-6)
checkpoint = ModelCheckpoint('best_model.h5', monitor='val_accuracy', save_best_only=True, mode='max')

# Treinar o modelo
history = model.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // batch_size,
    epochs=epochs,
    validation_data=validation_generator,
    validation_steps=validation_generator.samples // batch_size,
    callbacks=[early_stopping, reduce_lr, checkpoint]
)

# Plotar o histórico de treinamento
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Treino')
plt.plot(history.history['val_accuracy'], label='Validação')
plt.xlabel('Época')
plt.ylabel('Acurácia')
plt.title('Acurácia ao longo do treinamento')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
```

```
plt.plot(history.history['loss'], label='Treino')
plt.plot(history.history['val_loss'], label='Validação')
plt.xlabel('Época')
plt.ylabel('Perda')
plt.title('Perda ao longo do treinamento')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig('training_history.png')
plt.show()

# Avaliar o modelo no conjunto de teste
test_loss, test_acc = model.evaluate(test_generator)
print(f'Acurácia no conjunto de teste: {test_acc:.4f}')

# Fazer previsões no conjunto de teste
predictions = model.predict(test_generator)
y_pred = np.argmax(predictions, axis=1)

# Obter rótulos verdadeiros
y_true = test_generator.classes

# Relatório de classificação
print("\nRelatório de Classificação:")
print(classification_report(y_true, y_pred, target_names=class_names))

# Matriz de confusão
plt.figure(figsize=(10, 8))
cm = confusion_matrix(y_true, y_pred)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, yticklabels=class_names)
plt.xlabel('Previsão')
plt.ylabel('Valor Real')
plt.title('Matriz de Confusão')
plt.tight_layout()
plt.savefig('confusion_matrix.png')
plt.show()

# Fine-tuning: descongelar algumas camadas do modelo base
print("\nIniciando fine-tuning...")
for layer in base_model.layers[-30:]:  # Descongelar as últimas 30 camadas
    layer.trainable = True

# Recompilar o modelo com uma taxa de aprendizado menor
model.compile(optimizer=Adam(learning_rate=1e-5),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Continuar o treinamento com fine-tuning
history_ft = model.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // batch_size,
    epochs=10,
    validation_data=validation_generator,
```

```
    validation_steps=validation_generator.samples // batch_size,
    callbacks=[early_stopping, reduce_lr, checkpoint]
)

# Avaliar o modelo após fine-tuning
test_loss, test_acc = model.evaluate(test_generator)
print(f'Acurácia no conjunto de teste após fine-tuning: {test_acc:.4f}')

# Fazer previsões após fine-tuning
predictions = model.predict(test_generator)
y_pred = np.argmax(predictions, axis=1)

# Relatório de classificação após fine-tuning
print("\nRelatório de Classificação após Fine-tuning:")
print(classification_report(y_true, y_pred, target_names=class_names))

# Matriz de confusão após fine-tuning
plt.figure(figsize=(10, 8))
cm = confusion_matrix(y_true, y_pred)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, yticklabels=class_names)
plt.xlabel('Previsão')
plt.ylabel('Valor Real')
plt.title('Matriz de Confusão após Fine-tuning')
plt.tight_layout()
plt.savefig('confusion_matrix_ft.png')
plt.show()

# Salvar o modelo final
model.save('image_classifier_model.h5')
print("Modelo salvo como 'image_classifier_model.h5'")

# Função para fazer previsões em novas imagens
def predict_image(image_path, model, class_names):
    from tensorflow.keras.preprocessing import image

    img = image.load_img(image_path, target_size=(img_width, img_height))
    img_array = image.img_to_array(img)
    img_array = np.expand_dims(img_array, axis=0)
    img_array = img_array / 255.0

    prediction = model.predict(img_array)
    predicted_class = np.argmax(prediction, axis=1)[0]
    confidence = prediction[0][predicted_class]

    plt.figure(figsize=(6, 6))
    plt.imshow(img)
    plt.title(f"Classe: {class_names[predicted_class]}\nConfiança: {confidence:.2f}")
    plt.axis('off')
    plt.show()

    return class_names[predicted_class], confidence

# Exemplo de uso da função de previsão
```

```
# predict_image('caminho/para/sua/imagem.jpg', model, class_names)
```

Extensões Possíveis

1. Implementar visualização de mapas de ativação para entender o que o modelo está "olhando"
2. Adicionar explicabilidade com técnicas como LIME ou SHAP
3. Desenvolver uma interface web simples para fazer upload e classificar imagens
4. Adaptar para detecção de objetos usando modelos como YOLO ou SSD

Projeto 2: Análise de Sentimento em Textos

Neste projeto, vamos criar um sistema de análise de sentimento usando BERT para classificar textos como positivos, negativos ou neutros.

Objetivo

Desenvolver um classificador de sentimento capaz de analisar o tom emocional de textos.

Implementação

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import torch
from torch.utils.data import Dataset, DataLoader, random_split
from transformers import BertTokenizer, BertForSequenceClassification, AdamW
from transformers import get_linear_schedule_with_warmup
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import seaborn as sns
import time
import random
import os

# Configurar seed para reprodutibilidade
seed_val = 42
random.seed(seed_val)
np.random.seed(seed_val)
torch.manual_seed(seed_val)
torch.cuda.manual_seed_all(seed_val)

# Verificar se GPU está disponível
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Usando dispositivo: {device}")

# Carregar dados (exemplo com dataset de reviews)
# Substitua pelo seu próprio dataset ou use um dataset público
def load_data():
    # Exemplo: carregar de um CSV
    # df = pd.read_csv('seu_dataset.csv')
    # return df

    # Para este exemplo, vamos criar dados sintéticos
    texts = [
        "Este produto é incrível, superou todas as minhas expectativas!",
```

```
        "Não gostei nada, péssima qualidade e atendimento horrível.",
        "O produto é bom, mas o preço é um pouco alto.",
        "Adorei! Recomendo para todos os meus amigos.",
        "Experiência terrível, não compraria novamente.",
        "Produto mediano, nada excepcional mas cumpre o que promete.",
        "Excelente custo-benefício, muito satisfeito com a compra.",
        "Decepcionante, esperava muito mais pelo preço que paguei.",
        "Neutro, nem bom nem ruim, apenas ok.",
        "Fantástico! Um dos melhores produtos que já comprei."
    ]

    # 0: negativo, 1: neutro, 2: positivo
    labels = [2, 0, 1, 2, 0, 1, 2, 0, 1, 2]

    return pd.DataFrame({'text': texts, 'label': labels})

# Carregar os dados
df = load_data()
print(f"Dataset carregado com {len(df)} exemplos")
print(df.head())

# Verificar distribuição de classes
class_distribution = df['label'].value_counts()
print("\nDistribuição de classes:")
print(class_distribution)

# Mapear rótulos para nomes de classes
label_map = {0: "Negativo", 1: "Neutro", 2: "Positivo"}

# Criar dataset personalizado
class SentimentDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_length=128):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        text = self.texts[idx]
        label = self.labels[idx]

        encoding = self.tokenizer(
            text,
            add_special_tokens=True,
            max_length=self.max_length,
            padding='max_length',
            truncation=True,
            return_tensors='pt'
        )
```

```
        return {
            'input_ids': encoding['input_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].flatten(),
            'labels': torch.tensor(label, dtype=torch.long)
        }

# Carregar o tokenizador e o modelo pré-treinado
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertForSequenceClassification.from_pretrained(
    'bert-base-uncased',
    num_labels=3,  # 3 classes: negativo, neutro, positivo
    output_attentions=False,
    output_hidden_states=False
)

model.to(device)

# Criar dataset
dataset = SentimentDataset(df['text'].values, df['label'].values, tokenizer)

# Dividir em treino e teste (80% treino, 20% teste)
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = random_split(dataset, [train_size, test_size])

print(f"Tamanho do conjunto de treino: {train_size}")
print(f"Tamanho do conjunto de teste: {test_size}")

# Criar dataloaders
batch_size = 8
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size)

# Configurar otimizador
optimizer = AdamW(model.parameters(), lr=2e-5, eps=1e-8)

# Número de épocas
epochs = 4

# Total de passos de treinamento
total_steps = len(train_dataloader) * epochs

# Criar scheduler para ajustar a taxa de aprendizado
scheduler = get_linear_schedule_with_warmup(
    optimizer,
    num_warmup_steps=0,
    num_training_steps=total_steps
)

# Função para calcular a acurácia
def flat_accuracy(preds, labels):
    pred_flat = np.argmax(preds, axis=1).flatten()
    labels_flat = labels.flatten()
```

```
    return accuracy_score(labels_flat, pred_flat)

# Função de treinamento
def train():
    # Armazenar estatísticas de treinamento
    training_stats = []

    # Medir o tempo total de treinamento
    total_t0 = time.time()

    # Para cada época...
    for epoch_i in range(0, epochs):
        print(f"\n======== Época {epoch_i + 1} / {epochs} ========")
        print("Treinando...")

        # Medir o tempo da época
        t0 = time.time()

        # Resetar a perda total para esta época
        total_train_loss = 0

        # Colocar o modelo em modo de treinamento
        model.train()

        # Para cada batch de dados de treinamento...
        for step, batch in enumerate(train_dataloader):
            # Progresso
            if step % 10 == 0 and not step == 0:
                elapsed = time.time() - t0
                print(f"  Batch {step}  de  {len(train_dataloader)}.    Tempo decorrido: {elapsed:.2f}s

            # Desempacotar o batch e mover para o dispositivo
            b_input_ids = batch['input_ids'].to(device)
            b_attention_mask = batch['attention_mask'].to(device)
            b_labels = batch['labels'].to(device)

            # Zerar gradientes
            model.zero_grad()

            # Forward pass
            outputs = model(
                b_input_ids,
                token_type_ids=None,
                attention_mask=b_attention_mask,
                labels=b_labels
            )

            loss = outputs.loss
            total_train_loss += loss.item()

            # Backward pass
            loss.backward()
```

```
        # Clip norm para evitar explosão do gradiente
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

        # Atualizar parâmetros
        optimizer.step()

        # Atualizar learning rate
        scheduler.step()

    # Calcular a perda média de treinamento
    avg_train_loss = total_train_loss / len(train_dataloader)

    # Medir quanto tempo levou o treinamento
    training_time = time.time() - t0

    print(f"\n  Perda média de treinamento: {avg_train_loss:.2f}")
    print(f"  Tempo de treinamento: {training_time:.2f}s")

    # ========== Validação ==========

    print("\nValidando...")

    # Colocar o modelo em modo de avaliação
    model.eval()

    # Rastrear variáveis
    total_eval_accuracy = 0
    total_eval_loss = 0

    # Avaliar dados por batch
    for batch in test_dataloader:
        # Desempacotar o batch e mover para o dispositivo
        b_input_ids = batch['input_ids'].to(device)
        b_attention_mask = batch['attention_mask'].to(device)
        b_labels = batch['labels'].to(device)

        # Sem cálculo de gradientes
        with torch.no_grad():
            # Forward pass
            outputs = model(
                b_input_ids,
                token_type_ids=None,
                attention_mask=b_attention_mask,
                labels=b_labels
            )

        loss = outputs.loss
        logits = outputs.logits

        # Acumular perda
        total_eval_loss += loss.item()

        # Mover logits e labels para CPU
```

```
            logits = logits.detach().cpu().numpy()
            label_ids = b_labels.to('cpu').numpy()

            # Calcular a acurácia para este batch
            total_eval_accuracy += flat_accuracy(logits, label_ids)

        # Calcular a acurácia média
        avg_val_accuracy = total_eval_accuracy / len(test_dataloader)
        print(f"  Acurácia: {avg_val_accuracy:.2f}")

        # Calcular a perda média de validação
        avg_val_loss = total_eval_loss / len(test_dataloader)
        print(f"  Perda de validação: {avg_val_loss:.2f}")

        # Armazenar estatísticas
        training_stats.append({
            'epoch': epoch_i + 1,
            'train_loss': avg_train_loss,
            'val_loss': avg_val_loss,
            'val_accuracy': avg_val_accuracy,
            'training_time': training_time
        })

    print("\nTreinamento completo!")
    print(f"Tempo total de treinamento: {(time.time() - total_t0):.2f}s")

    return training_stats

# Treinar o modelo
training_stats = train()

# Plotar estatísticas de treinamento
stats_df = pd.DataFrame(training_stats)
stats_df = stats_df.set_index('epoch')

# Plotar perda de treinamento e validação
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(stats_df['train_loss'], label='Treino')
plt.plot(stats_df['val_loss'], label='Validação')
plt.title('Perda de Treinamento e Validação')
plt.xlabel('Época')
plt.ylabel('Perda')
plt.legend()
plt.grid(True)

# Plotar acurácia de validação
plt.subplot(1, 2, 2)
plt.plot(stats_df['val_accuracy'], label='Validação')
plt.title('Acurácia de Validação')
plt.xlabel('Época')
plt.ylabel('Acurácia')
plt.legend()
```

```
plt.grid(True)
plt.tight_layout()
plt.savefig('bert_training_stats.png')
plt.show()

# Avaliar o modelo no conjunto de teste
def evaluate_model():
    # Colocar o modelo em modo de avaliação
    model.eval()

    # Rastrear variáveis
    predictions = []
    true_labels = []

    # Avaliar dados por batch
    for batch in test_dataloader:
        # Desempacotar o batch e mover para o dispositivo
        b_input_ids = batch['input_ids'].to(device)
        b_attention_mask = batch['attention_mask'].to(device)
        b_labels = batch['labels'].to(device)

        # Sem cálculo de gradientes
        with torch.no_grad():
            # Forward pass
            outputs = model(
                b_input_ids,
                token_type_ids=None,
                attention_mask=b_attention_mask
            )

        logits = outputs.logits

        # Mover logits e labels para CPU
        logits = logits.detach().cpu().numpy()
        label_ids = b_labels.to('cpu').numpy()

        # Armazenar previsões e rótulos verdadeiros
        predictions.extend(np.argmax(logits, axis=1).flatten())
        true_labels.extend(label_ids.flatten())

    # Calcular métricas
    accuracy = accuracy_score(true_labels, predictions)
    report = classification_report(true_labels, predictions, target_names=list(label_map.values()))

    print(f"Acurácia no conjunto de teste: {accuracy:.4f}")
    print("\nRelatório de Classificação:")
    print(report)

    # Matriz de confusão
    cm = confusion_matrix(true_labels, predictions)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=list(label_map.values()), yticklabels
    plt.xlabel('Previsão')
```

```
    plt.ylabel('Valor Real')
    plt.title('Matriz de Confusão')
    plt.tight_layout()
    plt.savefig('bert_confusion_matrix.png')
    plt.show()

    return accuracy, report, cm

# Avaliar o modelo
accuracy, report, cm = evaluate_model()

# Salvar o modelo e o tokenizador
output_dir = './bert_sentiment_model/'
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

model.save_pretrained(output_dir)
tokenizer.save_pretrained(output_dir)
print(f"Modelo salvo em {output_dir}")

# Função para fazer previsões em novos textos
def predict_sentiment(text, model, tokenizer):
    # Colocar o modelo em modo de avaliação
    model.eval()

    # Tokenizar o texto
    encoding = tokenizer(
        text,
        add_special_tokens=True,
        max_length=128,
        padding='max_length',
        truncation=True,
        return_tensors='pt'
    )

    # Mover para o dispositivo
    input_ids = encoding['input_ids'].to(device)
    attention_mask = encoding['attention_mask'].to(device)

    # Fazer a previsão
    with torch.no_grad():
        outputs = model(
            input_ids,
            token_type_ids=None,
            attention_mask=attention_mask
        )

    logits = outputs.logits

    # Obter a classe prevista
    predicted_class = torch.argmax(logits, dim=1).item()

    # Obter as probabilidades
```

```
    probs = torch.nn.functional.softmax(logits, dim=1).cpu().numpy()[0]

    return {
        'text': text,
        'sentiment': label_map[predicted_class],
        'confidence': probs[predicted_class],
        'probabilities': {label_map[i]: float(prob) for i, prob in enumerate(probs)}
    }

# Exemplo de uso da função de previsão
test_texts = [
    "Este produto é simplesmente maravilhoso, superou todas as minhas expectativas!",
    "Não recomendo, péssima qualidade e preço alto demais.",
    "O produto é razoável, tem alguns pontos positivos e negativos."
]

for text in test_texts:
    result = predict_sentiment(text, model, tokenizer)
    print(f"\nTexto: {result['text']}")
    print(f"Sentimento: {result['sentiment']}")
    print(f"Confiança: {result['confidence']:.4f}")
    print("Probabilidades:")
    for sentiment, prob in result['probabilities'].items():
        print(f"  {sentiment}: {prob:.4f}")
```

Extensões Possíveis

1. Implementar análise de sentimento em tempo real para mídias sociais
2. Adicionar detecção de emoções mais granular (alegria, tristeza, raiva, etc.)
3. Criar uma API REST para integração com outros sistemas
4. Adicionar visualização de palavras-chave que influenciaram a classificação

Projeto 3: Previsão de Séries Temporais

Neste projeto, vamos criar um sistema de previsão de séries temporais usando LSTM para prever valores futuros com base em dados históricos.

Objetivo

Desenvolver um modelo capaz de prever valores futuros em uma série temporal.

Implementação

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout, Bidirectional
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import seaborn as sns
```

```
import os
import datetime

# Configurar seed para reprodutibilidade
np.random.seed(42)
tf.random.set_seed(42)

# Função para carregar dados
def load_data():
    # Exemplo: carregar de um CSV
    # df = pd.read_csv('seu_dataset.csv', parse_dates=['date'], index_col='date')
    # return df

    # Para este exemplo, vamos criar dados sintéticos
    # Simulando dados diários por 3 anos
    dates = pd.date_range(start='2020-01-01', end='2022-12-31', freq='D')
    n = len(dates)

    # Tendência
    trend = np.linspace(0, 10, n)

    # Sazonalidade (anual e semanal)
    annual_seasonality = 5 * np.sin(2 * np.pi * np.arange(n) / 365)
    weekly_seasonality = 2 * np.sin(2 * np.pi * np.arange(n) / 7)

    # Ruído
    noise = np.random.normal(0, 1, n)

    # Série temporal final
    values = trend + annual_seasonality + weekly_seasonality + noise

    # Criar DataFrame
    df = pd.DataFrame({'value': values}, index=dates)

    return df

# Carregar os dados
df = load_data()
print(f"Dataset carregado com {len(df)} exemplos")
print(df.head())

# Visualizar a série temporal
plt.figure(figsize=(15, 6))
plt.plot(df.index, df['value'])
plt.title('Série Temporal')
plt.xlabel('Data')
plt.ylabel('Valor')
plt.grid(True)
plt.tight_layout()
plt.savefig('time_series_data.png')
plt.show()

# Preparar os dados para o modelo LSTM
```

```
def create_sequences(data, seq_length):
    xs, ys = [], []
    for i in range(len(data) - seq_length):
        x = data[i:i+seq_length]
        y = data[i+seq_length]
        xs.append(x)
        ys.append(y)
    return np.array(xs), np.array(ys)

# Normalizar os dados
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(df[['value']])

# Parâmetros
seq_length = 30  # Usar 30 dias para prever o próximo dia
train_size = int(len(scaled_data) * 0.8)

# Dividir em treino e teste
train_data = scaled_data[:train_size]
test_data = scaled_data[train_size-seq_length:]

# Criar sequências
X_train, y_train = create_sequences(train_data, seq_length)
X_test, y_test = create_sequences(test_data, seq_length)

print(f"Forma de X_train: {X_train.shape}")
print(f"Forma de y_train: {y_train.shape}")
print(f"Forma de X_test: {X_test.shape}")
print(f"Forma de y_test: {y_test.shape}")

# Construir o modelo LSTM
def build_model(seq_length):
    model = Sequential([
        Bidirectional(LSTM(50, return_sequences=True), input_shape=(seq_length, 1)),
        Dropout(0.2),
        Bidirectional(LSTM(50)),
        Dropout(0.2),
        Dense(1)
    ])

    model.compile(optimizer=Adam(learning_rate=0.001), loss='mse')
    return model

# Criar o modelo
model = build_model(seq_length)
model.summary()

# Callbacks
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
checkpoint = ModelCheckpoint('best_lstm_model.h5', monitor='val_loss', save_best_only=True, mode='min')
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=1e-6)

# Treinar o modelo
```

```
history = model.fit(
    X_train, y_train,
    epochs=100,
    batch_size=32,
    validation_split=0.2,
    callbacks=[early_stopping, checkpoint, reduce_lr],
    verbose=1
)

# Plotar o histórico de treinamento
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Treino')
plt.plot(history.history['val_loss'], label='Validação')
plt.title('Perda do Modelo')
plt.xlabel('Época')
plt.ylabel('Perda (MSE)')
plt.legend()
plt.grid(True)
plt.savefig('lstm_training_history.png')
plt.show()

# Fazer previsões
train_predictions = model.predict(X_train)
test_predictions = model.predict(X_test)

# Inverter a normalização
train_predictions = scaler.inverse_transform(train_predictions)
y_train_inv = scaler.inverse_transform(y_train.reshape(-1, 1))
test_predictions = scaler.inverse_transform(test_predictions)
y_test_inv = scaler.inverse_transform(y_test.reshape(-1, 1))

# Calcular métricas
train_rmse = np.sqrt(mean_squared_error(y_train_inv, train_predictions))
test_rmse = np.sqrt(mean_squared_error(y_test_inv, test_predictions))
train_mae = mean_absolute_error(y_train_inv, train_predictions)
test_mae = mean_absolute_error(y_test_inv, test_predictions)
train_r2 = r2_score(y_train_inv, train_predictions)
test_r2 = r2_score(y_test_inv, test_predictions)

print(f'RMSE (Treino): {train_rmse:.4f}')
print(f'RMSE (Teste): {test_rmse:.4f}')
print(f'MAE (Treino): {train_mae:.4f}')
print(f'MAE (Teste): {test_mae:.4f}')
print(f'R² (Treino): {train_r2:.4f}')
print(f'R² (Teste): {test_r2:.4f}')

# Visualizar resultados
# Preparar índices para plotagem
train_dates = df.index[seq_length:train_size]
test_dates = df.index[train_size:][:len(test_predictions)]

plt.figure(figsize=(15, 6))
plt.plot(df.index, df['value'], label='Dados Reais', alpha=0.7)
```

```
plt.plot(train_dates, train_predictions, label='Previsões (Treino)', alpha=0.7)
plt.plot(test_dates, test_predictions, label='Previsões (Teste)', alpha=0.7)
plt.title('Previsões vs. Valores Reais')
plt.xlabel('Data')
plt.ylabel('Valor')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig('lstm_predictions.png')
plt.show()

# Zoom nos dados de teste
plt.figure(figsize=(15, 6))
plt.plot(test_dates, df.loc[test_dates, 'value'], label='Dados Reais', alpha=0.7)
plt.plot(test_dates, test_predictions, label='Previsões', alpha=0.7)
plt.title('Previsões vs. Valores Reais (Conjunto de Teste)')
plt.xlabel('Data')
plt.ylabel('Valor')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig('lstm_test_predictions.png')
plt.show()

# Previsão de valores futuros
def forecast_future(model, last_sequence, steps, scaler):
    future_predictions = []
    current_sequence = last_sequence.copy()

    for _ in range(steps):
        # Prever o próximo valor
        current_sequence_reshaped = current_sequence.reshape(1, seq_length, 1)
        next_pred = model.predict(current_sequence_reshaped)

        # Adicionar à lista de previsões
        future_predictions.append(next_pred[0, 0])

        # Atualizar a sequência para a próxima previsão
        current_sequence = np.append(current_sequence[1:], next_pred)

    # Inverter a normalização
    future_predictions = scaler.inverse_transform(np.array(future_predictions).reshape(-1, 1))
    return future_predictions

# Obter a última sequência do conjunto de dados
last_sequence = scaled_data[-seq_length:]

# Prever valores futuros (próximos 30 dias)
future_steps = 30
future_predictions = forecast_future(model, last_sequence, future_steps, scaler)

# Criar datas futuras
last_date = df.index[-1]
```

```
future_dates = pd.date_range(start=last_date + pd.Timedelta(days=1), periods=future_steps)

# Visualizar previsões futuras
plt.figure(figsize=(15, 6))
plt.plot(df.index[-90:], df['value'][-90:], label='Dados Históricos', alpha=0.7)
plt.plot(future_dates, future_predictions, label='Previsões Futuras', alpha=0.7, color='red')
plt.title('Previsões Futuras')
plt.xlabel('Data')
plt.ylabel('Valor')
plt.axvline(x=last_date, color='k', linestyle='--', alpha=0.5)
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig('lstm_future_predictions.png')
plt.show()

# Salvar o modelo
model.save('time_series_lstm_model.h5')
print("Modelo salvo como 'time_series_lstm_model.h5'")

# Criar uma função para fazer previsões com o modelo salvo
def load_and_predict(model_path, new_data, seq_length, scaler):
    # Carregar o modelo
    loaded_model = tf.keras.models.load_model(model_path)

    # Normalizar os dados
    scaled_new_data = scaler.transform(new_data.reshape(-1, 1))

    # Criar sequência
    X_new = scaled_new_data[-seq_length:].reshape(1, seq_length, 1)

    # Fazer previsão
    prediction = loaded_model.predict(X_new)

    # Inverter a normalização
    prediction = scaler.inverse_transform(prediction)

    return prediction[0, 0]

# Exemplo de uso da função de previsão
# Supondo que temos novos dados
new_data = df['value'][-seq_length:].values
prediction = load_and_predict('time_series_lstm_model.h5', new_data, seq_length, scaler)
print(f"Previsão para o próximo período: {prediction:.4f}")
```

Extensões Possíveis

1. Implementar modelos mais avançados como Temporal Fusion Transformers
2. Adicionar variáveis exógenas (como dados meteorológicos para previsão de demanda)
3. Implementar previsão probabilística para quantificar a incerteza
4. Criar um dashboard interativo para visualização e atualização das previsões

Projeto 4: Geração de Imagens com GANs

Neste projeto, vamos criar um sistema de geração de imagens usando Generative Adversarial Networks (GANs).

Objetivo

Desenvolver um modelo capaz de gerar imagens realistas de uma categoria específica.

Implementação

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Reshape, Flatten, Dropout, LeakyReLU, Conv2D, Conv2DT
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.datasets import fashion_mnist
import os
import time

# Configurar seed para reprodutibilidade
np.random.seed(42)
tf.random.set_seed(42)

# Carregar e preparar o dataset Fashion MNIST
(X_train, _), (_, _) = fashion_mnist.load_data()
X_train = X_train / 127.5 - 1.0  # Normalizar para [-1, 1]
X_train = np.expand_dims(X_train, axis=-1)  # Adicionar canal

# Parâmetros
img_rows = 28
img_cols = 28
channels = 1
img_shape = (img_rows, img_cols, channels)
latent_dim = 100

# Construir o gerador
def build_generator():
    model = Sequential()

    # Camada densa inicial
    model.add(Dense(7*7*256, use_bias=False, input_shape=(latent_dim,)))
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha=0.2))

    # Reshape para começar o processo de upsampling
    model.add(Reshape((7, 7, 256)))

    # Upsampling para 14x14
    model.add(Conv2DTranspose(128, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha=0.2))

    # Upsampling para 28x28
```

```
    model.add(Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha=0.2))

    # Camada de saída
    model.add(Conv2DTranspose(1, (5, 5), strides=(1, 1), padding='same', use_bias=False, activation='ta

    # Resumo do modelo
    model.summary()

    # Entrada de ruído e saída de imagem gerada
    noise = Input(shape=(latent_dim,))
    img = model(noise)

    return Model(noise, img)

# Construir o discriminador
def build_discriminator():
    model = Sequential()

    # Primeira camada convolucional
    model.add(Conv2D(64, (5, 5), strides=(2, 2), padding='same', input_shape=img_shape))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.3))

    # Segunda camada convolucional
    model.add(Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.3))

    # Achatar e camada de saída
    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))

    # Resumo do modelo
    model.summary()

    # Entrada de imagem e saída de classificação
    img = Input(shape=img_shape)
    validity = model(img)

    return Model(img, validity)

# Construir e compilar o discriminador
discriminator = build_discriminator()
discriminator.compile(loss='binary_crossentropy',
                      optimizer=Adam(learning_rate=0.0002, beta_1=0.5),
                      metrics=['accuracy'])

# Construir o gerador
generator = build_generator()

# Para o modelo combinado, não atualizamos os pesos do discriminador
```

```
discriminator.trainable = False

# A entrada do modelo combinado é ruído, a saída é a validade da imagem gerada
z = Input(shape=(latent_dim,))
img = generator(z)
validity = discriminator(img)

# O modelo combinado (gerador + discriminador)
combined = Model(z, validity)
combined.compile(loss='binary_crossentropy',
                 optimizer=Adam(learning_rate=0.0002, beta_1=0.5))

# Função para salvar imagens geradas
def save_imgs(epoch, generator, examples=25, dim=(5, 5), figsize=(10, 10)):
    noise = np.random.normal(0, 1, (examples, latent_dim))
    gen_imgs = generator.predict(noise)

    # Rescale images 0 - 1
    gen_imgs = 0.5 * gen_imgs + 0.5

    plt.figure(figsize=figsize)
    for i in range(examples):
        plt.subplot(dim[0], dim[1], i+1)
        plt.imshow(gen_imgs[i, :, :, 0], cmap='gray')
        plt.axis('off')
    plt.tight_layout()
    plt.savefig(f"gan_fashion/fashion_epoch_{epoch}.png")
    plt.close()

# Criar diretório para salvar imagens
os.makedirs("gan_fashion", exist_ok=True)

# Parâmetros de treinamento
epochs = 30000
batch_size = 128
save_interval = 1000

# Rótulos para dados reais e falsos
valid = np.ones((batch_size, 1))
fake = np.zeros((batch_size, 1))

# Histórico de perdas
d_losses = []
g_losses = []

# Loop de treinamento
for epoch in range(epochs):
    # Treinar o discriminador

    # Selecionar um batch aleatório de imagens
    idx = np.random.randint(0, X_train.shape[0], batch_size)
    imgs = X_train[idx]
```

```
    # Gerar um batch de novas imagens
    noise = np.random.normal(0, 1, (batch_size, latent_dim))
    gen_imgs = generator.predict(noise)

    # Treinar o discriminador
    d_loss_real = discriminator.train_on_batch(imgs, valid)
    d_loss_fake = discriminator.train_on_batch(gen_imgs, fake)
    d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

    # Treinar o gerador
    noise = np.random.normal(0, 1, (batch_size, latent_dim))
    g_loss = combined.train_on_batch(noise, valid)

    # Registrar histórico
    d_losses.append(d_loss[0])
    g_losses.append(g_loss)

    # Imprimir progresso
    if epoch % 100 == 0:
        print(f"{epoch} [D loss: {d_loss[0]:.4f}, acc.: {100*d_loss[1]:.2f}%] [G loss: {g_loss:.4f}]")

    # Salvar imagens geradas
    if epoch % save_interval == 0:
        save_imgs(epoch, generator)

        # Salvar modelos a cada 5000 épocas
        if epoch % 5000 == 0 and epoch > 0:
            generator.save(f"gan_fashion/generator_epoch_{epoch}.h5")
            discriminator.save(f"gan_fashion/discriminator_epoch_{epoch}.h5")

# Plotar histórico de perdas
plt.figure(figsize=(10, 5))
plt.plot(d_losses, label='Discriminator')
plt.plot(g_losses, label='Generator')
plt.title('Loss over training')
plt.legend()
plt.savefig("gan_fashion/loss_history.png")
plt.show()

# Gerar e salvar um conjunto final de imagens
save_imgs(epochs, generator, examples=25, dim=(5, 5), figsize=(10, 10))

# Salvar os modelos finais
generator.save("gan_fashion/generator_final.h5")
discriminator.save("gan_fashion/discriminator_final.h5")

# Função para gerar imagens com o modelo salvo
def generate_images(generator_path, num_images=16, dim=(4, 4), figsize=(10, 10)):
    # Carregar o gerador
    loaded_generator = tf.keras.models.load_model(generator_path)

    # Gerar ruído aleatório
    noise = np.random.normal(0, 1, (num_images, latent_dim))
```

```
    # Gerar imagens
    gen_imgs = loaded_generator.predict(noise)

    # Rescale images 0 - 1
    gen_imgs = 0.5 * gen_imgs + 0.5

    # Plotar imagens
    plt.figure(figsize=figsize)
    for i in range(num_images):
        plt.subplot(dim[0], dim[1], i+1)
        plt.imshow(gen_imgs[i, :, :, 0], cmap='gray')
        plt.axis('off')
    plt.tight_layout()
    plt.savefig("gan_fashion/generated_samples.png")
    plt.show()

    return gen_imgs

# Exemplo de uso da função de geração
# generate_images("gan_fashion/generator_final.h5")

# Função para interpolação no espaço latente
def latent_space_interpolation(generator_path, num_steps=10):
    # Carregar o gerador
    loaded_generator = tf.keras.models.load_model(generator_path)

    # Gerar dois pontos aleatórios no espaço latente
    z1 = np.random.normal(0, 1, (1, latent_dim))
    z2 = np.random.normal(0, 1, (1, latent_dim))

    # Interpolar entre os dois pontos
    alphas = np.linspace(0, 1, num_steps)
    z_interp = np.zeros((num_steps, latent_dim))

    for i, alpha in enumerate(alphas):
        z_interp[i] = alpha * z1 + (1 - alpha) * z2

    # Gerar imagens
    gen_imgs = loaded_generator.predict(z_interp)

    # Rescale images 0 - 1
    gen_imgs = 0.5 * gen_imgs + 0.5

    # Plotar imagens
    plt.figure(figsize=(15, 3))
    for i in range(num_steps):
        plt.subplot(1, num_steps, i+1)
        plt.imshow(gen_imgs[i, :, :, 0], cmap='gray')
        plt.axis('off')
    plt.tight_layout()
    plt.savefig("gan_fashion/latent_space_interpolation.png")
    plt.show()
```

```
# Exemplo de uso da função de interpolação
# latent_space_interpolation("gan_fashion/generator_final.h5")
```

Extensões Possíveis

1. Implementar StyleGAN para maior controle sobre as características geradas
2. Adicionar condicionamento para gerar imagens de categorias específicas
3. Implementar CycleGAN para tradução de imagem para imagem
4. Criar uma interface web para geração interativa de imagens

Projeto 5: Assistente de Conversação com Transformers

Neste projeto, vamos criar um assistente de conversação simples usando modelos Transformer pré-treinados.

Objetivo

Desenvolver um chatbot capaz de manter conversas coerentes e responder a perguntas.

Implementação

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer
import gradio as gr
import os
import time
import numpy as np
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
import pandas as pd

# Verificar se GPU está disponível
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Usando dispositivo: {device}")

# Carregar modelo e tokenizador
# Usaremos um modelo menor para este exemplo
model_name = "microsoft/DialoGPT-small"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)
model.to(device)

# Função para gerar resposta
def generate_response(input_text, chat_history=None, max_length=100, temperature=0.7, top_p=0.9):
    # Inicializar histórico de chat se não existir
    if chat_history is None:
        chat_history = []

    # Adicionar entrada do usuário ao histórico
    chat_history.append({"role": "user", "content": input_text})

    # Preparar o contexto da conversa
    conversation = ""
    for message in chat_history:
        if message["role"] == "user":
```

```
            conversation += f"User: {message['content']}\n"
        else:
            conversation += f"Assistant: {message['content']}\n"

    # Tokenizar a entrada
    inputs = tokenizer.encode(conversation + "Assistant:", return_tensors="pt").to(device)

    # Gerar resposta
    with torch.no_grad():
        outputs = model.generate(
            inputs,
            max_length=inputs.shape[1] + max_length,
            temperature=temperature,
            top_p=top_p,
            pad_token_id=tokenizer.eos_token_id,
            do_sample=True
        )

    # Decodificar a resposta
    response = tokenizer.decode(outputs[0][inputs.shape[1]:], skip_special_tokens=True)

    # Adicionar resposta ao histórico
    chat_history.append({"role": "assistant", "content": response})

    return response, chat_history

# Função para interface Gradio
def chatbot_interface(message, history):
    # Converter histórico do Gradio para nosso formato
    chat_history = []
    for user_msg, bot_msg in history:
        chat_history.append({"role": "user", "content": user_msg})
        if bot_msg:  # Pode ser None no início
            chat_history.append({"role": "assistant", "content": bot_msg})

    # Gerar resposta
    response, _ = generate_response(message, chat_history)

    return response

# Criar interface Gradio
demo = gr.Interface(
    fn=chatbot_interface,
    inputs=gr.Textbox(lines=2, placeholder="Digite sua mensagem aqui..."),
    outputs="text",
    title="Assistente de Conversação com Transformers",
    description="Um chatbot simples baseado no modelo DialoGPT",
    examples=[
        ["Olá, como você está?"],
        ["Pode me contar uma piada?"],
        ["O que você sabe sobre inteligência artificial?"],
        ["Qual é a capital da França?"],
        ["Como funciona uma rede neural?"]
```

```
    ],
    allow_flagging="never"
)

# Função para analisar embeddings de diferentes perguntas
def analyze_embeddings():
    # Lista de perguntas para análise
    questions = [
        "Olá, como você está?",
        "Bom dia, tudo bem?",
        "Oi, tudo certo?",
        "Pode me contar uma piada?",
        "Conte-me algo engraçado",
        "Faça-me rir com uma piada",
        "O que você sabe sobre inteligência artificial?",
        "Me explique o que é IA",
        "Como funcionam os algoritmos de machine learning?",
        "Qual é a capital da França?",
        "Onde fica Paris?",
        "Quais são as principais cidades da Europa?",
        "Como funciona uma rede neural?",
        "Explique o funcionamento de redes neurais",
        "O que são camadas em deep learning?"
    ]

    # Obter embeddings
    embeddings = []
    for question in questions:
        inputs = tokenizer.encode(question, return_tensors="pt").to(device)
        with torch.no_grad():
            outputs = model(inputs)
            # Usar a média dos embeddings da última camada oculta
            embedding = outputs.past_key_values[0][0].mean(dim=1).cpu().numpy()
            embeddings.append(embedding.flatten())

    # Converter para array numpy
    embeddings = np.array(embeddings)

    # Reduzir dimensionalidade com t-SNE
    tsne = TSNE(n_components=2, random_state=42)
    embeddings_2d = tsne.fit_transform(embeddings)

    # Criar DataFrame para visualização
    df = pd.DataFrame({
        'x': embeddings_2d[:, 0],
        'y': embeddings_2d[:, 1],
        'question': questions
    })

    # Definir categorias para cores
    categories = [
        "saudação", "saudação", "saudação",
        "humor", "humor", "humor",
```

```
        "IA", "IA", "IA",
        "geografia", "geografia", "geografia",
        "redes neurais", "redes neurais", "redes neurais"
    ]
    df['category'] = categories

    # Plotar
    plt.figure(figsize=(12, 8))
    for category, group in df.groupby('category'):
        plt.scatter(group['x'], group['y'], label=category, alpha=0.8)

    # Adicionar rótulos
    for i, row in df.iterrows():
        plt.annotate(row['question'], (row['x'], row['y']), fontsize=8)

    plt.title('Visualização t-SNE dos Embeddings de Perguntas')
    plt.legend()
    plt.grid(True)
    plt.savefig('question_embeddings.png')
    plt.show()

    return df

# Função para avaliar a qualidade das respostas
def evaluate_responses():
    # Lista de perguntas para avaliação
    test_questions = [
        "Olá, como você está?",
        "Pode me contar uma piada?",
        "O que você sabe sobre inteligência artificial?",
        "Qual é a capital da França?",
        "Como funciona uma rede neural?"
    ]

    # Gerar e avaliar respostas
    results = []
    for question in test_questions:
        # Gerar resposta
        start_time = time.time()
        response, _ = generate_response(question)
        end_time = time.time()

        # Calcular tempo de resposta
        response_time = end_time - start_time

        # Calcular comprimento da resposta
        response_length = len(response.split())

        # Adicionar aos resultados
        results.append({
            'question': question,
            'response': response,
            'response_time': response_time,
```

```
            'response_length': response_length
        })

    # Criar DataFrame
    df = pd.DataFrame(results)

    # Imprimir resultados
    print("\nAvaliação de Respostas:")
    for i, row in df.iterrows():
        print(f"\nPergunta: {row['question']}")
        print(f"Resposta: {row['response']}")
        print(f"Tempo de resposta: {row['response_time']:.2f} segundos")
        print(f"Comprimento da resposta: {row['response_length']} palavras")

    # Plotar estatísticas
    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)
    plt.bar(df['question'], df['response_time'])
    plt.xlabel('Pergunta')
    plt.ylabel('Tempo de Resposta (s)')
    plt.title('Tempo de Resposta por Pergunta')
    plt.xticks(rotation=45, ha='right')
    plt.tight_layout()

    plt.subplot(1, 2, 2)
    plt.bar(df['question'], df['response_length'])
    plt.xlabel('Pergunta')
    plt.ylabel('Comprimento da Resposta (palavras)')
    plt.title('Comprimento da Resposta por Pergunta')
    plt.xticks(rotation=45, ha='right')
    plt.tight_layout()

    plt.savefig('response_evaluation.png')
    plt.show()

    return df

# Executar análise de embeddings
embedding_df = analyze_embeddings()

# Executar avaliação de respostas
evaluation_df = evaluate_responses()

# Lançar a interface Gradio
demo.launch()
```

Extensões Possíveis

1. Implementar recuperação de conhecimento para respostas mais informativas
2. Adicionar memória de longo prazo para manter o contexto da conversa
3. Implementar detecção de intenção para direcionar a conversa
4. Adicionar suporte a múltiplos idiomas

Conclusão

Neste tópico, exploramos cinco projetos práticos que demonstram a aplicação de redes neurais e deep learning em diferentes domínios:

1. **Sistema de Reconhecimento de Imagens**: Utilizando transfer learning com CNNs para classificação de imagens
2. **Análise de Sentimento em Textos**: Usando BERT para classificar o tom emocional de textos
3. **Previsão de Séries Temporais**: Aplicando LSTMs para prever valores futuros com base em dados históricos
4. **Geração de Imagens com GANs**: Criando imagens realistas usando redes adversariais generativas
5. **Assistente de Conversação com Transformers**: Desenvolvendo um chatbot baseado em modelos de linguagem

Estes projetos fornecem uma base sólida para aplicar os conceitos teóricos que aprendemos nos tópicos anteriores. Você pode adaptar e expandir esses projetos para suas próprias necessidades, adicionando funcionalidades, melhorando o desempenho ou aplicando-os a diferentes conjuntos de dados.

Lembre-se de que o aprendizado em deep learning é um processo contínuo. À medida que você ganha experiência com esses projetos, você desenvolverá intuições sobre como ajustar hiperparâmetros, escolher arquiteturas apropriadas e resolver problemas comuns.

Recomendamos que você experimente com diferentes configurações, conjuntos de dados e técnicas para aprofundar seu entendimento e desenvolver suas habilidades práticas em deep learning.

Referências

1. Python.org - Documentação oficial: https://docs.python.org/3/
2. Real Python - Tutoriais e artigos: https://realpython.com/
3. NumPy - Documentação: https://numpy.org/doc/
4. Pandas - Documentação: https://pandas.pydata.org/docs/
5. Matplotlib - Documentação: https://matplotlib.org/stable/contents.html
6. Scikit-learn - Documentação: https://scikit-learn.org/stable/documentation.html
7. TensorFlow - Documentação: https://www.tensorflow.org/api_docs
8. PyTorch - Documentação: https://pytorch.org/docs/stable/index.html
9. Keras - Documentação: https://keras.io/api/
10. Deep Learning Book (Goodfellow, Bengio, Courville): https://www.deeplearningbook.org/