



Aprenda com quem faz

Bibliotecas para Análise de Dados

Prof. William Sdayle

2023



SUMÁRIO

Capítulo 1. Introdução a Python	5
Instalação Anaconda	6
Instalação Virtualenv	7
Google Colab	8
Variáveis	8
Condições	10
Laços de repetição.....	13
Funções	15
Capítulo 2. Numpy.....	18
Array	18
Criação de Arrays.....	20
Indexação de Arrays.....	23
Funções úteis.....	24
Operações com arrays.....	25
Capítulo 3. Introdução ao Pandas	28
O que é Pandas?	28
Séries e DataFrames.....	28
Leitura de Arquivos	30
Manipulação de dataframes.....	30
Capítulo 4. Pandas	34
Operações com DataFrames	34
DataFrame Merge e Concat.....	37
Agrupamentos e Agregações	41
Capítulo 5. Mineração de dados.....	44
Cenários de mineração de dados.....	44

Ciência de Dados e Mineração	46
Capítulo 6. Matplotlib	49
O que é Matplotlib?	49
Tipos de gráficos e suas funcionalidades.....	54
Capítulo 7. Scipy	60
O que é Scipy?	60
Álgebra Linear com Scipy	61
Capítulo 8. Scikit Learn.....	64
O que é Scikit-Learn?	64
Avaliação e Seleção de modelos com Scikit-Learn	66
Referências	70



XPe

> Capítulo 1



Capítulo 1. Introdução a Python

Para dar início a esta apostila, serão tratados assuntos relacionados ao uso da linguagem Python, tais como: instalação, ambientes de desenvolvimento, variáveis, condições, laços de repetição e funções. Com a finalidade de que não haja conflitos em instalações de dependências, será apresentado como construir os ambientes virtuais e como ativá-los utilizando o sistema operacional Linux. Também será apresentado como utilizar a linguagem através do sistema Colab do Google, tornando seu uso mais ágil e mais simples, independente do sistema operacional utilizado.

Visando manter uma linha de raciocínio entre as funções apresentadas, utilizaremos o dataset *Breast Cancer Wisconsin* presente neste [link](#), ou seja, todos os dados apresentados nesta apostila são oriundos deste dataset. Ao fim dos capítulos, haverá um *notebook* com funções e exemplos apresentados em cada assunto.

Python é uma das linguagens mais procuradas nos dias atuais, principalmente quando se trata de processamento de dados. Devido ao grande suporte de entusiastas da linguagem e alto poder científico das bibliotecas, Python é talvez a linguagem em maior uso para processamento de dados complexos.

A linguagem Python pode ser utilizada para processamento de imagens digitais¹, para análise de dados com machine learning², análise

¹ SHREYA, Debbata. DIGITAL IMAGE PROCESSING AND RECOGNITION USING PYTHON. In: International Journal of Engineering Applied Sciences and Technology, v. 5, 2021.

² JUPUDI, Lakshmi. Machine learning techniques using python for data analysis in performance evaluation. In: International Journal of Intelligent Systems Technologies and Applications, v. 17, n. 3, 2018.

temporal de valores³, entre outros. Um dos fatores que aumenta seu uso é a facilidade de compreensão, em que é mais objetiva, tornando seu uso mais atrativo entre os alunos.

Para começar o uso da linguagem, apresentaremos duas maneiras de construir os ambientes virtuais, através do `virtualenv` e do `anaconda`.

Instalação Anaconda

Abra um terminal e entre na pasta `/tmp`. Para instalar o `anaconda`, utilize o método `curl` com o link apresentado na Figura 1.

Figura 1 - Download do anaconda.

```
(base) wiusdy@wiusdy:/tmp$ curl https://repo.anaconda.com/archive/Anaconda3-2020.02-Linux-x86_64.sh --output anaconda.sh
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total   Spent    Left   Speed
100 521M  100 521M    0     0  10.8M      0  0:00:48  0:00:48 --:--:-- 11.1M
```

Para verificar se o download foi realizado com sucesso, utilize o comando `sha256sum anaconda.sh`, caso o resultado seja parecido com a Figura 2, a instalação foi realizada com êxito.

Figura 2 - Validação do download.

```
(base) wiusdy@wiusdy:/tmp$ sha256sum anaconda.sh
2b9f088b2022edb474915d9f69a803d6449d5fdb4c303041f60ac4aefcc208bb  anaconda.sh
(base) wiusdy@wiusdy:/tmp$
```

Para começar a instalação, digite `bash anaconda.sh` e aperte enter até chegar ao fim da descrição da ferramenta. Ao fim da descrição, será

³ MAULUD, Dastan; ABDULAZEEZ, Adnan Mohsin. A Review on Linear Regression Comprehensive in Machine Learning. In: Journal of Applied Science and Technology Trends, v. 1, 2020. p. 140-147.

solicitado a entrada do usuário de *yes* ou *no* para instalar a ferramenta, conforme na Figura 3.

Figura 3 - Instalação da ferramenta.

```
Do you accept the license terms? [yes|no]
[no] >>>
Please answer 'yes' or 'no':
>>>
Please answer 'yes' or 'no':
>>> yes

Anaconda3 will now be installed into this location:
/home/wiusdy/anaconda3

- Press ENTER to confirm the location
- Press CTRL-C to abort the installation
- Or specify a different location below

[/home/wiusdy/anaconda3] >>> /home/wiusdy/conda/
PREFIX=/home/wiusdy/conda
Unpacking payload ...
Extracting : py-1.8.1-py_0.conda: 16%|
```

Para criar o ambiente virtual com o anaconda, utilize o comando *conda create -n nome-ambiente-virtual*, em que *nome-ambiente-virtual* é o nome escolhido para o ambiente. Para ativá-lo, utilize o comando *conda activate nome-ambiente-virtual*.

Instalação Virtualenv

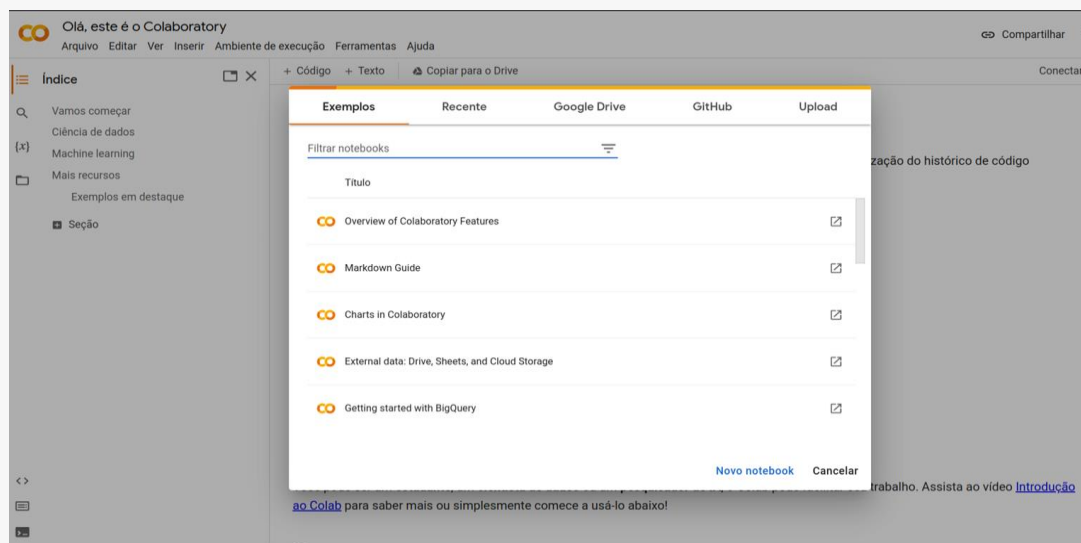
Para instalar o Python utilizando o Linux, digite o comando *sudo apt-get install python3*. Após instalar o Python, instale o comando *pip* para realizar os downloads e instalações das bibliotecas necessárias. Para instalar o *pip*, utilize o comando *sudo apt-get install python3-pip*. Após a instalação do *pip*, pode ser instalado o virtualenv através do comando *sudo pip3 install virtualenv*.

Para criar um ambiente virtual, utilize o comando *python3 -m venv nome-ambiente-virtual*. Para ativar o ambiente virtual, utilize o comando *source nome-ambiente-virtual/bin/activate*.

Google Colab

Para utilizar o Colab, é necessário ter uma conta Gmail no Google. Para acessá-lo, use este [link](#). A fim de validar que se está no Colab, verifique se a página inicial é similar à Figura 4.

Figura 4 - Página inicial do Google Colab.



O Colab é muito interessante de ser utilizado devido ao fato de não ser necessário instalar nada dentro da máquina, sendo mais simples a execução independente do sistema operacional em uso.

Variáveis

Variável, em programação, é o nome dado a um valor ou expressão em Python, usamos o sinal de igual (=) para atribuir um valor a uma variável. Cabe ao programador escolher nomes com base na necessidade da variável, com nomes autoexplicativos. Cada variável armazenará um determinado valor em memória e sempre que essa variável for acessada retornará seu valor armazenado.

As variáveis em Python são de tipagem fraca, ou seja, não é necessário definir qual o tipo da variável antes de declará-la, o próprio interpretador do Python identifica qual o tipo da variável e prepara a

estrutura de armazenamento necessária. Os tipos de variáveis em Python estão descritos na tabela 1.

Tabela 1 - Tipos de variáveis e suas definições.

Tipo de variável	Definição
Inteira	<code>a = 20</code>
Ponto flutuante	<code>c = 1.5</code>
Caractere	<code>letra = 'b'</code>
Frase	<code>nome = "William"</code>
Binário	<code>esta_chovendo = False</code>
Lista	<code>nomes = ["Carlos", "João"]</code>
Dicionário	<code>chamada = {"nome": "João"}</code>
Tupla	<code>valores = (20, 30.5, "João", False)</code>

Para definir as variáveis, é necessário se atentar a algumas regras, tais como: não começar com valores inteiros, não haver espaços entre palavras (e.g. `esta_chovendo`) e as variáveis precisam ter nomes os mais descritivos possíveis.

Uma variável do tipo inteira auxilia na manipulação de informações inteiras, como: idade, dia, mês, ano etc. Variáveis de ponto flutuante auxiliam na manipulação de quaisquer informações numéricas, tais como: nota, preço, distância, entre outros. Caracteres são pouco utilizados em Python já que uma frase (*string*) pode ser utilizada com maior frequência. Um caractere se atribui utilizando aspas simples (") já uma *string* se atribui com aspas duplas ("). Uma *string* nada mais é do que uma lista de caracteres. Variáveis binárias (*boolean*) geralmente são utilizadas em condições ao longo do código, principalmente para realizar verificações.

Listas em Python são de grande uso, já que dentro delas podemos adicionar quaisquer tipos de variáveis. Cada valor é armazenado na memória

apontando para a lista que foi criada, ou seja, uma lista pode armazenar n valores e cada valor pode ser acessado pelo próprio valor ou por seu id de posição.

Dicionários são variáveis que auxiliam a atribuir chaves a determinados valores. A estrutura em dicionário funciona de maneira similar a um dicionário de palavras, onde se busca através de uma chave um determinado valor que é a definição da palavra. Portanto, dicionários funcionarão com relacionamento entre chave e valor.

Tuplas se diferem de listas e dicionários pelo fato de que os valores dentro de uma tupla não podem ser alterados, ou seja, uma vez que um determinado valor foi adicionado a uma tupla, tal valor nunca poderá ser alterado. Isso é muito importante para verificação dos dados dentro de um pipeline de processamento.

Condições

Em Python, condições ou estruturas condicionais auxiliam na verificação de informações ao longo do funcionamento do algoritmo. Tal estrutura é muito útil para validar resultados, verificar dados e manipular o fluxo de funcionamento do código. As estruturas condicionais funcionam da seguinte maneira: se condição satisfeita, realizar procedimento. As três estruturas são if, elif e else.

Para entender o funcionamento das condições e como elas podem ser verificadas, é necessário compreender os operadores numéricos e os operadores lógicos. Operadores numéricos são funções matemáticas simples, adição (+), subtração (-), divisão (/), multiplicação (*), potencialização (**) e resto da divisão (%). Os operadores lógicos são os mais utilizados em verificações em algoritmos, conforme apresentado na tabela 2.

Tabela 2 - Operadores e suas funcionalidades.

Operador	Funcionalidade
>	<u>maior que</u>
<	<u>menor que</u>
>=	<u>maior ou igual que</u>
<=	<u>menor ou igual que</u>
==	<u>idêntico a</u>
!=	<u>diferente de</u>
Not	<u>Não</u>
And	<u>E</u>
Or	<u>Ou</u>

Para entender o funcionamento da estrutura de condições, podemos utilizar o exemplo de notas de um aluno, se a nota for acima de 7 ele é “Aprovado”, se a nota for abaixo de 7 e maior que 4 ele estará para “Recuperação” e se caso a nota for menor que 4 ele estará “Reprovado”. As figuras 5, 6 e 7 apresentam as três possíveis condições para os alunos.

Figura 5 - Aluno Aprovado.

```
✓ [1] nota = 7.5
0s  if nota > 7:
    print("Aprovado")
    elif nota < 7 and nota >= 4:
    print("Recuperação")
    else:
    print("Reprovado")
```

Aprovado

Figura 6 - Aluno de recuperação.

```
✓ [2] nota = 6.9
0s   if nota > 7:
      print("Aprovado")
      elif nota < 7 and nota >= 4:
          print("Recuperação")
      else:
          print("Reprovado")
```

Recuperação

Figura 7 - Aluno reprovado.

```
▶ nota = 3.9
   if nota > 7:
       print("Aprovado")
   elif nota < 7 and nota >= 4:
       print("Recuperação")
   else:
       print("Reprovado")
```

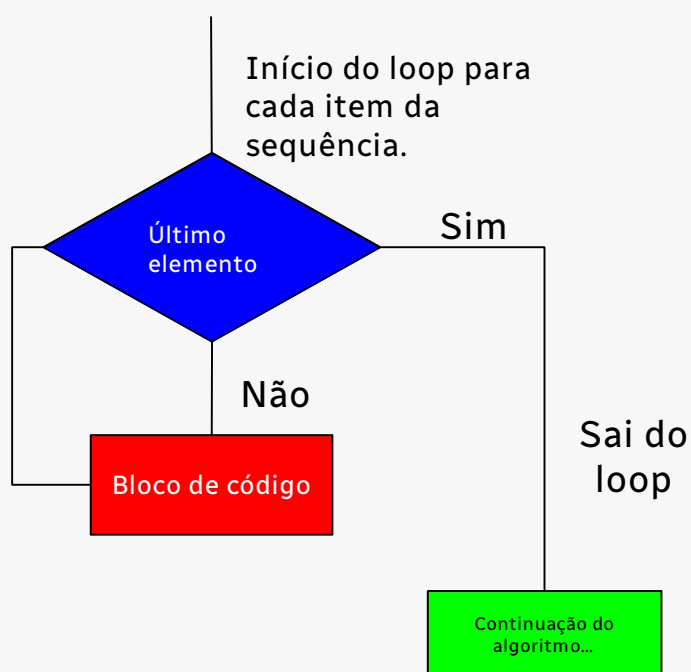
Reprovado

Nota-se que nas figuras 5, 6 e 7 a variável a ser analisada é a nota e com base nela fazemos as verificações se o aluno está aprovado ou não com base nas condições necessárias. Utilizamos if, elif e else para mostrar como funciona cada uma das estruturas, na figura 7 podemos perceber que foram criadas três condições para o aluno ser aprovado ou de recuperação e caso nenhuma dessas condições seja satisfeita assumimos que o aluno está reprovado. O fluxo do código é verificar primeiro o if, se a condição no if for satisfeita, ele fará o bloco de código dentro da condição que se dá através da indentação do código, caso o if não seja satisfeito, o fluxo pode ir para o elif que verificará também se o aluno está de recuperação ou não, em caso de não satisfação do if e nem do elif, o algoritmo executará o bloco de código que está em else.

Laços de repetição

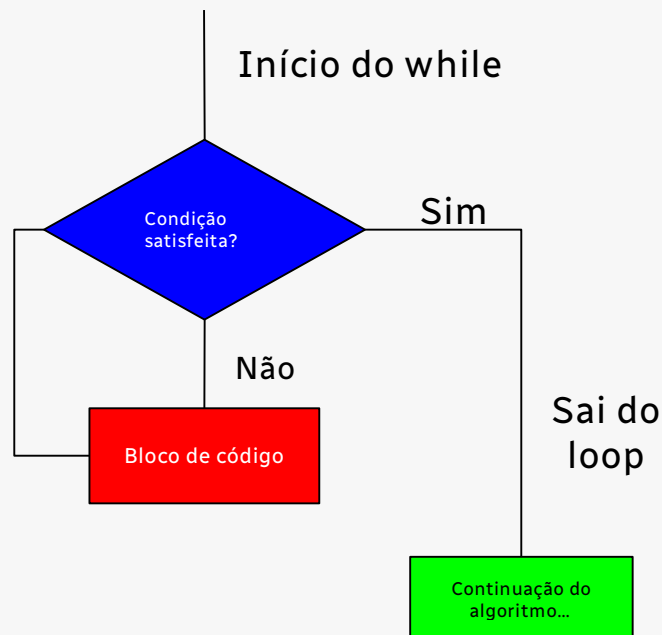
Laços de repetição são estruturas extremamente importantes para análise de dados, visto que várias ações podem ser tomadas de acordo com cada dado analisado. Um laço de repetição repetirá uma determinada ação n vezes. Em Python, os dois laços de repetição utilizados são `for` e `while`. A estrutura `for` é utilizada para operar qualquer tipo de sequência, seja uma string ou uma lista, as operações serão realizadas na ordem em que a sequência se apresenta. A estrutura `while` é usada quando precisamos repetir uma ação algumas vezes ou fazer uma iteração até que uma condição seja satisfeita. A estrutura `for` é apresentada na Figura 8.

Figura 8 - Funcionamento da estrutura de repetição `for`.



Na figura 9, é apresentado o funcionamento da estrutura de repetição `while`.

Figura 9 - Funcionamento da estrutura de repetição while.



Portanto, podemos afirmar que laços de repetição são muito úteis em tarefas repetitivas, e isso se aplica para processamento de variáveis sequenciais, tais como: listas, dicionários e tuplas. A estrutura for funcionará como apresentado na figura 10, e na figura 11 é apresentada a estrutura while.

Figura 10 - Funcionamento da estrutura for.

for

```
✓ [1] nomes = ["William", "Carlos", "Pedro", "Kevin", "Stewart"]  
0s for nome in nomes: # para cada nome na lista de nomes  
    print(nome)
```

```
William  
Carlos  
Pedro  
Kevin  
Stewart
```

Figura 11 - Funcionamento da estrutura while.

▼ While

✓
0s

▶

```
i = 0
while i < 10:
    print(i)
    i = i + 1 # a cada loop a variável i aumentará para seu valor atual mais 1
```

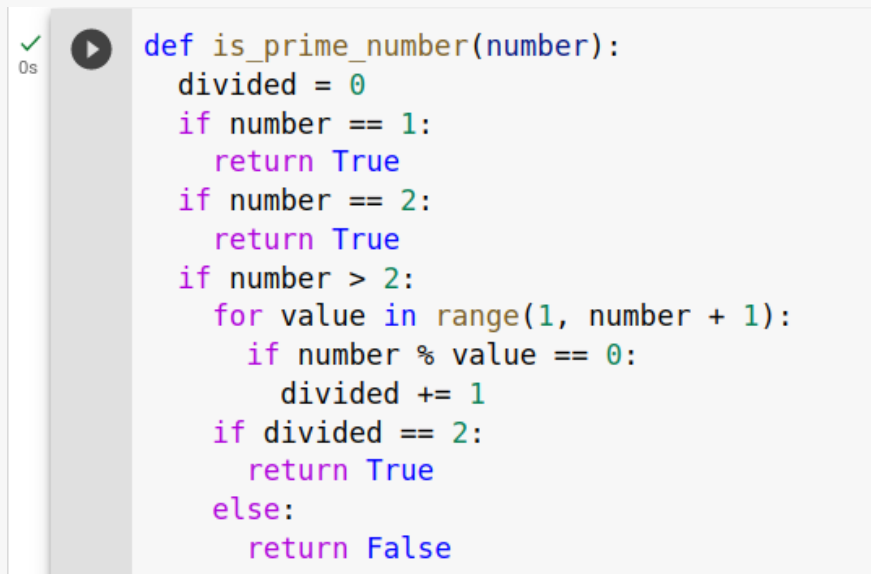
0
1
2
3
4
5
6
7
8
9

Funções

Em condições em que o algoritmo desenvolvido tenha um alto volume de linhas de código, ou que dentro do algoritmo a mesma operação seja realizada em locais diferentes, as funções podem auxiliar na melhora da leitura do código e no auxílio da manutenção de possíveis mudanças e solução de problemas.

Diferentemente das variáveis, as funções são definidas pela palavra-chave `def`, a partir dela se define uma função e seu nome pode ser o mais descritivo possível. As funções podem tomar como entrada alguns parâmetros, por exemplo uma função que soma dois valores pode ter como parâmetro os dois valores a serem somados. Um exemplo de função é apresentado na figura 12, neste exemplo é identificado se um determinado valor de parâmetro é ou não um número primo. Sabe-se que um número primo é um número que só é divisível por 1 e por ele mesmo.

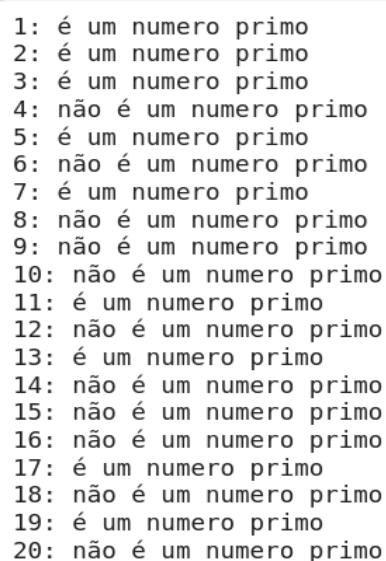
Figura 12 - Exemplo de função para verificar se um determinado número é ou não é primo.

A screenshot of a code editor interface. On the left, there is a vertical toolbar with a green checkmark icon and a play button icon. Below the play button, the text '0s' is visible. The main area of the editor contains a Python function definition for 'is_prime_number'. The code is as follows:

```
def is_prime_number(number):  
    divided = 0  
    if number == 1:  
        return True  
    if number == 2:  
        return True  
    if number > 2:  
        for value in range(1, number + 1):  
            if number % value == 0:  
                divided += 1  
        if divided == 2:  
            return True  
        else:  
            return False
```

No exemplo da figura 12, a lógica foi de percorrer todos os números de 1 até *number* e verificar se apenas duas vezes o resto da divisão entre *number* e *value* foi igual a zero, caso sim, *number* é primo, caso contrário, não. O resultado da função para os 20 primeiros números é apresentado na figura 13.

Figura 13 - Resultados da função *is_prime_number*.

A screenshot of a terminal window displaying the output of the 'is_prime_number' function for the first 20 natural numbers. The output is as follows:

```
1: é um numero primo  
2: é um numero primo  
3: é um numero primo  
4: não é um numero primo  
5: é um numero primo  
6: não é um numero primo  
7: é um numero primo  
8: não é um numero primo  
9: não é um numero primo  
10: não é um numero primo  
11: é um numero primo  
12: não é um numero primo  
13: é um numero primo  
14: não é um numero primo  
15: não é um numero primo  
16: não é um numero primo  
17: é um numero primo  
18: não é um numero primo  
19: é um numero primo  
20: não é um numero primo
```




XPe

> Capítulo 2



Capítulo 2. Numpy

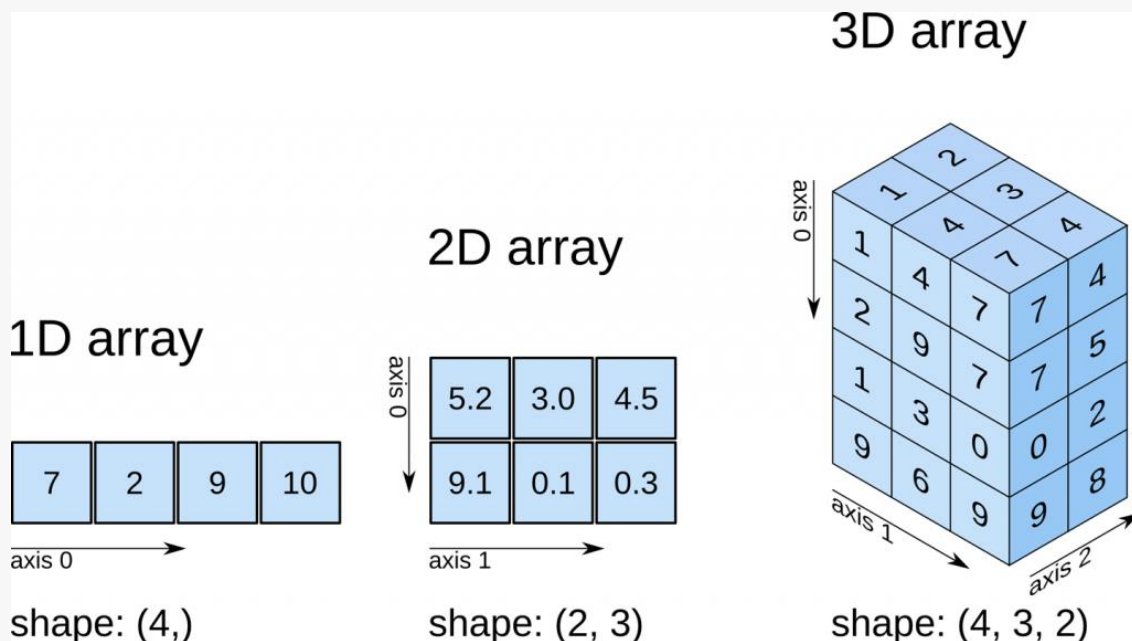
Essa poderosa biblioteca do Python é usada extensivamente por outras bibliotecas populares, como o Pandas, Scikit-Learn, Matplotlib e em grande parte de bibliotecas voltadas para a área de ciência de dados. Numpy é uma biblioteca criada baseada nos projetos Numeric e Numarray com o objetivo de centralizar a comunidade científica em uma biblioteca para processamento de arrays.

Devido sua construção e funcionalidades serem baseadas na estrutura de dados *ndarray*, a biblioteca oferece operações rápidas para tratamento e limpeza de dados, geração de subconjuntos e filtrações, estatísticas descritivas, manipulação de dados relacionais, manipulação de dados em grupos, entre outros tipos de processamento. Portanto, podemos afirmar que a biblioteca Numpy é de grande importância nas tarefas de ciência de dados.

Array

A principal estrutura da biblioteca Numpy é o array, um array é uma estrutura multidimensional que nos permite armazenar dados na memória do nosso computador, de modo que cada item localizado nessa estrutura pode ser encontrado por meio de um esquema de indexação. O NumPy Python denomina essa estrutura como *ndarray*, como forma de abreviação a array N-dimensional. A figura 14 ilustra as possíveis estruturas de arrays, unidimensionais, bidimensionais e tridimensionais.

Figura 14 - Diferentes dimensões para os Arrays.



No array 1D podemos notar que a estrutura é similar a uma lista sequencial, contendo apenas um eixo (*axis 0*) com 4 células. No array 2D podemos perceber que temos algo similar a uma matriz, com eixos x (*axis 1*) e eixo y (*axis 0*), portanto, para acessar uma determinada célula desta matriz, é necessário saber o valor do eixo x e do eixo y . No array 3D temos algo mais parecido a um cubo, adicionando um eixo z a estrutura similar a matriz, como uma espécie de profundidade. Arrays 3D são muito utilizados em bibliotecas de processamento de imagens, pois sabe-se que uma imagem RGB é processada computacionalmente através desta estrutura.

Existem várias vantagens no uso da biblioteca Numpy, já que são bibliotecas que possuem uma comunidade de desenvolvedores muito ampla e funções muito interessantes para uso em ciência de dados, além de que seus processos ocupam menos memória, são mais rápidos e possui alta facilidade no processamento de cálculos numéricos.

Criação de Arrays

De acordo com a documentação da biblioteca Numpy, podemos criar arrays utilizando 6 mecanismos principais: convertendo estruturas Python (e.g. listas e tuplas); criação intrínseca utilizando as funções Numpy (e.g. `arange`, `ones`, `zeros` etc.); replicando, concatenando ou manipulando arrays existentes; realizando leituras do disco; utilizando strings ou buffers para criação de arrays em bytes; utilizando bibliotecas especiais (e.g. `random`).

O primeiro passo é importar a biblioteca já que ela não é *built-in* da linguagem Python. A forma mais tradicional de se importar a biblioteca é utilizando o comando `"import numpy as np"`. Neste comando, utilizamos a palavra guardada de Python *as* para atribuir o apelido *np* à biblioteca *numpy*, sendo assim, ao longo de todo algoritmo, para utilizar qualquer função da biblioteca, utilizamos o apelido *np*.

Para criar um array a partir de uma lista Python, podemos utilizar o comando:

```
lista = [10, 20, 30, 40]
array = np.array(lista)
```

Nos códigos acima, criamos um array unidimensional utilizando a variável `lista`. Podemos criar um array bidimensional e tridimensional utilizando a mesma estratégia da lista Python.

```
lista2D = [[10, 20, 30], [30, 40, 50]]
array2D = np.array(lista2D)

lista3D = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]
array3D = np.array(lista3D)
```

Para analisar as dimensões de cada array, podemos utilizar a função *shape*, com ela teremos acesso a linhas, colunas e profundidade.

Conseguimos criar *arrays* com tamanhos e valores específicos, a fim de realizar isso utilizamos as funções *np.ones* para criação de arrays somente com valores iguais a 1 e *np.zeros* para criação de arrays somente com valores iguais a zero. Somos capazes também de utilizar a função *np.asarray* para converter em arrays *n*-dimensionais.

Para manipular cada array, existem várias funções, podemos utilizar algumas mais simples como: *copy*, *hstack*, *vstack*, *block*. E para salvar os dados de um array Numpy, podemos utilizar uma simples função *np.save* e para carregar arquivos *.npy* utilizamos a função *np.load*. A fim de facilitar a compreensão das funções e centralizar as mesmas, adicionamos a tabela 3.

Tabela 3 - Funções para criação de arrays.

Função	Descrição
array	Converte a entrada para um <i>ndarray</i> , podendo especificar através de dtype qual o tipo da entrada. Realiza a cópia do dado de entrada por padrão.
asarray	Converte o dado de entrada para um <i>ndarray</i> , mas não realiza a cópia do dado caso já seja um <i>ndarray</i> .
arange	Cria uma lista de números a partir de um número de início e um de fim.
ones	Cria um array contendo apenas valores iguais a um, podendo ser atribuído o tamanho do array e seu tipo de dado.
ones_like	Toma um array como entrada e produz um de saída com as mesmas dimensões, porém, com apenas valores iguais a um.

zeros	Cria um array contendo apenas valores iguais a zero, podendo ser atribuído o tamanho do array e seu tipo de dado.
zeros_like	Toma um array como entrada e produz um de saída com as mesmas dimensões, porém, com apenas valores iguais a zero.
empty	Cria um novo array alocando nova memória com valores aleatórios.
empty_like	Aloca nova memória utilizando um array como entrada. Similar ao zeros_like e ones_like, porém, sem adicionar nenhum valor igual.
full	Cria um array com as dimensões adicionadas e também com o valor dado de entrada.
full_like	Toma um array como entrada e produz um de saída com as mesmas dimensões, porém, com apenas valores iguais ao de entrada.
eye	Cria uma matriz identidade nxn .
identity	Cria uma matriz identidade nxn .

Indexação de Arrays

Assim como as listas, podemos acessar informações dos arrays através dos índices dos elementos. Cada elemento do array possui um índice, e para acessar tal elemento, podemos indicar um determinado valor da posição deste elemento dentro do array. Por exemplo, para acessar o valor 10 no “`array([30, 20, 15, 10])`” passamos o comando `array[3]`, sendo 3 o índice do elemento 10 dentro de `array`. Todos arrays, listas, dicionários, tuplas ou qualquer elemento sequencial em qualquer linguagem de programação inicia seu índice com o valor zero.

Podemos também acessar pedaços do array, chamamos essa funcionalidade de *slicing*, neste caso passamos dois índices para o array, sendo o início e o fim do pedaço do array. Também podemos realizar o *slicing* de maneira condicional, para caso uma condição seja satisfeita, o array mantém ou não determinados valores. A tabela 4 apresenta alguns exemplos de *slicing* e indexação com arrays.

```
array = np.array([20, 30, 40, 50, 60, 70, 80, 90, 1000])
```

```
matriz = np.array([[10, 20, 30], [54, 50, 60], [100, 200, 300]])
```

Tabela 4 - Exemplos de indexação e slicing.

Código	Descrição
<code>array[3]</code>	retornará o valor da posição 4 no array.
<code>array[2: 4]</code>	retornará um array contendo os valores entre as posições 2 e 4.
<code>array[array > 50]</code>	retornará um array contendo apenas valores acima de 50.
<code>array[array == 90]</code>	retornará um array verificado contendo apenas valores iguais a 90.

<code>matriz[1:, 1:]</code>	retornará os valores da matriz do eixo x = 1 até o fim do eixo x, e do eixo y = 1 até o fim do eixo y.
-----------------------------	--

Funções úteis

Para realizar operações com base nas dimensões dos arrays, podemos usar algumas funções muito importantes, como: *shape*, *len*, *reshape*, *ndim*, *size*, *flatten* e *transpose*. A função *shape* mostrará as informações das dimensões do array, *len* mostrará o número de elementos presentes apenas na primeira dimensão do array, portanto, caso seja apenas uma lista (array unidimensional), a função *len* pode ser muito útil. A função *ndim* apresentará o número de dimensões presentes no array, *size* retornará a quantidade de elementos presentes no array, *reshape* realizará a transformação das dimensões do array, por exemplo, transformar um array de 4 linhas e 3 colunas para um array de 3 linhas e 4 colunas. A função *flatten* transforma um array multidimensional em um array unidimensional, realizando o achatamento dos dados. A função *transpose* realizará a transposição dos elementos do array. As funções e seus resultados são apresentados na tabela 5.

```
array = np.array([[100, 200, 300], [1, 2, 3], [50, 60, 70], [45, 130, 145]])
```

Tabela 5 - Funções e seus resultados.

Função	Resultado
<code>array.shape</code>	(4, 3), um array bidimensional com 4 linhas e 3 colunas.
<code>len(array)</code>	4, número de elementos presentes na primeira dimensão.
<code>array.ndim</code>	2, número de dimensões do array.

<code>array.size</code>	12, número total de elementos presentes no array.
<code>array.reshape(3, 4).shape</code>	(3, 4), um array bidimensional com 3 linhas e 4 colunas após o reshape.
<code>array.flatten()</code>	retornará um array unidimensional com todos os valores presentes no array bidimensional.
<code>array.transpose()</code>	transposição dos elementos do array.
<code>np.random.random((4, 3))</code>	cria uma matriz com 4 linhas e 3 colunas contendo apenas valores aleatórios.

Operações com arrays

Podemos calcular as operações aritméticas básicas dentro de um array, tais como: soma, subtração, multiplicação, divisão, exponenciação, entre outras. Quando temos dois arrays com formas iguais, também podemos realizar algumas operações e comparações. Quando possuímos arrays com formas diferentes, o método usado para operação se chama *Broadcasting*.

Conseguimos também aplicar funções universais, sendo funções matemáticas aplicadas a todos os elementos de um array. É também possível aplicar funções estatísticas dentro dos arrays, tais como: *mean*, *median* e *std*. Sendo *mean* para a média de valores presentes em um array, *median* a mediana dos valores presentes no array e *std* é o desvio padrão dos elementos do array.

Por fim, em funções que operam diretamente nas dimensões do array, utilizaremos a função *np.concatenate*, ela realizará a função entre dois arrays A e B. A concatenação pode ser feita em eixos selecionados, por

padrão a junção ocorre na dimensão número 1. As funções e seus resultados são apresentados na tabela 6.

```
a = np.array([10, 20, 30])
```

```
b = np.array([3])
```

Tabela 6 - Funções Numpy e seus resultados.

Função	Resultado
<code>np.mean(a)</code>	20
<code>np.median(a)</code>	20
<code>np.std(a)</code>	8.164
<code>a - b</code>	<code>array([7, 17, 27])</code>
<code>a + b</code>	<code>array([13, 23, 33])</code>
<code>a * b</code>	<code>array([30, 60, 90])</code>
<code>a / b</code>	<code>array([3.33, 6.66, 10])</code>
<code>a ** b</code>	<code>array([1000, 8000, 27000])</code>
<code>np.concatenate((a, b), axis=0)</code>	<code>array([10, 20, 30, 3])</code>
<code>np.concatenate((a, b), axis=1)</code>	Erro, pois os dois arrays não possuem dimensões iguais.



XPe

> Capítulo 3



Capítulo 3. Introdução ao Pandas

Pandas é uma biblioteca do Python muito utilizada em programação científica. Ela trouxe um plus ao Python, pois possibilita trabalhar com análise de dados sem ter que recorrer a outras linguagens. O nome Pandas se dá pela junção das palavras *Panel Datas* e possui como ícone da biblioteca o animal panda.

A biblioteca Pandas pode ser utilizada para várias finalidades, tais como: manipulação de dados, análise de dados, uso no mercado financeiro, economia, entre outros.

O que é Pandas?

Visto uma breve introdução da biblioteca Pandas, podemos definir ela como sendo uma biblioteca para manipulação e análise de dados, nesta biblioteca podemos ler, manipular, agregar e apresentar as análises exploratórias com poucos passos e em poucas linhas de código.

A principal estrutura do Pandas é o *DataFrame*, que simula uma tabela de dados com linhas e colunas. Em uma simples analogia, *DataFrame* é como uma planilha de Excel ou uma tabela de banco de dados, contendo linhas, colunas e índices. Outra estrutura muito interessante e que é altamente usada é a *Série*. Séries são matrizes unidimensionais capazes de armazenar dados em qualquer tipo. Uma série pode suportar indexação sendo por valor inteiro ou por valores em rótulos.

Séries e DataFrames

Dentro da biblioteca Pandas, podemos definir que a diferença entre séries e data frames são suas dimensões. Enquanto um data frame pode variar sua dimensão de acordo com a tabela processada, as séries serão sempre do mesmo tamanho. Podemos analisar em que série seria o conjunto

de informações de uma linha da tabela (*data frame*), ou seja, os dados de cada coluna de uma única linha da tabela. DataFrame pode ser definido como sendo todo o conjunto de dados da tabela. Na figura 15 apresentamos as séries e na figura 16 apresentamos um dataframe.

Figura 15 - Exemplo de construção de uma série.

```
dados = {"nome": "william", "altura": 1.88, "idade": 26, "cidade": "Cambará", "estado": "Paraná"}
series = pd.Series(data=dados, index=dados.keys())
series
```

nome	william
altura	1.88
idade	26
cidade	Cambará
estado	Paraná
dtype:	object

Na Figura 15, utilizamos um dicionário contendo as informações da série a ser criada, mas uma série também pode ser o de dados de uma única linha do *DataFrame*.

Figura 16 - Exemplo de construção de um *DataFrame*.

```
import numpy as np
nomes = np.array(["William", "Carlos", "Jõao", "Pedro"])
idades = np.array([26, 20, 30, 25])
alturas = np.array([1.88, 1.90, 1.75, 1.80])

data = []
for nome, idade, altura in zip(nomes, idades, alturas):
    data.append([nome, idade, altura])

data = np.array(data)
dataframe = pd.DataFrame(data, columns=["Nomes", "Idades", "Alturas"])
dataframe
```

	Nomes	Idades	Alturas
0	William	26	1.88
1	Carlos	20	1.9
2	Jõao	30	1.75
3	Pedro	25	1.8

Na figura 16, utilizamos 3 arrays contendo dados de diferentes categorias. Através destes 3 arrays, combinamos cada informação dentro

deles para formar um *Data Frame* e definimos as colunas como *Nomes*, *Idades e Alturas*.

Leitura de Arquivos

Em Pandas podemos realizar leituras de arquivos de tabelas de acordo com sua extensão. Existem funções nativas para leitura de dados, podendo ser com extensão *csv*, *xlsx*, *json*, *xml*, *html*, entre outras.

Como definido no Capítulo 1, utilizaremos o dataset *Breast Cancer Wisconsin Diagnostic* para realizar todas nossas tarefas de processamento dos dados. Realizaremos o download dentro do *datafolder* do seguinte arquivo: [link](#). Neste arquivo a extensão é *.data*, mas as colunas são delimitadas por vírgulas, nesse caso podemos simplesmente alterar a extensão para *.csv* e indicar que o delimitador são vírgulas ou manter como *.data* e indicar o delimitador.

Portanto, nessa situação, podemos utilizar o seguinte comando para realizar a leitura do dataset.

```
dataframe = pandas.read_csv("breast-cancer-wisconsin.data",  
delimeter=",")
```

Manipulação de dataframes

Visto que, geralmente, tabelas possuem vários dados, dessa forma, podemos utilizar algumas funções da biblioteca Pandas para nos auxiliar na visualização dos dados. Podemos descobrir as dimensões de um data frame, quantas linhas, quantas colunas, conseguimos descobrir também a média de aparições em colunas numéricas, verificar variáveis categóricas, entre outras funcionalidades.

Para obtermos informações relacionadas ao data frame, podemos utilizar as funções descritas na tabela 7.

Tabela 7 - Funções de informações do data frame.

Função	Descrição
<code>dataframe.head()</code>	Apresenta as 5 primeiras linhas do dataframe.
<code>dataframe.head(10)</code>	Apresenta as 10 primeiras linhas do dataframe.
<code>dataframe.tail()</code>	Apresenta as 5 últimas linhas do dataframe.
<code>dataframe.tail(10)</code>	Apresenta as 10 últimas linhas do dataframe.
<code>dataframe.info()</code>	Apresenta informações estruturais do data frame, mostra os nomes das colunas e os tipos de dados que essas colunas comportam.
<code>dataframe.columns</code>	Apresenta em forma de Index as colunas do data frame.
<code>dataframe.isna()</code>	Valida o valor em cada célula do data frame e verifica se ele é vazio, None ou NaN. Retorna a outro dataframe com valores True ou False, em que True será caso o valor da célula esteja vazio, None ou Nan.
<code>dataframe.describe()</code>	Apresenta uma breve descrição das variáveis do data frame.
<code>dataframe.describe(include=["O"])</code>	Apresenta uma breve descrição das variáveis categóricas do data frame.
<code>dataframe.describe(include="all")</code>	Apresenta uma breve descrição de

	todas as variáveis do data frame.
--	-----------------------------------

Para averiguar melhor as funções *describe*, podemos detalhar elas aqui. A função *describe* sem nenhum parâmetro apresentará algumas informações estatísticas de cada coluna que possui dados numéricos, tais como: *count* (somatório de todas as células da coluna), *mean* (média dos valores da coluna), *std* (desvio padrão dos valores da célula em relação à média), *min* (menor valor da coluna), 25% (primeiro quartil), 50% (segundo quartil ou mediana), 75% (terceiro quartil) e *max* (é o valor máximo da coluna). A função *describe* com `include=["O"]` apresentará contagens para as variáveis categóricas, tais como: *count* (quantidade de valores categóricos presentes na coluna), *unique* (quantidade de variáveis categóricas), *top* (primeira variável categórica) e *freq* (apresenta a maior frequência no data frame para uma dada variável categórica). A função *describe* com parâmetro `include="all"` mostrará todas as contagens, tanto para variáveis categóricas quanto para as variáveis descritivas (numéricas).



XPe

> Capítulo 4



Capítulo 4. Pandas

Visto que tivemos uma introdução básica da biblioteca Pandas, podemos agora trabalhar com métodos mais complexos e também apresentar exemplos do uso da ferramenta. Neste capítulo veremos como realizar operações matemáticas utilizando a estrutura básica do Pandas (*DataFrame*), veremos possíveis processamentos de junção e concatenação de dados e também como agrupar e agregar dados de diferentes tabelas utilizando funções já prontas na biblioteca.

Operações com DataFrames

Para iniciar os assuntos relacionados a operações em data frames, podemos começar com os métodos *loc* e *iloc*. Os dois possuem funcionalidades semelhantes, no entanto, são diferentes. A função *iloc* realizará a seleção dos dados com base em valores inteiros, ou seja, similar ao fatiamento de matrizes detalhado no Capítulo 2. A função *loc* realizará a seleção com base nos nomes das variáveis.

Tomando como base o dataset utilizado posteriormente (*breast cancer*), apresentaremos alguns exemplos de uso dos métodos *iloc* e *loc*.

Figura 17 - Selecionando linhas com *iloc*, retorno em formato de Séries.

```
dataframe.iloc[1]
```

Sample code number	1015425
Clump Thickness	3
Uniformity of Cell Size	1
Uniformity of Cell Shape	1
Marginal Adhesion	1
Single Epithelial Cell Size	2
Bare Nuclei	2
Bland Chromatin	3
Normal Nucleoli	1
Mitoses	1
Class	2

Name: 1, dtype: object

Figura 18 - Selecionando linhas com *iloc*, retorno em formato de DataFrame.

▶ dataframe.iloc[[1]]

	Sample code number	Clump Thickness	Uniformity of Cell Size	Uniformity of Cell Shape	Marginal Adhesion	Single Epithelial Cell Size	Bare Nuclei
1	1015425	3	1	1	1	2	2

Figura 19 - Seleção de múltiplas linhas com *iloc*, retorno em formato de DataFrame.

▶ dataframe.iloc[1: 5]

	Sample code number	Clump Thickness	Uniformity of Cell Size	Uniformity of Cell Shape	Marginal Adhesion	Single Epithelial Cell Size	Bare Nuclei
1	1015425	3	1	1	1	2	2
2	1016277	6	8	8	1	3	4
3	1017023	4	1	1	3	2	1
4	1017122	8	10	10	8	7	10

Figura 20 - Seleção de colunas com *iloc*, selecionado todas as linhas da coluna *Clump Thickness*.

▶ dataframe.iloc[:, 1]

0	5
1	3
2	6
3	4
4	8
	..
693	3
694	2
695	5
696	4
697	4

Name: Clump Thickness, Length: 698, dtype: int64

Figura 21 - Fatiamento de linhas e colunas com *iloc*, fatiamento das linhas da segunda coluna até a sexta.

```
dataframe.iloc[:, 1: 6]
```

	Clump Thickness	Uniformity of Cell Size	Uniformity of Cell Shape	Marginal Adhesion	Single Epithelial Cell Size
0	5	4	4	5	7
1	3	1	1	1	2
2	6	8	8	1	3
3	4	1	1	3	2
4	8	10	10	8	7
...
693	3	1	1	1	3
694	2	1	1	1	2
695	5	10	10	3	7
696	4	8	6	4	3
697	4	8	8	5	4

698 rows x 5 columns

Figura 22 - Seleção de linhas e colunas com *iloc*, selecionando dados da terceira quarta coluna até a sexta da primeira linha.

```
dataframe.iloc[0, 3: 6]
```

Uniformity of Cell Shape	4
Marginal Adhesion	5
Single Epithelial Cell Size	7
Name: 0, dtype: object	

Visto que apresentamos alguns exemplos de funcionalidade do método *iloc*, podemos apresentar o método *loc*. Os dois métodos possuem características semelhantes, no entanto, o método *loc* pode ser usado de maneira mais simples na seleção de colunas.

Figura 23 - Selecionando com *loc* as linhas 3, 4 e 5 nas colunas Uniformity of Cell Shape, Marginal Adhesion, Single Epithelial Cell Size.

```
dataframe.loc[[3, 4, 5], ["Uniformity of Cell Shape", "Marginal Adhesion", "Single Epithelial Cell Size"]]
```

	Uniformity of Cell Shape	Marginal Adhesion	Single Epithelial Cell Size
3	1	3	2
4	10	8	7
5	1	1	2

Figura 24 - Selecionando apenas amostras em que valores de *Uniformity of Cell Size* é maior que 5.

```
dataframe.loc[dataframe["Uniformity of Cell Size"] > 5]
```

	Sample code number	Clump Thickness	Uniformity of Cell Size	Uniformity of Cell Shape	Marginal Adhesion	Single Epithelial Cell Size	Bare Nuclei	Bland Chromatin	Normal Nucleoli	Mitoses	Class
2	1016277	6	8	8	1	3	4	3	7	1	2
4	1017122	8	10	10	8	7	10	9	7	1	4
13	1044572	8	7	5	10	7	9	5	5	4	4
17	1050670	10	7	7	6	4	10	4	1	2	4
31	1072179	10	7	7	3	8	5	7	4	3	4
...
680	1371026	5	10	10	10	4	10	5	6	3	4
690	695091	5	10	10	5	4	5	4	4	1	4
695	888820	5	10	10	3	7	3	8	10	2	4
696	897471	4	8	6	4	3	4	10	6	1	4
697	897471	4	8	8	5	4	5	10	4	1	4

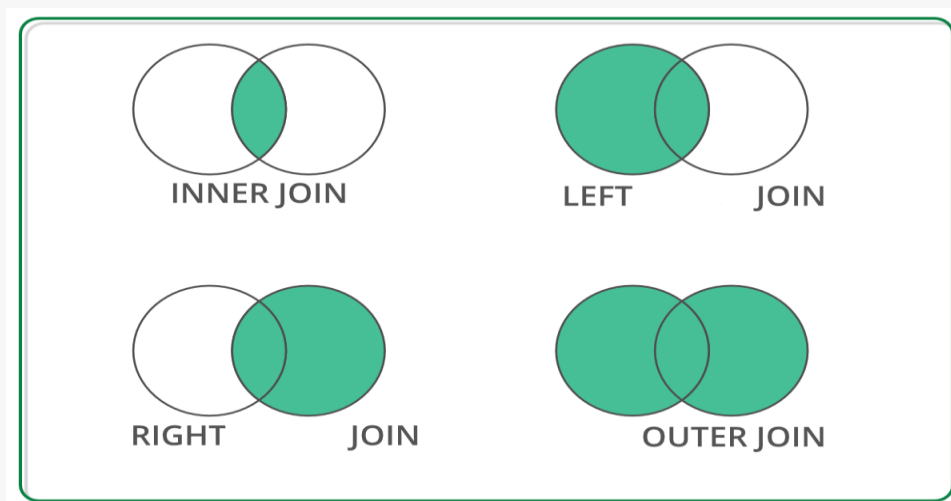
148 rows × 11 columns

DataFrame Merge e Concat

Em Pandas, a função *merge* irá unir um ou mais dataframes com base em colunas que eles possuem em comum, enquanto a função *concat* fará a união de data frames através do uso de um eixo.

Para operar a função *merge*, primeiro precisamos entender seu funcionamento e seus parâmetros. Para realizar a junção de duas tabelas com base em suas colunas, a função *merge* toma como parâmetro um método de junção, que pode ser *left*, *right*, *outer* ou *inner*. A figura 25 ilustra como cada dado é selecionado dentro da tabela utilizando o método de junção.

Figura 25 - Métodos de junção.



De acordo com a figura 25, o método *inner join* é o parâmetro de junção *inner*, *left join* é o parâmetro *left*, *right join* é o parâmetro *right* e *outer join* é o parâmetro *outer*. Todos estes métodos são formas de selecionar linhas de amostras de dados em diferentes tabelas de dados. Cada parâmetro citado terá sua própria estrutura de junção e com ela trará resultados de acordo com sua estrutura. Para exemplificar tal funcionamento, serão aplicadas as quatro possibilidades da função de *merge*, todas utilizando os dois mesmos dataframes, apresentados na figura 26.

Figura 26 - DataFrames para processamento.

[45] df_clientes

	nome	idade	cidade
0	William	26	Cambará
1	Kleyton	25	Ubatuba
2	Kevin	30	Santos
3	Arthur	22	Carlópolis
4	Calvin	30	Santos

df_vendas

	valor	nome	horário
0	33.20	William	10:25am
1	200.20	Arthur	11:30am
2	100.10	Kleyton	11:25pm
3	15.20	Kevin	08:25am
4	41.25	Pedro	09:25am

(A) Data Frame clientes

(B) Data Frame vendas

O resultado da junção dos Data Frames utilizando o parâmetro *left* é apresentado na figura 27.

Figura 27 - Junção com a função *left*.

```
# left join
compras_e_clientes = pd.merge(df_vendas, df_clientes, on="nome", how="left")
compras_e_clientes
```

	valor	nome	horário	idade	cidade
0	33.20	William	10:25am	26.0	Cambará
1	200.20	Arthur	11:30am	22.0	Carlópolis
2	100.10	Kleyton	11:25pm	25.0	Ubatuba
3	15.20	Kevin	08:25am	30.0	Santos
4	41.25	Pedro	09:25am	NaN	NaN

Nota-se que na amostra de index 4, idade e cidade ficaram como *NaN* (*Not a Number*) no resultado da junção, isso se dá porque esse index só existe no data frame vendas e não em clientes, e para esse caso o data frame principal era o da esquerda, que no contexto é o de vendas. O resultado da junção dos data frames utilizando o parâmetro *right* é apresentado na figura 28.

Figura 28 - Junção com a função *right*.

```
[38] # right join
compras_e_clientes = pd.merge(df_vendas, df_clientes, on="nome", how="right")
compras_e_clientes
```

	valor	nome	horário	idade	cidade
0	33.2	William	10:25am	26	Cambará
1	100.1	Kleyton	11:25pm	25	Ubatuba
2	15.2	Kevin	08:25am	30	Santos
3	200.2	Arthur	11:30am	22	Carlópolis
4	NaN	Calvin	NaN	30	Santos

Nota-se que a amostra de index 4 ficou com valor e horário como *NaN*, isso ocorreu porque neste caso o data frame principal era o da direita, que nessa situação era o de clientes e este cliente em específico não realizou nenhuma compra, por isso a falta de valores para essa ocorrência. O resultado da junção dos data frames utilizando o parâmetro *outer* é apresentado na figura 29.

Figura 29 - Função com a junção *outer*.

```
[40] # outer join
compras_e_clientes = pd.merge(df_vendas, df_clientes, on="nome", how="outer")
compras_e_clientes
```

	valor	nome	horário	idade	cidade
0	33.20	William	10:25am	26.0	Cambará
1	200.20	Arthur	11:30am	22.0	Carlópolis
2	100.10	Kleyton	11:25pm	25.0	Ubatuba
3	15.20	Kevin	08:25am	30.0	Santos
4	41.25	Pedro	09:25am	NaN	NaN
5	NaN	Calvin	NaN	30.0	Santos

Nota-se que nas amostras de index 4 e 5 apareceram os valores *NaN*, isso ocorreu porque neste caso aconteceu a junção dos dois data frames completos e algumas colunas não existem para todos os valores dos data frames. O resultado da junção dos data frames utilizando o parâmetro *inner* é apresentado na figura 30.

Figura 30 - Função com a junção *inner*.

```
[41] # inner join
compras_e_clientes = pd.merge(df_vendas, df_clientes, on="nome", how="inner")
compras_e_clientes
```

	valor	nome	horário	idade	cidade
0	33.2	William	10:25am	26	Cambará
1	200.2	Arthur	11:30am	22	Carlópolis
2	100.1	Kleyton	11:25pm	25	Ubatuba
3	15.2	Kevin	08:25am	30	Santos

Nota-se que nenhum valor *NaN* foi selecionado, isso ocorreu porque na estrutura de *inner* apenas as amostras que possuem valores tanto na tabela A quanto na tabela B serão selecionadas, *inner* é a intersecção entre duas tabelas de dados.

A figura 31 apresenta como concatenar as duas tabelas utilizando o eixo y, ou seja, concatenando uma tabela B abaixo da tabela A.

Figura 31 - Concatenando duas tabelas.

```
concatenation = pd.concat([df_clientes, df_vendas])
concatenation
```

	nome	idade	cidade	valor	horário
0	William	26.0	Cambará	NaN	NaN
1	Kleyton	25.0	Ubatuba	NaN	NaN
2	Kevin	30.0	Santos	NaN	NaN
3	Arthur	22.0	Carlópolis	NaN	NaN
4	Calvin	30.0	Santos	NaN	NaN
0	William	NaN	NaN	33.20	10:25am
1	Arthur	NaN	NaN	200.20	11:30am
2	Kleyton	NaN	NaN	100.10	11:25pm
3	Kevin	NaN	NaN	15.20	08:25am
4	Pedro	NaN	NaN	41.25	09:25am

Agrupamentos e Agregações

No Pandas temos uma função de agrupamento dos dados que agrupa, calcula algumas propriedades dos grupos formados e sumariza os resultados, essa função chama *groupby*. Podemos querer algo além de informações dos grupos, e, para isso, podemos usar a função *agg* de agregação.

A fim de exemplificar as funções de agrupamento e agregação, utilizaremos o *dataset breast cancer* como base para processamento. Para uma primeira impressão de processamento dos dados, podemos utilizar agrupamentos condicionais, ou seja, criar condições para agrupar os dados, conforme apresentado nas figuras 32 e 33.

Figura 32 - Agrupamento condicional, agrupando apenas amostras da classe maligno.

```
#agrupamento condicional
apenas_malignos = dataframe.loc[dataframe["Class"] == 4]
apenas_malignos
```

	Sample code number	Clump Thickness	Uniformity of Cell Size	Uniformity of Cell Shape	Marginal Adhesion	Single Epithelial Cell Size	Bare Nuclei	Bland Chromatin	Normal Nucleoli	Mitoses	Class
4	1017122	8	10	10	8	7	10	9	7	1	4
11	1041801	5	3	3	3	2	3	4	4	1	4
13	1044572	8	7	5	10	7	9	5	5	4	4
14	1047630	7	4	6	4	6	1	4	3	1	4
17	1050670	10	7	7	6	4	10	4	1	2	4
...
680	1371026	5	10	10	10	4	10	5	6	3	4
690	695091	5	10	10	5	4	5	4	4	1	4
695	888820	5	10	10	3	7	3	8	10	2	4
696	897471	4	8	6	4	3	4	10	6	1	4
697	897471	4	8	8	5	4	5	10	4	1	4

241 rows x 11 columns

Figura 33 - Agrupamento condicional, selecionando apenas amostras benignas.

```
#agrupamento condicional
apenas_benignos = dataframe.loc[dataframe["Class"] == 2]
apenas_benignos
```

	Sample code number	Clump Thickness	Uniformity of Cell Size	Uniformity of Cell Shape	Marginal Adhesion	Single Epithelial Cell Size	Bare Nuclei	Bland Chromatin	Normal Nucleoli	Mitoses	Class
0	1002945	5	4	4	5	7	10	3	2	1	2
1	1015425	3	1	1	1	2	2	3	1	1	2
2	1016277	6	8	8	1	3	4	3	7	1	2
3	1017023	4	1	1	3	2	1	3	1	1	2
5	1018099	1	1	1	1	2	10	3	1	1	2
...
689	654546	1	1	1	3	2	1	1	1	1	2
691	714039	3	1	1	1	2	1	1	1	1	2
692	763235	3	1	1	1	2	1	2	1	2	2
693	776715	3	1	1	1	3	2	1	1	1	2
694	841769	2	1	1	1	2	1	1	1	1	2

457 rows x 11 columns

O agrupamento condicional é interessante ser utilizado quando não há necessidade de se agregar nenhum valor ao agrupamento feito, como nos exemplos acima, simplesmente selecionamos amostras benignas e malignas, mas poderíamos agregar valores a essas seleções feitas. A figura 34 apresenta o agrupamento feito pela função *groupby*.

Figura 34 - Agrupamento dos dados através da classe de cada amostra utilizando groupby.

```
agrupamentos = dataframe.groupby(["Class"])
agrupamentos.sum()
```

<ipython-input-33-36289d3521a3>:2: FutureWarning: The default value of numeric_only in DataFrameGroupBy.sum is deprecated. In a future version, numeric_only will be removed. Use numeric_only=True to silence this warning.

	Sample code number	Clump Thickness	Uniformity of Cell Size	Uniformity of Cell Shape	Marginal Adhesion	Single Epithelial Cell Size	Bland Chromatin	Normal Nucleoli	Mitoses
Class									
2	506276453	1349	606	660	624	969	959	590	486
4	241844687	1734	1584	1581	1337	1277	1441	1413	624

Nota-se na figura 34, que atribuímos valor ao resultado, conseguimos verificar o somatório de cada coluna de todas as amostras, é possível selecionar a média, mediana e desvio padrão de cada coluna também. A figura 35 apresenta a agregação feita para apresentar o somatório e a média de cada coluna do agrupamento feito no dataset breast cancer.

Figura 35 - Agregação de novos valores utilizando o agrupamento das classes.

```
agrupamentos.agg(["mean", "sum"]).T
```

<ipython-input-36-47719f931499>:1: FutureWarning: ['Bare Nuclei'] is deprecated. Use ['Bare Nucleoli'] instead.

	Class	2	4
Sample code number	mean	1.107826e+06	1.003505e+06
	sum	5.062765e+08	2.418447e+08
Clump Thickness	mean	2.951860e+00	7.195021e+00
	sum	1.349000e+03	1.734000e+03
Uniformity of Cell Size	mean	1.326039e+00	6.572614e+00
	sum	6.060000e+02	1.584000e+03
Uniformity of Cell Shape	mean	1.444201e+00	6.560166e+00
	sum	6.600000e+02	1.581000e+03
Marginal Adhesion	mean	1.365427e+00	5.547718e+00
	sum	6.240000e+02	1.337000e+03
Single Epithelial Cell Size	mean	2.120350e+00	5.298755e+00
	sum	9.690000e+02	1.277000e+03
Bland Chromatin	mean	2.098468e+00	5.979253e+00
	sum	2.098468e+00	5.979253e+00



XPe

> Capítulo 5



Capítulo 5. Mineração de dados

Visto que milhares de dados são gerados a cada segundo e que os métodos de big data são cada vez mais acionados no cotidiano dos cientistas de dados, ter domínio dos conceitos de mineração de dados se torna primordial para tal tarefa.

A mineração de dados é o processo de descoberta de informações em grandes conjuntos de dados. Através de análises matemáticas, podemos detectar tendências e padrões que existem nos dados, padrões que geralmente não são reconhecidos com a exploração de dados clássica devido ao alto volume de dados e por existirem relações muito complexas a serem desvendadas⁴.

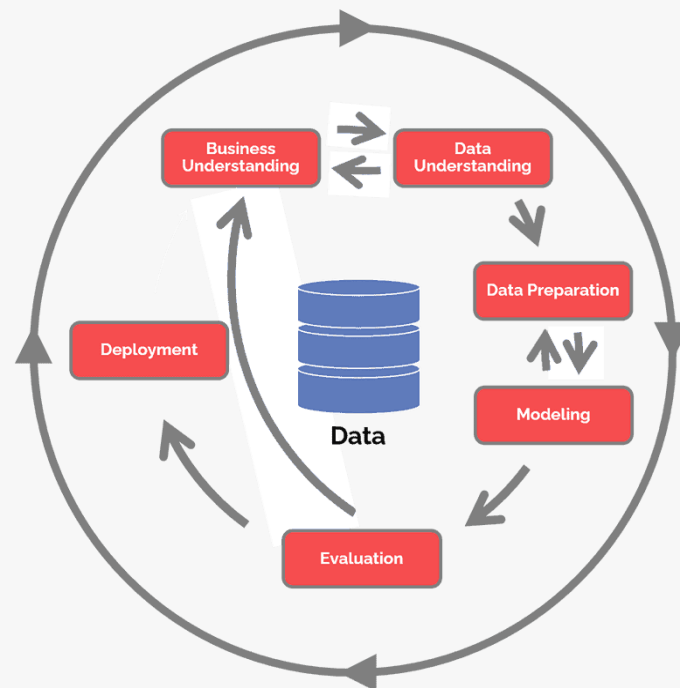
Cenários de mineração de dados

Minerar dados consiste em analisar uma quantidade de dados utilizando técnicas e algoritmos para extração de características. Para realizar tal feito, podemos destacar três metodologias mais importantes, tais como: CRISP-DM, KDD e SEMMA.

CRISP-DM ou *Cross Industry Standard Process for Data Mining* é um processo mais voltado ao negócio, suas principais características são: entendimento de negócio, entendimento dos dados, preparação dos dados, modelagem, avaliação e produção, conforme apresentado na figura 36.

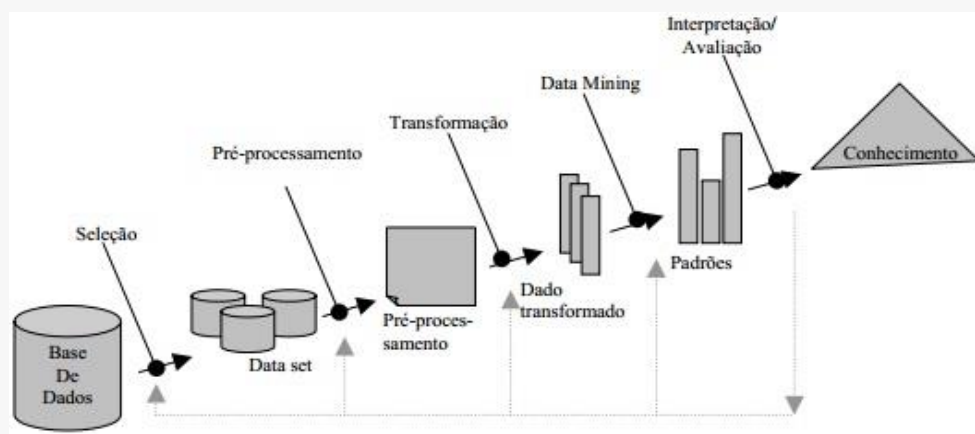
⁴ CONCEITOS de mineração de dados. Microsoft Build, 27 de setembro de 2022. Disponível em: <<https://learn.microsoft.com/pt-br/analysis-services/data-mining/data-mining-concepts?view=asallproducts-allversions>>. Acesso em: 25 abr. 2023.

Figura 36 - Metodologia CRISP-DM em diagrama.



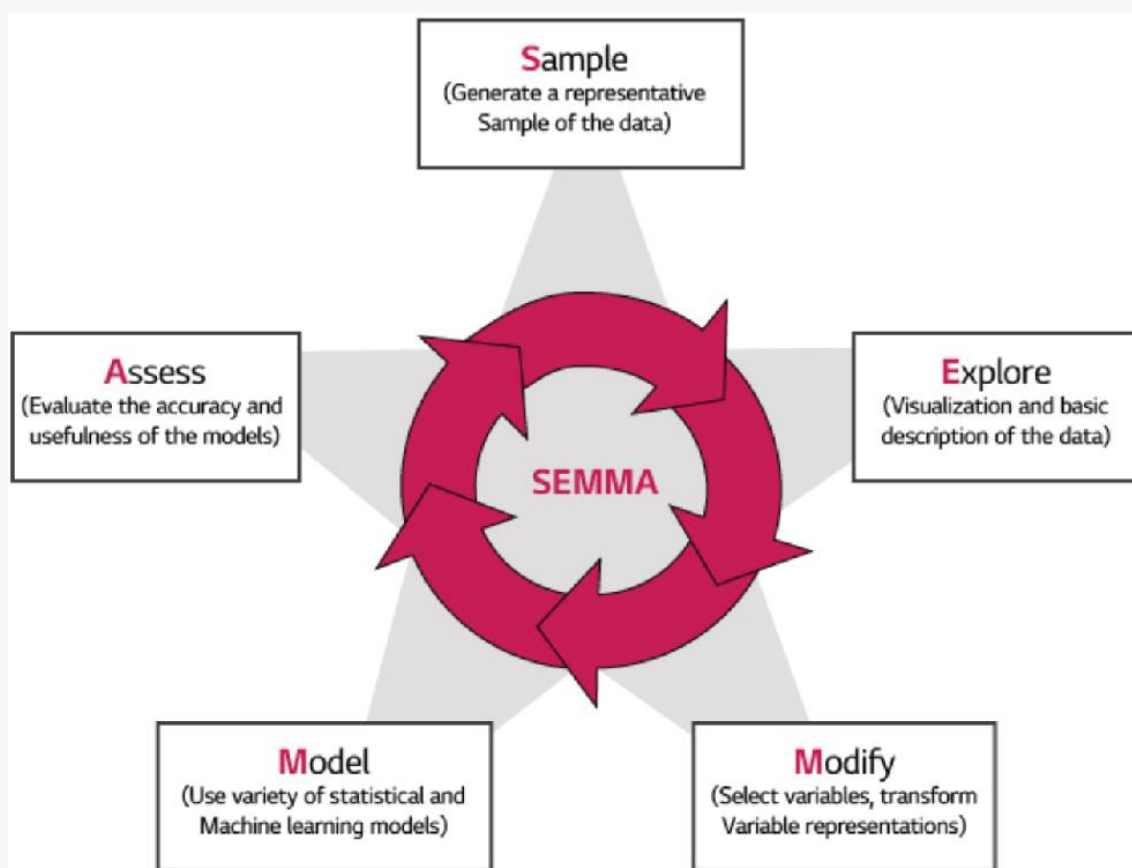
Outra metodologia muito interessante para o entendimento da mineração de dados é a KDD ou *Knowledge Discovery in Databases*. Talvez o processo mais clássico no meio de processamento de dados e mineração de informações, KDD é mais voltado para o levantamento de conhecimento, tendo como características: seleção de dados, pré-processamento e transformação, mineração de dados, interpretação e avaliação. A metodologia KDD é apresentada na figura 37.

Figura 37 - Metodologia KDD.



O último método para mineração de dados apresentado é o SEMMA ou *Sample Explore Modify Model and Assess*, também voltado para solucionar problemas de negócios, suas características são: amostragem, exploração, modificação, modelagem e avaliação. A figura 38 mostra o diagrama em estrela do método SEMMA.

Figura 38 - Metodologia SEMMA.



Ciência de Dados e Mineração

O primeiro passo a ser dado em mineração de dados é conhecer o dado que está sendo tratado. É necessário conhecer os tipos de dados presentes nas tabelas, levantar informações relevantes sobre os dados, até que fique mais familiarizado com o comportamento dos dados e suas respectivas características.

Como visto em capítulos anteriores, com Pandas podemos usar a função *describe* para analisar informações muito importantes do dataset,

como de dados numéricos e categóricos. Podemos também utilizar a função *info* para analisar a possibilidade de valores faltantes no dataset. Conseguimos também analisar a possibilidade de valores duplicados dentro do dataset e como tratar tais cenários.

Após a filtragem do dataset, podemos começar a analisar os dados com maior profundidade e buscar conhecê-los. Para isso iniciaremos nosso estudo sobre apresentação de dados em abstração, ou seja, informações em gráficos.

No próximo capítulo, veremos como analisar os dados com maior profundidade utilizando a biblioteca *Matplotlib* e *Seaborn*.



XPe

> Capítulo 6



Capítulo 6. Matplotlib

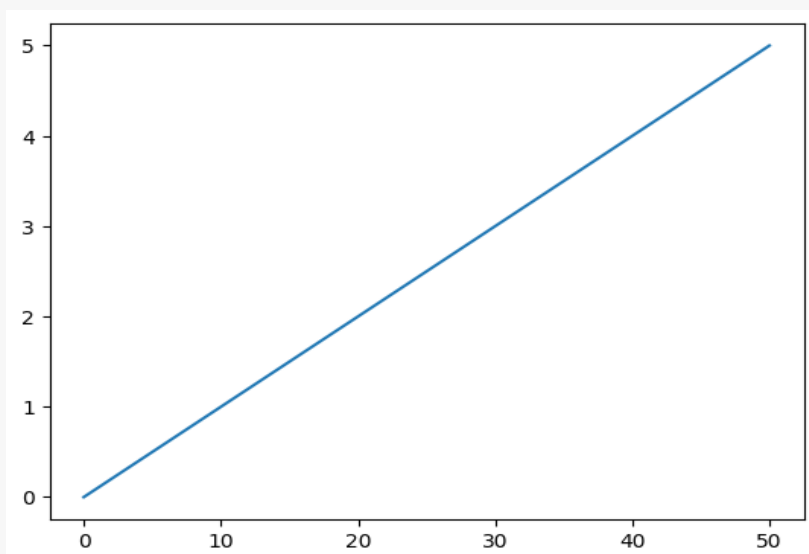
Dentro da ciência de dados, análises abstratas se tornam uma poderosa ferramenta perante o reconhecimento de possíveis padrões e tendências dentro de altos volumes de dados. Sendo assim, uma boa maneira de se analisar o comportamento dos dados é através de gráficos, apresentando dados estruturados e informações de relacionamento.

Com a linguagem Python, podemos utilizar algumas bibliotecas para apresentar dados estruturados em gráficos, sendo elas: *Matplotlib*, *Seaborn*, *Scikit-Image*, *OpenCV*, entre outras. Neste capítulo apresentaremos a biblioteca *Matplotlib* em parceria com a biblioteca *Seaborn*, desta forma, podemos tornar ainda mais poderosas nossas apresentações de dados.

O que é Matplotlib?

Focado em apresentação de dados, matplotlib é uma biblioteca de plotagem, que auxilia no reconhecimento de padrões, tendências e análise de dados. A criação de um simples gráfico é feita através de poucas linhas, a figura 39 apresenta um gráfico de linha simples.

Figura 39 - Gráfico simples com matplotlib.

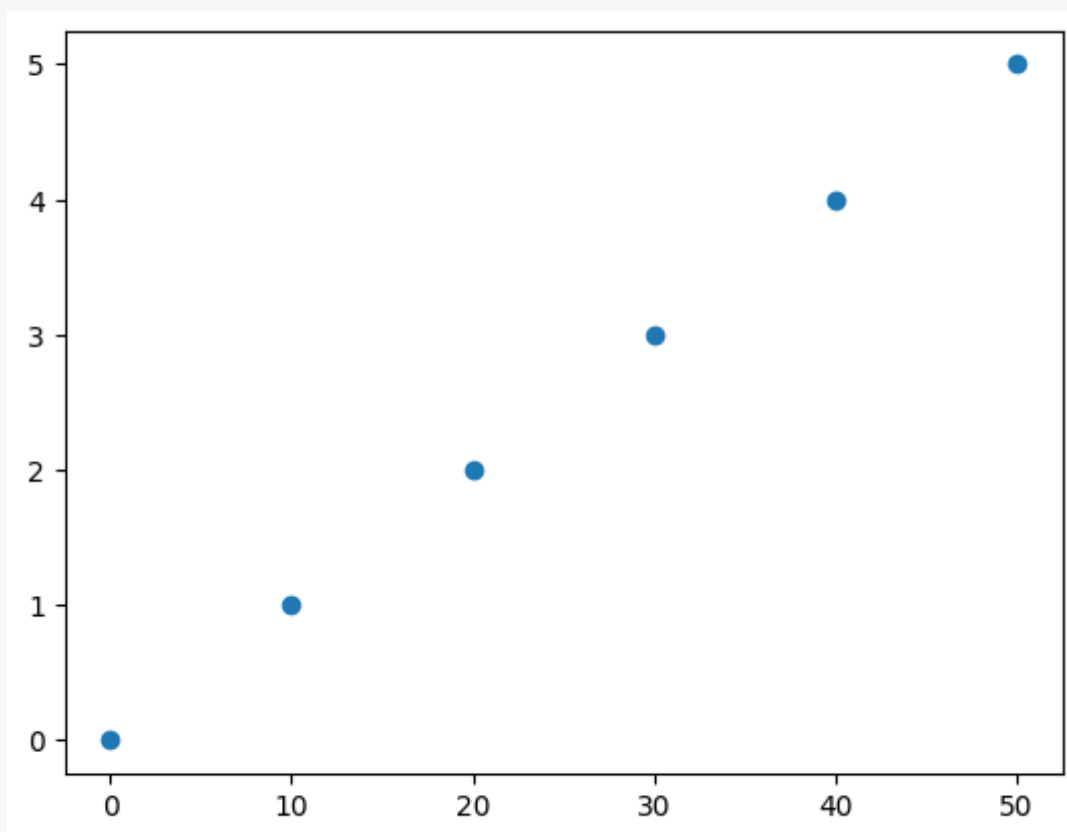


Para criação dos gráficos, principalmente em 2D, precisamos nos atentar aos valores para x e y. No exemplo da figura 39, os valores de x cresceram de 10 em 10, enquanto em y cresceram de 1 a 1. Para realizar tal ato, é altamente utilizado listas e arrays para carregar cada valor em x e cada valor em y a ser plotado. Para construir isto, utilizamos o seguinte código:

```
import matplotlib.pyplot as plt  
x = [0, 10, 20, 30, 40, 50]  
y = [0, 1, 2, 3, 4, 5]  
plt.plot(x, y)
```

O gráfico gerado é chamado de gráfico de linhas, podemos alterar o formato de como as amostras são apresentadas nos gráficos apresentando como pontos, por exemplo, conforme na figura 40.

Figura 40 - Gráfico simples em pontos.

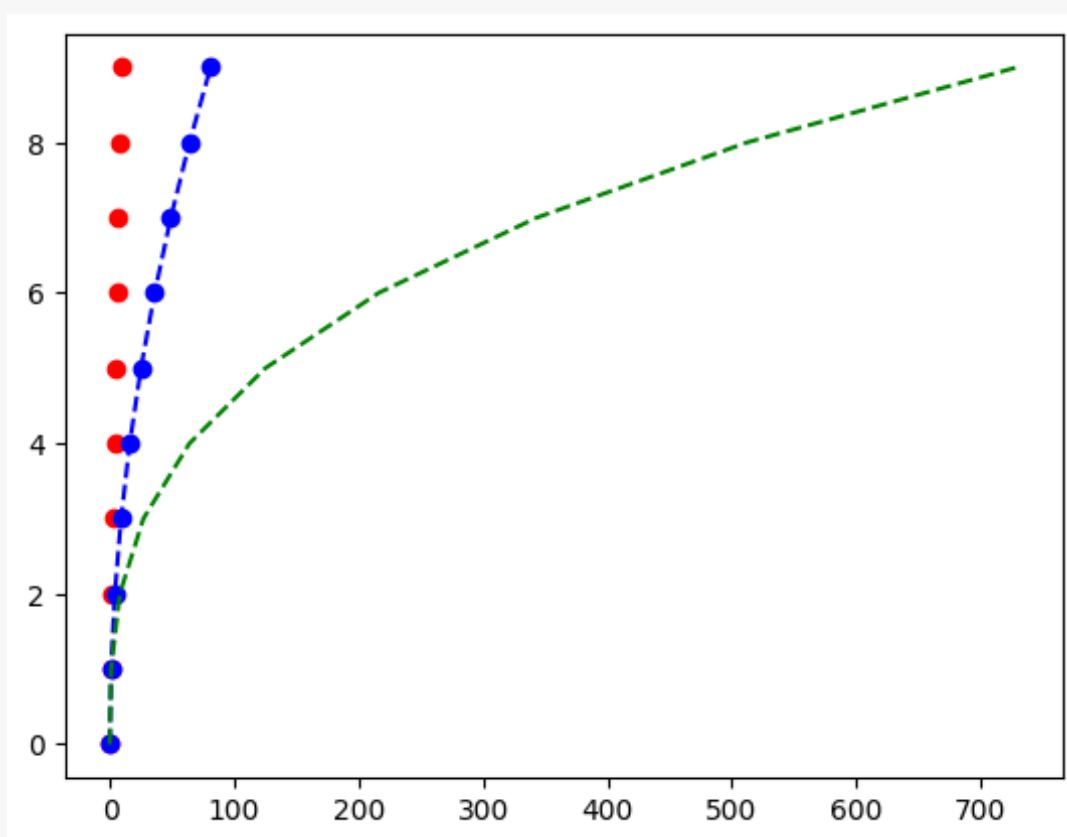


Para construir o gráfico apresentado na figura 40, utilizamos o seguinte código:

```
import matplotlib.pyplot as plt  
x = [0, 10, 20, 30, 40, 50]  
y = [0, 1, 2, 3, 4, 5]  
plt.plot(x, y, "o")
```

Nota-se que o código é praticamente o mesmo, no entanto, adicionamos um parâmetro a mais "o", com ele a biblioteca interpretará automaticamente que o formato de plot a ser feito é em pontos. Podemos adicionar várias informações em um único gráfico, conforme apresentado na figura 41.

Figura 41 - Gráfico com diferentes formas de apresentação.

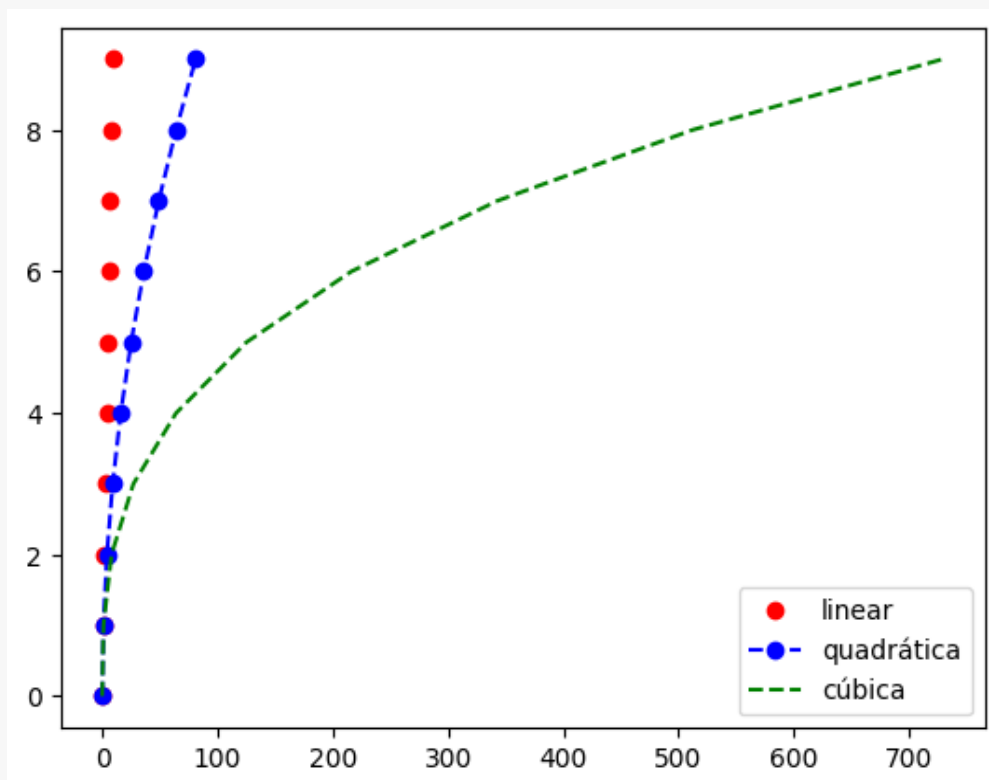


Para construir o gráfico apresentado na figura 41, utilizamos o seguinte código:

```
import numpy as np
y = np.arange(10)
linear = np.arange(10)
quadrático = np.arange(10) ** 2
cúbico = np.arange(10) ** 3
plt.plot(linear, y, "o", color="r")
plt.plot(quadrático, y, "--o", color="b")
plt.plot(cúbico, y, "--", color="g")
```

De acordo com o código apresentado, cada plotagem no gráfico possui um formato e uma cor específica, facilitando a sua visualização. No entanto, quem não enxerga o código não sabe o que cada linha representa, pela falta de uma legenda. Dessa forma, em matplotlib, podemos adicionar uma legenda que mostra qual é cada dado na plotagem, conforme apresentado na figura 42.

Figura 42 - Plotagem com legenda para cada linha.

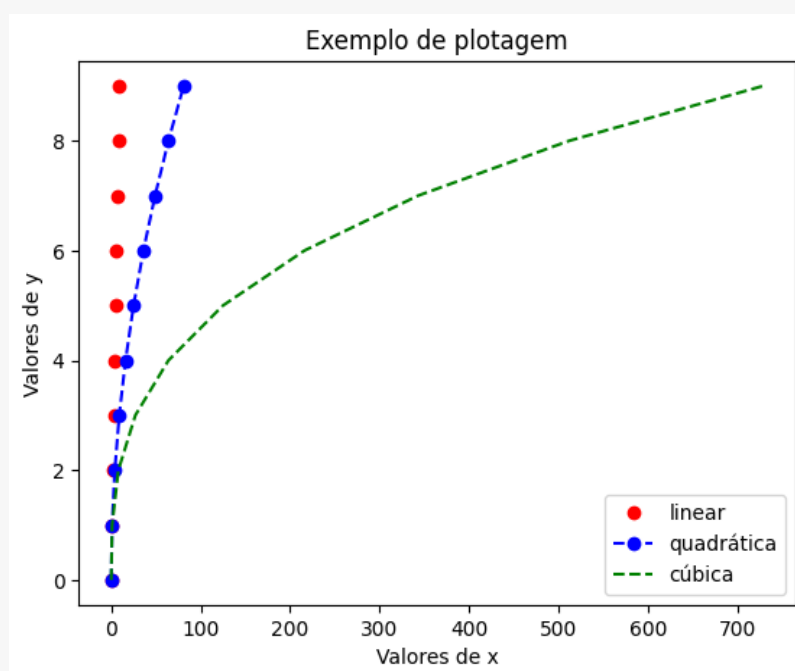


Para construir o gráfico apresentado na figura 42, utilizamos o seguinte código:

```
import numpy as np
y = np.arange(10)
linear = np.arange(10)
quadratico = np.arange(10) ** 2
cubico = np.arange(10) ** 3
plt.plot(linear, y, "o", color="r", label="linear")
plt.plot(quadratico, y, "--o", color="b", label="quadrática")
plt.plot(cubico, y, "--", color="g", label="cúbica")
plt.legend()
```

Nota-se a adição do parâmetro label dentro das plotagens, cada label aponta para a definição da plotagem feita no gráfico. Para cada plotagem, podemos adicionar um título para o gráfico e inserir os nomes dos eixos x e y para aumentar o nível de informação presente no gráfico, conforme a figura 43.

Figura 43 - Gráfico com título e eixos nomeados.



Para construir o gráfico presente na figura 43, utilizamos o seguinte algoritmo:

```
import numpy as np
y = np.arange(10)
linear = np.arange(10)
quadratico = np.arange(10) ** 2
cubico = np.arange(10) ** 3
plt.title("Exemplo de plotagem")
plt.plot(linear, y, "o", color="r", label="linear")
plt.plot(quadratico, y, "--o", color="b", label="quadrática")
plt.plot(cubico, y, "--", color="g", label="cúbica")
plt.xlabel("Valores em x")
plt.ylabel("Valores em y")
plt.legend()
```

O interessante da biblioteca é que ela nos oferece uma gama muito alta de formatos de gráficos que podem ser utilizados para apresentação de dados, mostraremos alguns formatos no tópico seguinte.

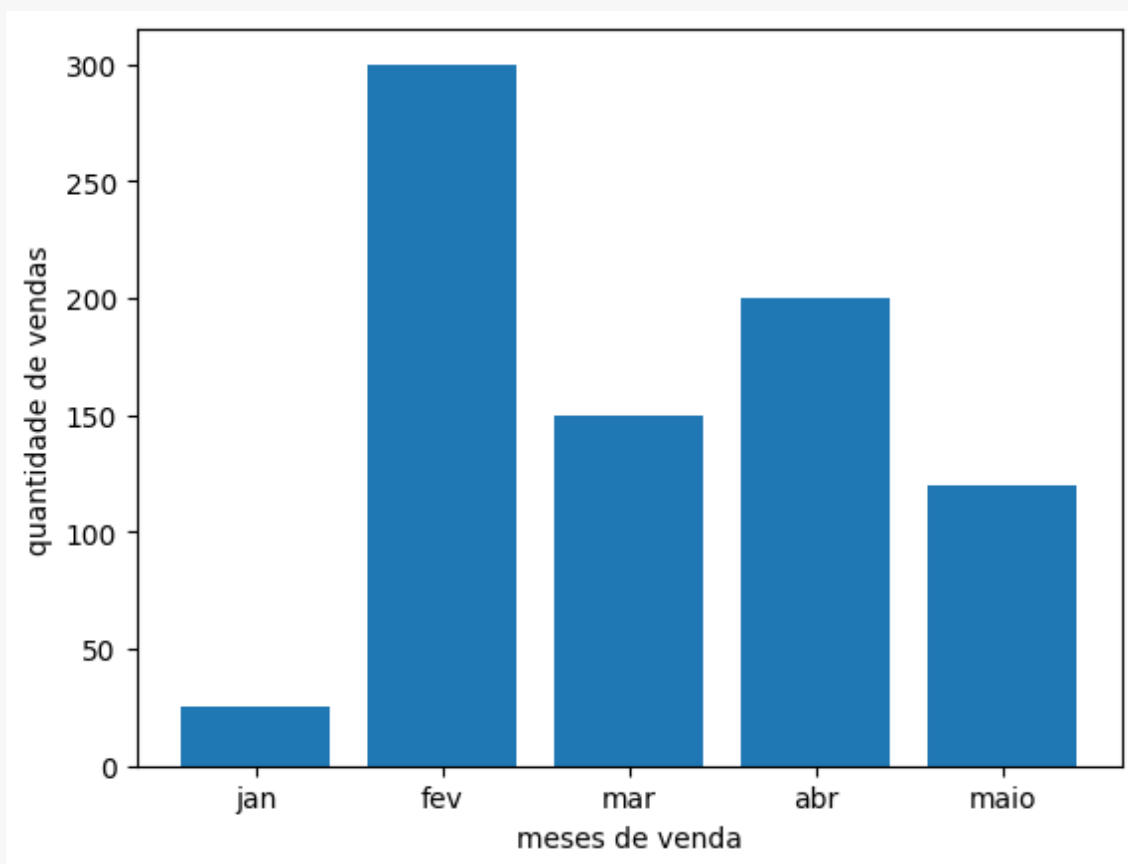
Tipos de gráficos e suas funcionalidades

Alguns gráficos auxiliam na compreensão de informações abstratas, um exemplo simples disso é o gráfico de barras para demonstração populacional, gráficos de erro para demonstração de taxas de variabilidade entre amostras, gráficos em pizza para demonstração de porcentagem, entre outros.

Neste capítulo veremos alguns gráficos de simples construção e que podem ser muito valiosos na apresentação de dados. A primeira estrutura de gráfico a ser apresentada é a de barras, ela é altamente indicada quando se busca cruzar dados que tenham algum tipo de relação e que, quando

confrontados, expressam um desempenho qualquer. A figura 44 apresenta um exemplo de gráfico de barras.

Figura 44 - Exemplo de gráfico de barras.



Para construir o gráfico presente na figura 44, utilizamos o seguinte algoritmo:

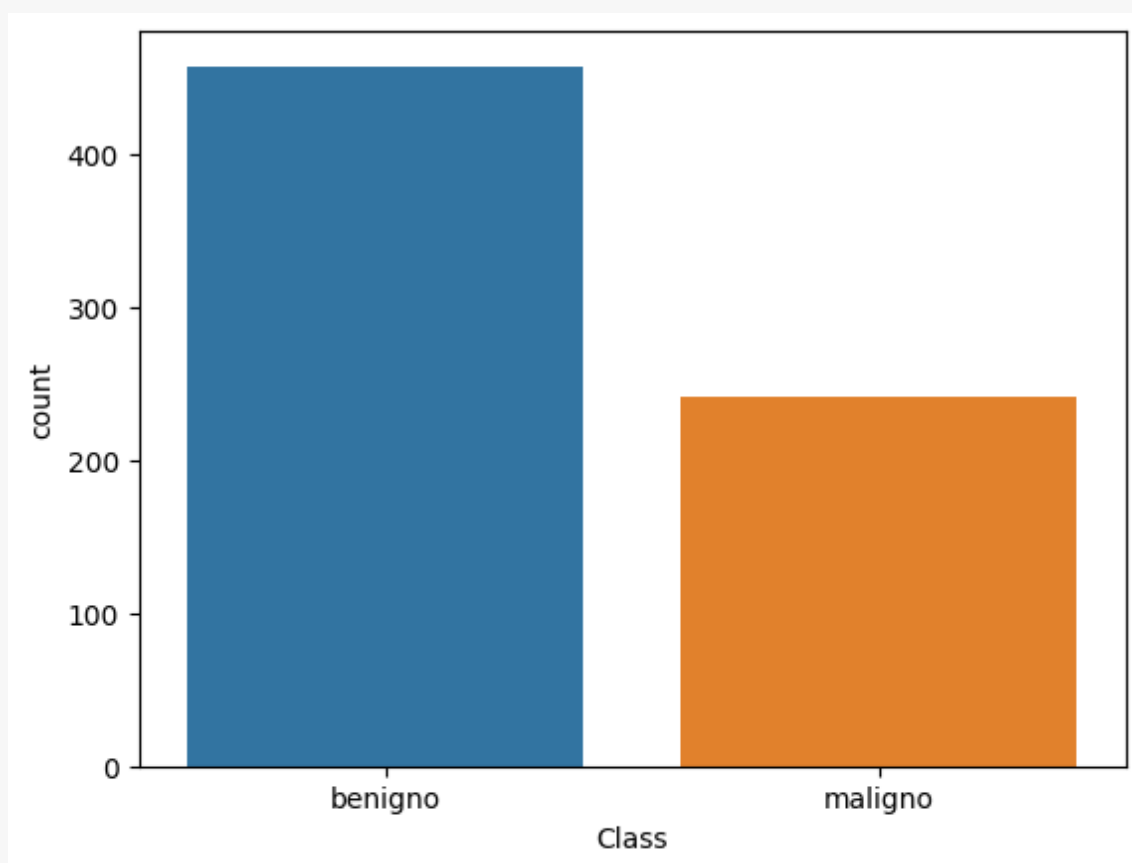
```
vendas_no_mes = [25, 300, 150, 200, 120]
meses = ["jan", "fev", "mar", "abr", "maio"]
plt.bar(meses, vendas_no_mes)
plt.xlabel("meses de venda")
plt.ylabel("quantidade de vendas")
```

Visto que podemos construir gráficos de maneiras bem simples e com poucas linhas de código, apresentaremos alguns exemplos com base em mineração de dados, como foi citado no capítulo 5, para isso, usaremos

os dados do dataset *breast cancer*. A fim de apresentar os dados utilizaremos três bibliotecas em conjunto, sendo elas: Pandas para processamento dos dados e Matplotlib e Seaborn para apresentação dos dados com os gráficos.

O primeiro exemplo é o de apresentação da quantidade de amostras de acordo com a classe da amostra, ou seja, utilizamos um gráfico de barras para apresentar quantas amostras temos da classe benigno e quantas existem da classe maligno, o gráfico é apresentado na figura 45.

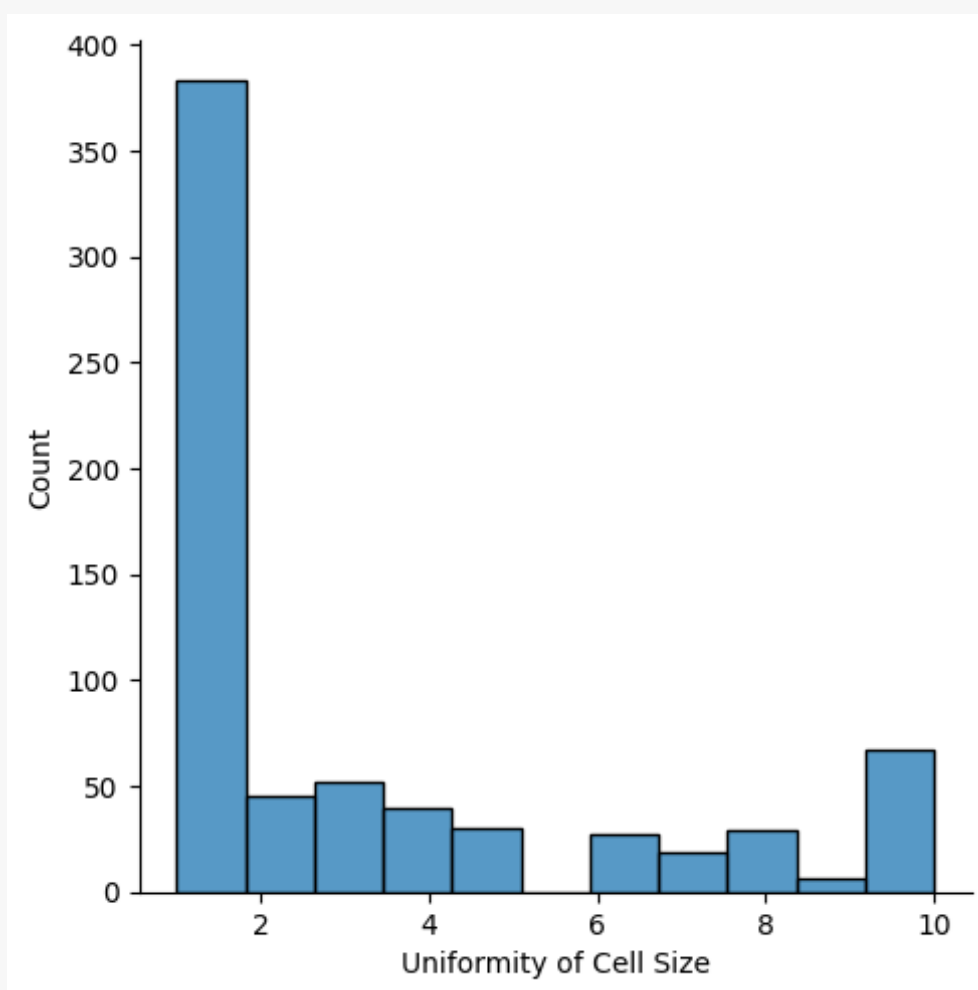
Figura 45 - Contagem e apresentação de amostras por classe.



Com esse gráfico em mãos podemos ter uma noção da quantidade de amostras para cada classe e começar a discutir os tipos de algoritmos a serem utilizados, visto que existe um desbalanceamento entre as duas classes, há um valor considerável de amostras de câncer benigno a mais do que maligno.

Podemos também analisar a distribuição dos dados para uma determinada variável do dataset, por exemplo, na figura 46 apresentamos a distribuição da variável *Uniformity of the cell size*.

Figura 46 - Distribuição dos dados na variável *Uniformity of Cell Size*.

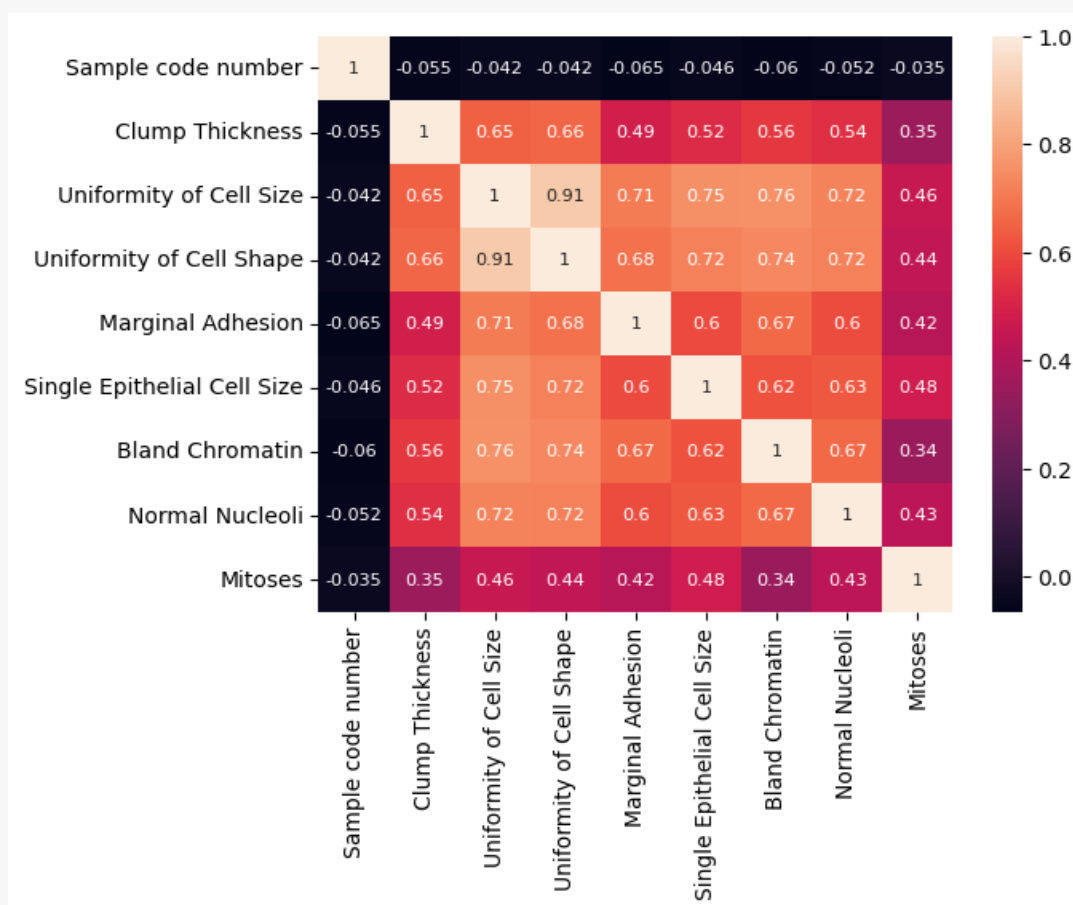


De acordo com a figura 46, nota-se com a quantidade de amostras que a uniformidade do tamanho da célula igual a 1 é muito maior do que qualquer outra uniformidade, podemos dizer que a distribuição para este valor é muito mais alta do que para os outros valores, o que não condiz com o “padrão” da distribuição. Para enxergar este padrão citado, podemos utilizar um gráfico chamado de *box plot* a fim de auxiliar na visualização da distribuição dos dados. O box plot fornece informação sobre as seguintes características do conjunto de dados: localização, dispersão, assimetria, comprimento da cauda e outliers (medidas discrepantes).

Visto que podemos analisar cada variável de maneira isolada e descobrir seus impactos, conseguimos verificar se existe alguma correlação entre duas ou mais variáveis através das matrizes de correlação. A biblioteca Pandas permite que utilizemos 3 métodos de correlação: Pearson, Spearman e Kendall.

Após a criação da matriz, ainda assim existe um grau alto de complexidade para verificar a correlação entre as variáveis, principalmente se estivermos trabalhando com uma base que possui uma quantidade elevada de colunas. Para isso, podemos utilizar o *heatmap*, um gráfico que aponta em nível de cor a correlação entre duas variáveis. A figura 47 apresenta um exemplo de *heatmap* da matriz de correlação utilizando a base *breast cancer*.

Figura 47 - Matriz de correlação entre as variáveis do dataset *Breast Cancer*.





XPe

> Capítulo 7



Capítulo 7. Scipy

Tendo em vista que em ciência de dados são utilizados vários métodos matemáticos para diferentes análises, sejam elas: extração de características, reconhecimento de padrões, algoritmos de aprendizado não supervisionado, entre outros. Neste capítulo veremos as aplicações e possíveis cenários de uso da biblioteca *Scipy*, que tem como intuito facilitar o uso de matemática computacional e computação científica.

O que é Scipy?

O SciPy pode ser comparado a outras bibliotecas de computação científica padrão, como a GSL (GNU Scientific Library para C e C + +), ou caixas de ferramentas do Matlab. O SciPy é o pacote principal de rotinas científicas em Python, que se destina a operar de forma eficiente em matrizes numpy, de modo que numpy e scipy trabalhem lado a lado⁵.

Podemos indicar como sendo os principais módulos da biblioteca Scipy os presentes na tabela 8. O pacote scipy contém várias ferramentas dedicadas a problemas comuns em computação científica. Seus diferentes submódulos correspondem a diferentes aplicações, tais como interpolação, integração, otimização, processamento de imagens, estatísticas, funções especiais, entre outras.

⁵ SCIPY. Estruturas UFPR. Disponível em: <<http://www.estruturas.ufpr.br/disciplinas/pos-graduacao/introducao-a-computacao-cientifica-com-python/introducao-python/capitulo-4-scipy/>>. Acesso em: 25 abr. 2023.

Tabela 8 - Módulos Scipy e suas funcionalidades.

Módulo	Descrição
scipy.io	Entrada e saída de arquivos
scipy.special	Funções matemáticas especiais
scipy.linalg	Operações com álgebra linear
scipy.constants	Constantes matemáticas e físicas
scipy.ndimage	Processamento de imagens digitais
scipy.signal	Processamento de sinais
scipy.interpolate	Interpolação entre dados
scipy.optimize	Otimização e ajuste de variáveis
scipy.stats	Estatísticas e valores aleatórios
scipy.cluster	Quantização de vetores, algoritmos não supervisionados e extração de características.

Como apresentado na tabela 8, scipy é uma biblioteca muito versátil em termos de métodos computacionais a serem usados, podendo ser feito: processamento de sinais, processamento de imagens, otimização de parâmetros, entre outros.

Álgebra Linear com Scipy

Álgebra Linear é um ramo da matemática que lida com equações e funções lineares que são representadas através de matrizes e vetores. Em geral, com scipy conseguimos realizar operações algébricas de maneira simples e com poucas linhas de código, principalmente utilizando matrizes como entrada. Podemos utilizar as funções conforme apresentado na tabela 9.

Tabela 9 - Funções de álgebra linear com scipy e suas operações.

Função	Descrição
<code>linalg.inv</code>	Calcula a matriz inversa de uma determinada matriz de entrada A.
<code>linalg.cosm</code>	Calcula a matriz cosine de uma determinada matriz de entrada A.
<code>linalg.sinm</code>	Calcula a matriz seno de uma determinada matriz de entrada A.
<code>linalg.expm</code>	Calcula a matriz exponencial de uma determinada matriz de entrada A.
<code>linalg.logm</code>	Calcula a matriz logarítmica de uma determinada matriz de entrada A.
<code>linalg.det</code>	Calcula o determinante de uma determinada matriz de entrada A.
<code>linalg.solve</code>	Solução de equações lineares fornecidas.

Na função *linalg.solve* cabe a menção de que podemos solucionar sistemas lineares utilizando essa função. Por exemplo, se tivermos o seguinte sistema:

$$\begin{cases} x + 2y = 11 \\ 13x + 15y = 7 \end{cases}$$

podemos utilizar a função *linalg.solve* para descobrir os valores de x e y, utilizando o seguinte código.

```
a = np.array([[1, 2], [13, 15]])
b = np.array([11, 7])
x_y = linalg.solve(a, b)
# resultado; x = -13.72727273; y = 12.36363637
```



XPe

> Capítulo 8



Capítulo 8. Scikit Learn

Ao longo dos capítulos presentes nesta apostila, vimos bibliotecas de grande ajuda em questões de apresentação, manutenção, distribuição e processamento de dados. No entanto, em ciência de dados pode haver a necessidade de validar informações ou construir modelos supervisionados, não supervisionados e até mesmo de auto aprendizado.

Neste capítulo, veremos a biblioteca *Scikit-Learn* que tem como maior intuito disponibilizar algoritmos de inteligência artificial e aprendizado de máquina para uso simples. Veremos como importar modelos de classificação e como selecionar os modelos com base em seus respectivos resultados. Todos os resultados podem ser analisados utilizando métodos disponíveis na biblioteca *Scikit-Learn*.

O que é Scikit-Learn?

Scikit-learn é uma biblioteca criada para construção simples de algoritmos de machine learning, podendo criar algoritmos de classificação, regressão, clusterização, redução de dimensionalidade, seleção de modelos, pré-processamento, entre outros.

Para reduzir a quantidade de algoritmos de machine learning, estaremos apresentando 3 algoritmos com scikit-learn, sendo eles: *KNN (K-Nearest Neighbors)*, *Decision Tree* e *Random Forest*. Ao longo do capítulo iremos detalhar como carregar os modelos e como realizar o treinamento deles para realizar a avaliação dos modelos e então selecionar eles com base em seus resultados. O dataset utilizado será o *breast cancer*.

Antes de realizar qualquer treinamento de modelo, devemos nos atentar ao processamento dos dados, construindo as variáveis X e Y, sendo X os dados característicos e Y as classes de cada dado. Para processar os

dados, podemos utilizar a biblioteca Pandas, e para separar cada conjunto de treino e teste podemos utilizar a função `train_test_split` do próprio scikit-learn. Essa função utiliza como entrada os valores de X e Y para realizar a separação dos conjuntos de treino e teste, podendo ser passado como parâmetro a porcentagem de treino e teste desejada.

Após a divisão dos conjuntos de treino e teste, podemos utilizar uma técnica de normalização de dados, buscando diminuir a possibilidade de *overfitting* dos modelos de classificação.

Para realizar o carregamento dos modelos utilizando scikit-learn, podemos utilizar o seguinte código:

```
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier

x_train, y_train, x_test, y_test = train_test_split(X, Y, train_size=0.7, test_size=0.3)

random_forest = RandomForestClassifier(n_estimators=100, criterion="entropy")
knn = KNeighborsClassifier(n_neighbors=5, method="minkowski", p=2)
decision_tree = DecisionTreeClassifier()
```

Após o carregamento dos modelos, conseguimos detalhar um pouco melhor como cada um deles pode funcionar. A árvore de decisão (*Decision Tree*) é um algoritmo supervisionado que busca criar uma estrutura de árvore em que cada ramificação é uma possibilidade para o dado de entrada. As árvores de decisão são algoritmos bem eficientes para problemas complexos, no entanto, podem cair em *overfitting* com grande facilidade. O algoritmo *KNN* seleciona um valor ímpar para agrupar seus *k*-vizinhos mais próximos e através desse agrupamento indica a classe predominante

perante a este agrupamento, por isso a necessidade do valor ser ímpar, para que não entre em empate. O algoritmo de Floresta Aleatória (Random Forest) busca sanar o problema de *overfitting* da árvore de decisão criando n árvores de decisão e utilizando elas em conjunto para classificar os dados.

Avaliação e Seleção de modelos com Scikit-Learn

Após o treinamento dos modelos citados, podemos realizar predições para verificar a qualidade dos treinamentos perante a nossa base *breast cancer*. Para avaliar os modelos, podemos utilizar métricas de avaliação a fim de nos auxiliar na seleção do modelo que conseguiu se adaptar da melhor maneira aos dados do *breast cancer*.

Para realizar as avaliações, utilizaremos alguns métodos do scikit-learn em conjunto com matplotlib e seaborn. Utilizaremos as métricas: acurácia, precisão, revocação, f1-score, matriz de confusão, curvas ROC e AUC.

Antes de entendermos como funciona cada métrica, precisamos aprender de fato como são as entradas dessas funções. Em algoritmos supervisionados temos os labels de cada dado treinado no modelo e cada dado a ser testado após o treinamento, portanto, podemos realizar um teste de mesa entre a classe prevista pelo modelo treinado e a verdadeira classe do dado. Com este teste podemos encontrar verdadeiros positivos, verdadeiros negativos, falsos positivos e falsos negativos, conforme apresentado na figura 48.

Figura 48 - Comparação entre os previstos e dados reais.

		P R E D I T O	
		👍 POSITIVO	👎 NEGATIVO
R E A L	👍 POSITIVO	✅👍 TP verdadeiro positivo	❌👎 FN falso negativo
	👎 NEGATIVO	❌👍 FP falso positivo	✅👎 TN verdadeiro negativo

Acurácia indica a performance geral do modelo, analisando todas as classificações. O cálculo é feito da seguinte maneira:

$$\text{Acurácia} = (TP + TN) \div (TP + FN + FP + TN)$$

Precisão indica quantas classificações de classes verdadeiras o modelo fez. O cálculo é feito da seguinte maneira:

$$\text{Precisão} = TP \div (TP + FP)$$

Revocação ou sensibilidade indica dentre todas as situações de classe positiva quantas estão corretas. O cálculo é feito da seguinte maneira:

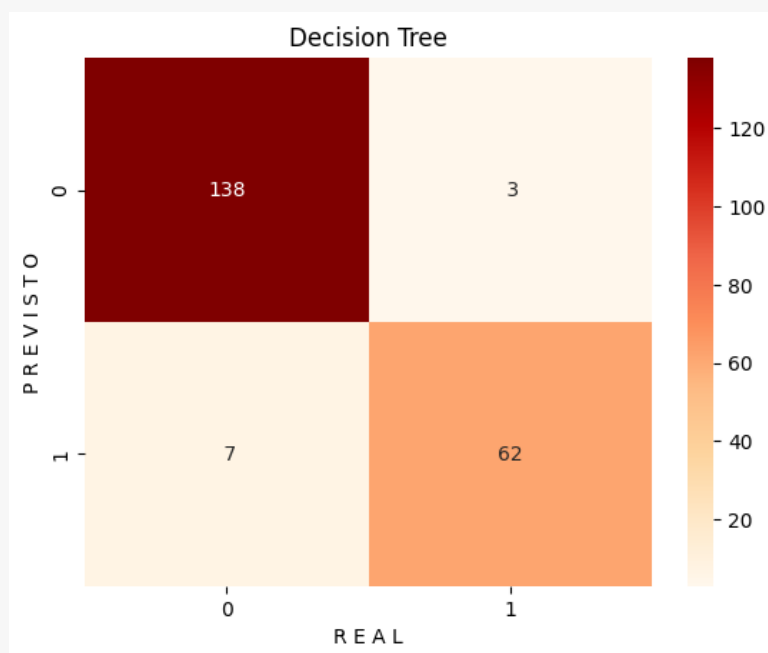
$$\text{Revocação} = TP \div (TP + TN)$$

F1-score é a média harmônica entre precisão e sensibilidade. O cálculo é feito da seguinte maneira:

$$\text{F1-Score} = 2 * \textit{precisão} * \textit{revocação} \div (\textit{precisão} + \textit{revocação})$$

Além das métricas citadas, podemos utilizar outra estrutura de validação para avaliar a performance dos modelos treinados, a matriz de confusão. A Matriz de Confusão é calculada pela função de pesquisa Classificação. Ela exibe a distribuição dos registros em termos de suas classes atuais e de suas classes previstas. Isso indica a qualidade do modelo atual. A figura 49 apresenta um exemplo de matriz de confusão.

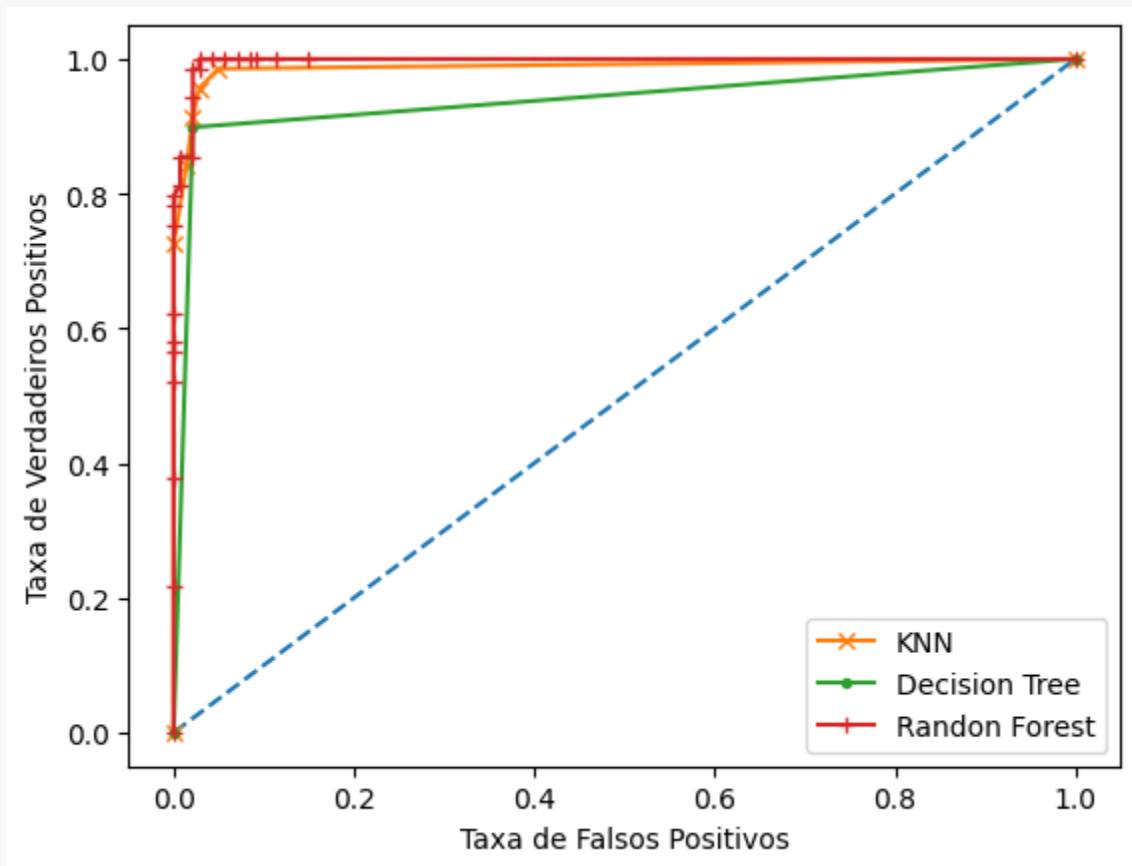
Figura 49 - Matriz de confusão do modelo árvore de decisão. Modelo treinado com dados do dataset *breast cancer*.



Na figura 49 podemos analisar que temos somente duas possíveis classes nas colunas a serem processadas. A matriz de confusão vai indicar onde o modelo está se confundindo no quesito das classes, neste caso, temos apenas as classes benigno e maligno, classe 0 benigno e classe 1 maligno. O modelo conseguiu classificar corretamente 200 amostras de um total de 210, um resultado bem interessante em termos de classificação.

Para aumentar ainda mais o nível de avaliação dos modelos, podemos utilizar as curvas ROC e a área abaixo da curva para medir a qualidade dos modelos. As curvas ROC (*receiver operator characteristic curve*) são uma forma de representar a relação, normalmente antagônica, entre a sensibilidade e a especificidade de um teste diagnóstico quantitativo. A figura 50 apresenta um exemplo de curva ROC utilizando as classificações feitas pelos modelos citados neste capítulo.

Figura 50 - Curvas ROC com as classificações feitas com os modelos KNN, Decision Tree e Random Forest.



A área abaixo da curva pode ser utilizada para medir o tamanho da curva ROC. Juntas, AUC e ROC são talvez o método mais eficiente para seleção de modelos de classificação de dados. Os resultados de AUC dos modelos treinados foram: *Decision Tree* 93.90%, *KNN* 98.80% e *Random Forest* 99.60%. Portanto, analisando estes resultados, podemos dizer que para a classificação dos dados do dataset *breast cancer*, o modelo Random Forest foi o melhor.

Referências

CONCEITOS de mineração de dados. Microsoft Build, 27 de setembro de 2022. Disponível em: <<https://learn.microsoft.com/pt-br/analysis-services/data-mining/data-mining-concepts?view=asallproducts-allversions>>. Acesso em: 25 abr. 2023.

JUPUDI, Lakshmi. Machine learning techniques using python for data analysis in performance evaluation. In: International Journal of Intelligent Systems Technologies and Applications, v. 17, n. 3, 2018.

MAULUD, Dastan; ABDULAZEEZ, Adnan Mohsin. A Review on Linear Regression Comprehensive in Machine Learning. In: Journal of Applied Science and Technology Trends, v. 1, 2020. p. 140-147.

SCIPY. Estruturas UFPR. Disponível em: <<http://www.estruturas.ufpr.br/disciplinas/pos-graduacao/introducao-a-computacao-cientifica-com-python/introducao-python/capitulo-4-scipy/>>. Acesso em: 25 abr. 2023.

SHREYA, Debbata. DIGITAL IMAGE PROCESSING AND RECOGNITION USING PYTHON. In: International Journal of Engineering Applied Sciences and Technology, v. 5, 2021.