escola britânica de artes criativas & tecnologia

Módulo | Análise de Dados: Fundamentos de Matemática

Caderno de **Aula**
Professor André Perez

Tópicos

1. Vetorização;
2. Arrays Numpy;
3. Operações.

Aulas

0. Abordagens estatísticas

- **Descritiva**: foco no passado para entender o **presente**.
- **Preditiva**: foca no passado para inferir o **futuro**.

1. Vetorização

1.1. Introdução

Derivado da matemática computacional, **vetorização** é o processo de transformar um código escalar em sua forma vetorial. Na prática, geralmente eliminamos operações matemáticas que lidam com um conjunto reduzido de números por laço de repetição (escalar) por uma única operação com todos os números (vetor).

> A **vetorização** (geralmente) é mais **rápida** e utiliza menos **memória**

Exemplo:

Uma *fintech* quer saber a qualidade dos investimentos dos seus clientes. Para tanto, é proposta a seguinte análise: para um determinado mês, deve-se comparar o retorno de cada cliente obtido através do investimento com o valor corrigido pela inflação. Deve-se então calcular quantos clientes "perderam" para inflação.

- Para 1 cliente:

In [ ]:
```
montante = 1000
montante_final = 1001.5 # ~ 0.15% de retorno
```

In [ ]:
```
ipca = 0.25 / 100
montate_inflacao = montante * (1 + ipca)

print(montate_inflacao)
```

In [ ]:
```
print(montante_final < montate_inflacao)
```

- Para 100 mil clientes:

In [ ]:
```
from random import random, randint

montante_lista = [randint(0, 5000) for _ in range(0, 100000)]
montante_final_lista = [round(montante * (1 + (0.3 * random() / 100)), 2)
                        for montante in montante_lista]
```

In [ ]:
```
print(montante_lista[0:5])
print(montante_final_lista[0:5])
```

O laço repetição *for/in* opera de forma **escalar**, ou seja, processa um conjunto limitado de números por repetição.

In [ ]:
```
from time import time

perdeu = list()

# calculo
inicio = time() # inicio da contagem do tempo
for montante, montante_final in zip(montante_lista, montante_final_lista):
  perdeu.append(montante_final - montante * (1 + ipca))
fim = time() # fim da contagem do tempo

# resultado
perdeu = list(filter(lambda val: True if val < 0 else False, perdeu))
print(f"{len(perdeu)} cliente perderam para inflação")

# tempo
tempo_lista = fim - inicio
print(fim - inicio)
```

1.2. Pacote NumPy

Pacote Python para manipulação numérica construido na linguagem de programação C, muitos pacotes o utilizam como base, como o Pandas e SciPy. A documentação pode ser encontrada neste link. A abstração base do NumPy é o *array*, uma estrutura de dados Python de duas ou mais dimensões utilizado para representar vetores, matrizes, tensores, etc.

Vamos vetorizar nosso cálculo através da criação de vetores NumPy.

In [ ]:
```
import numpy as np

montante_array = np.array(montante_lista)
montante_final_array = np.array(montante_final_lista)
```

In [ ]:
```
print(montante_array)
print(montante_final_array)
```

Vamos então fazer o mesmo cálculo com os vetores NumPy de forma **vetorial**, ou seja, processando todos os números de uma vez.

In [ ]:
```
from time import time

# calculo
inicio = time() # inicio da contagem do tempo
perdeu = montante_final_array - montante_array * (1 + ipca)
fim = time() # fim da contagem do tempo

# resultado
perdeu = list(filter(lambda val: True if val < 0 else False, perdeu))
print(f"{len(perdeu)} cliente perderam para inflação")

# tempo
tempo_array = fim - inicio
print(f"Duração: {tempo_array}")
```

Por fim, vamos comparar os tempos de execução das operações escalares e vetoriais.

In [ ]:
```
tempo_lista / tempo_array
```

2. Arrays Numpy

2.1. Arrays vs Listas

As listas Python e os arrays NumPy são similares em muitos sentidos: ambos servem param armazenar dados sequencialmente na memória, possuem sintaxe parecidas, etc. Contudo, é importante maximizar as suas diferenças:

- **Manipulação algébrica**: Arrays apresentam uma sintaxe mais simples e eficiente (velocidade e memória) por realizar operaçõs de forma **vetorial**. Listas sempre trabalham de forma **escalar**. Exemplo:

In [ ]:
```
# escalar

l1 = [1, 2, 3]
l2 = [4, 5, 6]

l3 = [a + b for a, b in zip(l1, l2)]
print(l3)
```

In [ ]:
```
# vetorial

a1 = np.array(l1)
a2 = np.array(l2)

a3 = a1 + a2
print(a3)
```

- **Tipo**: Arrays trabalham melhor com elementos do mesmo tipo.

- **Mutabilidade**: Arrays são menos eficientes quanto a inserção e remoção de elementes.

2.2. Arrays 1D: Vetores

Arrays NumPy de uma dimensão (1D) são conhecidos como **vetores**, como listas, de uma linha e uma ou mais colunas.

- **Criação**

In [ ]:
```
a1 = np.array([2, 4, 6, 8])
print(a1)
```

In [ ]:
```
a1 = np.arange(0, 10, 2)
print(a1)
```

In [ ]:
```
a1 = np.zeros(10)
print(a1)

a1 = 10 * np.ones(10)
print(a1)
```

- **Manipulação**

In [ ]:
```
a1 = np.array([2, 4, 6, 8])
print(a1)
```

In [ ]:
```
a1[0]
```

In [ ]:
```
a1[0:2]
```

In [ ]:
```
a1[a1 > 4]
```

- **Atributos**

In [ ]:
```
a1.ndim
```

In [ ]:
```
a1.shape
```

In [ ]:
```
a1.size
```

In [ ]:
```
a1.dtype
```

- **Métodos**

In [ ]:
```
a1.sort() # "inplace"
print(a1)
```

In [ ]:
```
a1.tolist()
```

2.3. Arrays 2D: Matrizes

Arrays NumPy de duas dimensão (2D) são conhecidos como **matrizes**, como tabelas, com linhas e colunas.

- **Criação**

In [ ]:
```
m1 = np.array([[1, 2, 3], [4, 5, 6]]) # vetores como linhas
print(m1)
```

- **Manipulação**

In [ ]:
```
m1[1,2] # linha x coluna
```

In [ ]:
```
m1[1,:]
```

In [ ]:
```
m1[:,1]
```

- **Atributos**

In [ ]:
```
m1.ndim
```

In [ ]:
```
m1.shape
```

In [ ]:
```
m1.size
```

In [ ]:
```
m1.dtype
```

- **Métodos**

In [ ]:
```
m1.sort() # "inplace"
print(m1)
```

In [ ]:
```
m1.tolist()
```

2.4. Arrays 3D, 4D, etc.

Arrays NumPy de três ou mais dimensões são apenas estruturas de dados com mais de duas dimensões.

3. Operações

O NumPy oferece uma grande quantidade de operações matemáticas, potencialmente vetoriais, além de constantes para auxiliar os cálculos. Também oferece suporte para operações mais avançadas, como álgebra vetorial. Uma lista completa das operações disponíveis está disponível neste link. Vamos utilizar os seguintes arrays para exemplicar as operações:

In [ ]:
```
a1 = np.array([1, 2, 3])
a2 = np.array([3, 4, 5])
```

3.1. Constantes

- **Numérico**

In [ ]:
```
np.pi # famoso pi, muito usado em trigonometria
```

In [ ]:
```
np.e # euler, muito usado em logaritmos
```

- **Nulo**

In [ ]:
```
np.nan, np.NaN, np.NAN
```

In [ ]:
```
type(np.nan)
```

- **Infinito**

In [ ]:
```
np.inf, np.Inf, np.Infinity
```

In [ ]:
```
type(np.inf)
```

3.2. Funções Elementares

- **Soma**

In [ ]:
```
a3 = a1 + a2
print(a3)

a3 = a1 - a2
print(a3)
```

- **Multiplicação**

In [ ]:
```
a3 = a1 * a2
print(a3)

a3 = a1 / a2
print(a3)
```

- **Exponenciação**

In [ ]:
```
a3 = a1 ** 2
print(a3)

a3 = np.sqrt(a1)
print(a3)
```

- **Logaritmo**

In [ ]:
```
a3 = np.log(a1)
print(a3)

a3 = np.exp(a1)
print(a3)
```

- **Trigonometria**

In [ ]:
```
a3 = np.sin(a1)
print(a3)

a3 = np.tan(a1)
print(a3)
```

3.3. Álgebra Vetorial

- **Produto escalar**

O produto escalar representa a projeção de um vetor em outro. Portanto, é uma operação que leva dois vetores em um escalar. É usando em diversas aplicações da física e engenharia, como o posicionamento de placas solares.

> $\bar{\textbf{x}}, \bar{\textbf{y}} \to z$

Sendo que:

> $\bar{\textbf{x}}, \bar{\textbf{y}} \in \Re^{i}$
>
> $z \in \Re$

É calculado pela soma dos produtos dos elementos dos vetores.

> $\bar{\textbf{x}} \cdot \bar{\textbf{y}} = \sum_{i}^{j} x_iy_i$

In [ ]:

```
a1 = np.array([1, 2, 3])
a2 = np.array([3, 4, 5])

a3 = np.dot(a1, a2)
print(a3)
```