Relatório 4 - Prática: Principais Bibliotecas e Ferramentas Python para Aprendizado de Máquina (I)

Igor Carvalho Marchi

Descrição da atividade

Neste card foi apresentado duas bibliotecas, sendo elas a NumPy e Pandas, onde ambas contribuem para ciência de dados e análise numérica, durante as aulas foi apresentado que o Numpy fornece o suporte para arrays multidimensionais e funções matemáticas aplicadas nos arrays. Porém já a outra biblioteca serve para manipulação e análise de dados estruturados como tabelas, por exemplo análise de uma tabela do Excel como foi apresentado durante o card.

NumPy

Aula 20:

Np.array(), onde eu transformo minha lista em array de numpy:

```
np.array(minha_lista)
```

```
array([1, 2, 3])
```

```
minha_matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
np.array(minha_matriz)
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Como apresentado na imagem acima o comando array tem como função gerar arrays(listas) simples onde guarda os números inseridos.

```
np.arange (0, 10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
np.arange (0, 10 , 2)
```

```
array([0, 2, 4, 6, 8])
```

A imagem acima mostra o comando arange onde ele tem a função de gerar um array baseado como se fosse um for, por exemplo a primeira linha fala para ele ir da sequência numérica de 0 a 10, porém como ele não tem número do passo ele entende

que o passo é 1, por isso dentro do array tem do 0 ao 9, já no outro comando mostra o número 2 no passo, ou seja ele vai andando em 2 e até completar a sequência.

Zeros:

```
:  np.zeros(3)

:  array([0., 0., 0.])
```

O comando de zeros é responsável por criar uma lista de zeros, onde é definida a quantidade de zeros.

Ones:

```
[18]: np.ones((3, 3))

[18]: array([[1., 1., 1.],
             [1., 1., 1.],
             [1., 1., 1.]])
```

O comando Ones é responsável por criar uma lista de apenas números 1, na imagem acima é possível perceber que ele criou 3 listas que guardam 3 números 1.

Eye:

```
[20]: np.eye(4)

[20]: array([[1., 0., 0., 0.],
             [0., 1., 0., 0.],
             [0., 0., 1., 0.],
             [0., 0., 0., 1.]])
```

O comando eye tem como finalidade criar uma matriz de identidade, ou seja, ele cria uma matriz onde a diagonal principal tem números 1, porém o resto da matriz tem os números 0.

Linspace:

```
[21]: np.linspace(0, 10, 2)

[21]: array([ 0., 10.])

[22]: np.linspace(0, 10, 3)

[22]: array([ 0.,  5., 10.])
```

No comando linspace o usuário deve determinar uma sequência numérica e a quantidade dos espaçamentos iguais dentro dela, na imagem acima tem dois exemplos com a mesma sequência numérica, porém com a quantidade de espaçamentos diferentes.

Random:

```
[29]: np.random.rand(4)

[29]: array([0.73270853, 0.610071  , 0.2596268 , 0.79361699])

[30]: np.random.randn(4)

[30]: array([-0.53431149, -0.27624699,  2.91712632,  0.34349988])
```

O comando random tem como finalidade gerar um array com números aleatórios, porém quando acrescentamos outros comandos neles podemos determinar um ponto para esse número aleatório deve estar, por exemplo o random.rand são números aleatórios que estão entre 0 e 1, já o random.randn são números aleatórios entre -1 e 0, existe bem mais comando para combinar como o random.

Reshape:

```
array
```

```
array([0.41203844, 0.50879582, 0.80141425, 0.55927561, 0.7016656 ,
       0.06413814, 0.54023816, 0.48383437, 0.82183691, 0.08859371,
       0.58882535, 0.07315406, 0.02243744, 0.87470578, 0.31339313,
       0.61725542, 0.34170806, 0.322724  , 0.61260991, 0.49450597])
```

```
array.reshape(5,4)
```

```
array([[0.41203844, 0.50879582, 0.80141425, 0.55927561],
       [0.7016656 , 0.06413814, 0.54023816, 0.48383437],
       [0.82183691, 0.08859371, 0.58882535, 0.07315406],
       [0.02243744, 0.87470578, 0.31339313, 0.61725542],
       [0.34170806, 0.322724  , 0.61260991, 0.49450597]])
```

```
array.reshape(2,10)
```

```
array([[0.41203844, 0.50879582, 0.80141425, 0.55927561, 0.7016656 ,
        0.06413814, 0.54023816, 0.48383437, 0.82183691, 0.08859371],
       [0.58882535, 0.07315406, 0.02243744, 0.87470578, 0.31339313,
        0.61725542, 0.34170806, 0.322724  , 0.61260991, 0.49450597]])
```

O comando reshape serve para reformatar a lista de arrays onde o primeiro número representa a quantidade de linhas e a segunda é a coluna.

Max e min:

```
array([[0.39265544, 0.1263744 , 0.90788995, 0.69550912, 0.56025822],
       [0.0973678 , 0.91486565, 0.69601876, 0.21176454, 0.26538792],
       [0.22394016, 0.42090031, 0.5791962 , 0.33418668, 0.02450032],
       [0.42048044, 0.8257862 , 0.45775391, 0.08449992, 0.32722693],
       [0.47327297, 0.56286675, 0.19138398, 0.9519195 , 0.04493932]])
```

```
arr.max()
```

```
0.9519195040512641
```

```
arr.min()
```

```
0.024500324129057005
```

O max tem como finalidade achar o maior número do array, porém já o min tem como finalidade achar o menor número do array.

argMax e argMin:

```
array([[0.39265544, 0.1263744 , 0.90788995, 0.69550912, 0.56025822],
       [0.0973678 , 0.91486565, 0.69601876, 0.21176454, 0.26538792],
       [0.22394016, 0.42090031, 0.5791962 , 0.33418668, 0.02450032],
       [0.42048044, 0.8257862 , 0.45775391, 0.08449992, 0.32722693],
       [0.47327297, 0.56286675, 0.19138398, 0.9519195 , 0.04493932]])
```

```
arr.argmax()
```

```
23
```

```
arr.argmin()
```

```
14
```

O argMax é responsável por retornar o índice do maior numero, já o argMin é responsável por retornar o índice do menor número.

Aula 21:

Na aula 21 começamos a aprender sobre como funciona o indexamento e o fatiamento dos arrays.

```
In [2]: arr = np.arange(0, 30, 3)

In [3]: arr

Out[3]: array([ 0,  3,  6,  9, 12, 15, 18, 21, 24, 27])

In [5]: arr[3]

Out[5]: 9
```

A imagem acima é a representação de que quando usamos o comando array[x] pegamos o valor que está dentro da posição que é oferecida pela quantidade x, no exemplo acima é 3, então contando de 0 a 3 o número que está nessa posição é o 9.

```
In [6]: arr[2:5]

Out[6]: array([ 6,  9, 12])

In [7]: arr[:5]

Out[7]: array([ 0,  3,  6,  9, 12])

In [8]: arr[2:]

Out[8]: array([ 6,  9, 12, 15, 18, 21, 24, 27])
```

A imagem acima representa 3 casos de fatiamento, quando tem um número antes dos “:” é a posição que deve começar a contagem, quando é depois é onde ele deve parar, porém quando tem ambos ele vai iniciar do valor de antes e vai parar quando chegar na posição do numero que está depois dos dois pontos.

```
In [9]: arr[2:] = 100

In [11]: arr[2:]

Out[11]: array([100, 100, 100, 100, 100, 100, 100, 100])

In [12]: arr

Out[12]: array([  0,   3, 100, 100, 100, 100, 100, 100, 100, 100])
```

Quando usamos um comando igual array[x:] = y, todos os valores após a posição de x vão ser substituídos por y, como representado na imagem acima, onde todos valores da posição 2 para frente foram trocados por 2.

```
In [15]: arr = np.arange(50).reshape((5, 10))

In [16]: arr

Out[16]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
                [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
                [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
                [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
                [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]])
```

A imagem acima representa os fatiamentos com arrays bidimensionais, onde ele pega um array com 50 posições e divide em 5, onde terão 10 posições cada.

Não dá para criar um array, dar valor de um outro array e alterar o array criado dessa forma, pois o python não faz uma cópia direta, O arr2 é alterado e altera juntamente o arr, se quiser fazer algo parecido necessita usar .copy como na imagem abaixo

```
In [48]: arr2 = arr[:3].copy()

In [49]: arr2

Out[49]: array([[100, 100, 100, 100, 100, 100, 100, 100, 100, 100],
                [100, 100, 100, 100, 100, 100, 100, 100, 100, 100],
                [100, 100, 100, 100, 100, 100, 100, 100, 100, 100]])

In [50]: arr2[:] = 10

In [51]: arr2

Out[51]: array([[10, 10, 10, 10, 10, 10, 10, 10, 10, 10],
                [10, 10, 10, 10, 10, 10, 10, 10, 10, 10],
                [10, 10, 10, 10, 10, 10, 10, 10, 10, 10]])
```

É possível fazer o fatiamento de um array através de comandos booleanos:

```
In [54]: arr > 50

Out[54]: array([[ True,  True,  True,  True,  True,  True,  True,  True,  True,
                  True],
                [ True,  True,  True,  True,  True,  True,  True,  True,  True,
                  True],
                [ True,  True,  True,  True,  True,  True,  True,  True,  True,
                  True],
                [False, False, False, False, False, False, False, False, False,
                 False],
                [False, False, False, False, False, False, False, False, False,
                 False]])

In [55]: bol = arr > 50

In [57]: arr[bol]

Out[57]: array([100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100,
                100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100,
                100, 100, 100, 100])
```

Aula 22:

Na aula 22 estaremos usando operações matemáticas básicas com os arrays, contudo é importante lembrar que para isso os arrays precisam ter o mesmo tamanho.

```
In [3]: arr = np.arange(0, 16)

In [4]: arr

Out[4]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])

In [5]: arr + arr

Out[5]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30])

In [6]: arr - arr

Out[6]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

In [7]: arr * arr

Out[7]: array([  0,   1,   4,   9,  16,  25,  36,  49,  64,  81, 100, 121, 144,
               169, 196, 225])

In [8]: arr / arr

C:\Users\User\AppData\Local\Temp\ipykernel_7352\3001117470.py:1: RuntimeWarning
  arr / arr

Out[8]: array([nan,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
               1.,  1.,  1.])
```

Na imagem acima foi demostrado as operações de array de soma, subtração, multiplicação e divisão.

Além de fazer operações entre arrays é possível fazer operações com números inteiros seja exponencialmente ou entre as operações básicas como na imagem abaixo:

```
In [11]: arr ** 2

Out[11]: array([  0,   1,   4,   9,  16,  25,  36,  49,  64,  81, 100, 121, 144,
                169, 196, 225])

In [12]: arr * 100

Out[12]: array([   0,  100,  200,  300,  400,  500,  600,  700,  800,  900, 1000,
                1100, 1200, 1300, 1400, 1500])

In [13]: arr - 100

Out[13]: array([-100,  -99,  -98,  -97,  -96,  -95,  -94,  -93,  -92,  -91,  -90,
                 -89,  -88,  -87,  -86,  -85])

In [14]: arr * 100

Out[14]: array([   0,  100,  200,  300,  400,  500,  600,  700,  800,  900, 1000,
                1100, 1200, 1300, 1400, 1500])
```

```
In [15]: np.sqrt(arr)

Out[15]: array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
                2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ,
                3.16227766, 3.31662479, 3.46410162, 3.60555128, 3.74165739,
                3.87298335])

In [16]: np.exp(arr)

Out[16]: array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
                5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
                2.98095799e+03, 8.10308393e+03, 2.20264658e+04, 5.98741417e+04,
                1.62754791e+05, 4.42413392e+05, 1.20260428e+06, 3.26901737e+06])

In [20]: np.mean(arr)

Out[20]: 7.5

In [22]: np.std(arr)

Out[22]: 4.6097722286464435

In [23]: np.sin(arr)

Out[23]: array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
                -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849,
                -0.54402111, -0.99999021, -0.53657292,  0.42016704,  0.99060736,
                 0.65028784])
```

- np.sqrt(arr) → Raiz quadrada de cada valor.
- np.exp(arr) → Exponencial (e^x) de cada valor.
- np.mean(arr) → Média dos valores.
- np.std(arr) → Desvio padrão dos valores.
- np.sin(arr) → Seno (em radianos) de cada valor.

Além das operações básicas, a biblioteca NumPy oferece suporte a cálculos matemáticos avançados utilizando arrays, como raiz quadrada, funções exponenciais, desvio padrão e funções trigonométricas. Essas operações são aplicadas de forma vetorizada e otimizada, permitindo cálculos eficientes entre múltiplos arrays. Esse conjunto de funcionalidades torna o NumPy um recurso essencial para aplicações em estatística, simulações numéricas e machine learning.

Pandas

Aula 28:

DATAFRAME: DataFrame é o objeto principal da biblioteca Pandas, ele é basicamente um conjunto de series, eles podem ser analisados quase que em forma de tabela do excel, porém ele só aparece formatado assim em algumas IDES que é o caso do Jupyter:

```
In [1]: import pandas as pd
        import numpy as np

In [2]: np.random.seed(101)

In [4]: df = pd.DataFrame(np.random.randn (5, 4), index = 'A B C D E'.split(), columns = 'W X Y Z'.split())

In [5]: df
```

Out[5]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	-0.319318	-0.848077	0.605965
C	-2.018168	0.740122	0.528813	-0.589001
D	0.188695	-0.758872	-0.933237	0.955057
E	0.190794	1.978757	2.605967	0.683509

Na imagem acima utilizando numpy foi pego seeds random, para tebela e usando a biblioteca pandas foi utilizada para uma tabela onde são 5 linhas e 4 colunas.

Na imagem abaixo podemos ver que podemos pegar dados de apenas uma coluna ou linha, no caso peguei os dados da coluna 'W'. Além disso através dos comandos é possível verificar os tipos da coluna e da tabela.

```
In [7]: df['W']

Out[7]: A    2.706850
        B    0.651118
        C   -2.018168
        D    0.188695
        E    0.190794
        Name: W, dtype: float64

In [10]: type(df['W'])

Out[10]: pandas.core.series.Series

In [12]: type (df)

Out[12]: pandas.core.frame.DataFrame
```

Também posso acrescentar uma nova coluna como na imagem abaixo, onde crio a coluna “NEW” e passo os dados somados das colunas W e X.

```
In [15]: df ['new'] = df['W'] + df['X']

In [16]: df
```

Out[16]:

	W	X	Y	Z	new
A	2.706850	0.628133	0.907969	0.503826	3.334983
B	0.651118	-0.319318	-0.848077	0.605965	0.331800
C	-2.018168	0.740122	0.528813	-0.589001	-1.278046
D	0.188695	-0.758872	-0.933237	0.955057	-0.570177
E	0.190794	1.978757	2.605967	0.683509	2.169552

Além de adicionar eu também posso retirar uma coluna usando o comando drop representado na imagem abaixo:

```
In [22]: df.drop('new', axis=1, inplace=True)

In [23]: df
```

Out[23]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	-0.319318	-0.848077	0.605965
C	-2.018168	0.740122	0.528813	-0.589001
D	0.188695	-0.758872	-0.933237	0.955057
E	0.190794	1.978757	2.605967	0.683509

Utilizando o loc eu posso localizar de forma mais precisa os dados que eu desejo, como na imagem abaixo onde no primeiro código eu procuro o valor da linha A e coluna W, já na segunda eu procuro pelas linha A e B nas colunas X, Y e Z.

```
In [26]: df.loc['A','W']

Out[26]: 2.706849839399938

In [27]: df.loc[['A', 'B'], ['X', 'Y', 'Z']]
```

Out[27]:

	X	Y	Z
A	0.628133	0.907969	0.503826
B	-0.319318	-0.848077	0.605965

Outro código é o iloc, onde le acessa dados por índice de posição (números inteiros):

```
In [30]: df.iloc[1:4, 2:]
```

Out[30]:

	Y	Z
B	-0.848077	0.605965
C	0.528813	-0.589001
D	-0.933237	0.955057

Aula 29:

Na imagem abaixo é mostrado que podemos verificar usando booleanas se é verdadeiro ou falso, onde a condição é 0, então os lugares que estão com true são maiores que 0 e os que estão falsos são os que são iguais ou menores que zero.

```
In [7]: df
```

Out[7]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	-0.319318	-0.848077	0.605965
C	-2.018168	0.740122	0.528813	-0.589001
D	0.188695	-0.758872	-0.933237	0.955057
E	0.190794	1.978757	2.605967	0.683509

```
In [8]: df > 0
```

Out[8]:

	W	X	Y	Z
A	True	True	True	True
B	True	False	False	True
C	False	True	True	False
D	True	False	False	True
E	True	True	True	True

```
In [9]: bol = df > 0

In [11]: df[bol]
```

Out[11]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	NaN	NaN	0.605965
C	NaN	0.740122	0.528813	NaN
D	0.188695	NaN	NaN	0.955057
E	0.190794	1.978757	2.605967	0.683509

Quando uso essa condição e salvo ela, ela irá colocar no lugar das posições falsas o NaN para mostrar que não existe dentro da condição.

Na imagem abaixo mostra um caso ele determina a condição para a coluna 'W' todos os elementos serem acima de 0, assim ficando sem a linha 'C':

```
In [11]: df[bol]
```

Out[11]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	NaN	NaN	0.605965
C	NaN	0.740122	0.528813	NaN
D	0.188695	NaN	NaN	0.955057
E	0.190794	1.978757	2.605967	0.683509

```
In [12]: df[df['W'] > 0]
```

Out[12]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	-0.319318	-0.848077	0.605965
D	0.188695	-0.758872	-0.933237	0.955057
E	0.190794	1.978757	2.605967	0.683509

Também podem ser utilizado condições para separar dados da tabela como mostrado na imagem abaixo:

```
In [21]: df[(df['W'] > 0) & (df['Y'] > 1)]
```

Out[21]:

	W	X	Y	Z
E	0.190794	1.978757	2.605967	0.683509

```
In [23]: df[(df['W'] > 0) | (df['Y'] > 1)]
```

Out[23]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	-0.319318	-0.848077	0.605965
D	0.188695	-0.758872	-0.933237	0.955057
E	0.190794	1.978757	2.605967	0.683509

Com essa condição é possível selecionar as linhas utilizando argumentos lógicos.

```
In [16]: df[df['W'] > 0]['Y']

Out[16]: A    0.907969
         B   -0.848077
         D   -0.933237
         E    2.605967
         Name: Y, dtype: float64

In [17]: bol = df['W'] > 0
         df2 = df[bol]
         df2['Y']

Out[17]: A    0.907969
         B   -0.848077
         D   -0.933237
         E    2.605967
         Name: Y, dtype: float64
```

Na imagem acima mostra uma filtragem da mesma maneira, mas imprimir outra coluna com o condicional feito.

Com um método é possível trocar os índices para se tornarem colunas, e transformando o índice no padrão numpy que vai de 0 até N, da seguinte forma:

```
In [25]: df.reset_index(inplace=True)

In [26]: df
```

Out[26]:

	index	W	X	Y	Z
0	A	2.706850	0.628133	0.907969	0.503826
1	B	0.651118	-0.319318	-0.848077	0.605965
2	C	-2.018168	0.740122	0.528813	-0.589001
3	D	0.188695	-0.758872	-0.933237	0.955057
4	E	0.190794	1.978757	2.605967	0.683509

Porém ele não fica alterado permanente, é necessário passar como argumento do método o (inplace = True).

```
In [27]: col = 'RS RJ SP AM SC'.split()

In [29]: col

Out[29]: ['RS', 'RJ', 'SP', 'AM', 'SC']

In [30]: df['Estado'] = col

In [31]: df
```

Out[31]:

	index	W	X	Y	Z	Estado
0	A	2.706850	0.628133	0.907969	0.503826	RS
1	B	0.651118	-0.319318	-0.848077	0.605965	RJ
2	C	-2.018168	0.740122	0.528813	-0.589001	SP
3	D	0.188695	-0.758872	-0.933237	0.955057	AM
4	E	0.190794	1.978757	2.605967	0.683509	SC

A imagem acima mostra como criar uma coluna de forma especifica.

```
In [34]: df.set_index('Estado', inplace=True)
```

Contudo para ficar de forma permanente na tabela se deve colocar o inplace=True.

Aula 30:

Índices multinível permitem que um DataFrame do Pandas tenha mais de um índice por linha ou coluna, criando uma estrutura hierárquica**.**

Isso facilita:

- Agrupamentos por categorias (ex: por grupo e subgrupo)
- Consultas organizadas (ex: acessar todos os dados do grupo G1)
- Operações complexas (como agregações por nível)
- Visualizações mais estruturadas

```
In [2]: # Niveis de Índice
        outside = ['G1','G1','G1','G2','G2','G2']
        inside = [1,2,3,1,2,3]
        hier_index = list(zip(outside,inside))
        hier_index = pd.MultiIndex.from_tuples(hier_index)

In [3]: outside

Out[3]: ['G1', 'G1', 'G1', 'G2', 'G2', 'G2']

In [4]: inside

Out[4]: [1, 2, 3, 1, 2, 3]

In [5]: hier_index = list(zip(outside,inside))

In [6]: hier_index

Out[6]: [('G1', 1), ('G1', 2), ('G1', 3), ('G2', 1), ('G2', 2), ('G2', 3)]
```

Aqui neste código foi criado um MultiIndex, associando 2 listas.

```
In [7]: hier_index = pd.MultiIndex.from_tuples(hier_index)

In [8]: hier_index

Out[8]: MultiIndex([('G1', 1),
                    ('G1', 2),
                    ('G1', 3),
                    ('G2', 1),
                    ('G2', 2),
                    ('G2', 3)],
                   )

In [9]: df = pd.DataFrame(np.random.randn(6,2), index = hier_index, columns= ['A', 'B'])

In [10]: df
```

Out[10]:

		A	B
G1	1	0.418425	-2.291348
	2	0.891681	-1.655216
	3	0.450355	0.765987
G2	1	0.421281	0.486147
	2	0.309411	0.961524
	3	1.827803	1.249382

Assim foi feita a criação do dataframe com dados aleatórios, índice hierarquizado e 2 colunas.

Com o comando loc podemos pegar dados mais específicos, assim como nas aulas apresentadas, por exemplo:

```
In [14]: df.loc['G1'].loc[1]

Out[14]: A    0.418425
         B   -2.291348
         Name: 1, dtype: float64
```

Nessa imagem acima é localizado os dados do G1 na linha 1 dele.

É possível também para facilitar o entendimento do DataFrame é nomear cada índice para uma melhor compreensão dos dados e uma manipulação menos complicada, exemplo:

```
In [16]: df.index.names = ['Grupo', 'Numero']

In [17]: df
```

Out[17]:

		A	B
Grupo	Numero		
G1	1	0.418425	-2.291348
	2	0.891681	-1.655216
	3	0.450355	0.765987
G2	1	0.421281	0.486147
	2	0.309411	0.961524
	3	1.827803	1.249382

Xs() é muito útil para navegar em dados hierárquicos, no primeiro código da imagem abaixo acessa todas as linhas do grupo 'G1' no nível superior (nível 0 do índice) e mostra os valores para o subnível 'Número' dentro de 'G1'. Já no segundo código acessa todas as linhas onde o nível 'Numero' é 1, independente do grupo, assim filtrando o segundo nível do índice.

```
In [19]: df.xs('G1')
```

Out[19]:

Numero	A	B
1	0.418425	-2.291348
2	0.891681	-1.655216
3	0.450355	0.765987

```
In [21]: df.xs(1, level='Numero')
```

Out[21]:

Grupo	A	B
G1	0.418425	-2.291348
G2	0.421281	0.486147

Aula 31:

Nesta aula, aprendemos a lidar com dados ausentes em DataFrames. No entanto, o tratamento adequado depende do tipo e da origem dos dados, por isso é essencial conhecer bem as informações com as quais se está trabalhando.

Dropna

```
d = {'A':[1,2,np.nan],'B':[5,np.nan,np.nan],'C':[1,2,3]}
```

```
df  = pd.DataFrame(d)
```

```
df
```

	A	B	C
0	1.0	5.0	1
1	2.0	NaN	2
2	NaN	NaN	3

```
df.dropna()
```

	A	B	C
0	1.0	5.0	1

O comando dropna removeu as linhas que tinham os valores NaN(vazios).

Ffill e Bfill:

```
df
```

	A	B	C
0	1.0	5.0	1
1	2.0	NaN	2
2	NaN	5.0	3

```
df.bfill()
```

	A	B	C
0	1.0	5.0	1
1	2.0	5.0	2
2	NaN	5.0	3

```
df
```

	A	B	C
0	1.0	5.0	1
1	2.0	NaN	2
2	NaN	NaN	3

```
df.ffill()
```

	A	B	C
0	1.0	5.0	1
1	2.0	5.0	2
2	2.0	5.0	3

Eles são os novos comandos do parâmetro fillna() que não está mais funcionando, o Preenche NaN com o valor seguinte.

Aula 32:

O groupBy consiste em agrupar mesmos elementos em uma determinada coluna e realizar uma operação nas demais colunas.

In [1]:
```
import pandas as pd
# Cria um DataFrame
data = {'Empresa':['GOOG','GOOG','MSFT','MSFT','FB','FB'],
        'Nome':['Sam','Charlie','Amy','Vanessa','Carl','Sarah'],
        'Venda':[200,120,340,124,243,350]}
```

In [2]:
```
df = pd.DataFrame(data)
```

In [3]:
```
df
```

Out[3]:

	Empresa	Nome	Venda
0	GOOG	Sam	200
1	GOOG	Charlie	120
2	MSFT	Amy	340
3	MSFT	Vanessa	124
4	FB	Carl	243
5	FB	Sarah	350

GroupBy:

[3]:

	Empresa	Nome	Venda
0	GOOG	Sam	200
1	GOOG	Charlie	120
2	MSFT	Amy	340
3	MSFT	Vanessa	124
4	FB	Carl	243
5	FB	Sarah	350

```
[5]: group = df.groupby('Empresa')
```

```
[6]: group
```

```
[6]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x0000018862E90BC0>
```

```
[16]: group.sum()
```

[16]:

Empresa	Nome	Venda
FB	CarlSarah	593
GOOG	SamCharlie	320
MSFT	AmyVanessa	464

```
In [21]: group.count()
```

Out[21]:

Empresa	Nome	Venda
FB	2	2
GOOG	2	2
MSFT	2	2

```
In [18]: group.describe()
```

Out[18]:

	Venda							
	count	mean	std	min	25%	50%	75%	max
Empresa								
FB	2.0	296.5	75.660426	243.0	269.75	296.5	323.25	350.0
GOOG	2.0	160.0	56.568542	120.0	140.00	160.0	180.00	200.0
MSFT	2.0	232.0	152.735065	124.0	178.00	232.0	286.00	340.0

O groupBy é uma função usada para agrupar elementos de uma coleção (como uma lista ou array) com base em alguma chave de agrupamento. Na imagem acima ela agrupa as linhas do DataFrame de acordo com os valores únicos da coluna 'Empresa'.

Dentro do groupBy podem ser usados:

- sum(): Calcula a soma total dos valores numéricos de cada grupo.
- count(): Conta a quantidade de registros em cada grupo, sem somar valores.
- describe(): Exibe estatísticas descritivas dos dados, como média, mínimo, máximo e desvio padrão.

  Podemos usar os recursos de forma filtrada, onde na imagem usamos o método sum() apenas na linha da “Amy”.

```
']: group.sum().loc['Amy']

']: Empresa    MSFT
    Venda       340
    Name: Amy, dtype: object
```

Aula 33:

Concatenar DataFrames significa empilhar ou alinhar vários DataFrames, um após o outro, desde que eles possuam a mesma estrutura.

Mesclar é combinar DataFrames com base em colunas ou índices que compartilham valores em comum, de forma semelhante ao funcionamento de um join em bancos de dados relacionais.

Juntar envolve unir DataFrames com base nos seus índices, mesmo que os índices não sejam exatamente iguais, o método tenta combinar as informações, preenchendo com valores ausentes onde não houver correspondência.

Criação dos dataframes:

```
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']},
                    index=[0, 1, 2, 3])
```

```
df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                    'B': ['B4', 'B5', 'B6', 'B7'],
                    'C': ['C4', 'C5', 'C6', 'C7'],
                    'D': ['D4', 'D5', 'D6', 'D7']},
                     index=[4, 5, 6, 7])
```

```
df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
                    'B': ['B8', 'B9', 'B10', 'B11'],
                    'C': ['C8', 'C9', 'C10', 'C11'],
                    'D': ['D8', 'D9', 'D10', 'D11']},
                    index=[8, 9, 10, 11])
```

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

```
df2
```

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

```
df3
```

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Concatenar:

Pegamos os DataFrames acima e iremos concatenar e tornas os 3 em 1:

```
[8]: pd.concat([df1, df2, df3])
```

[8]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

A concatenação desses 3 dataframes com o eixo padrão 0 Criará um dataframe com as dimensões 12x4.

Contudo, quando os índices não correspondem da mesma maneira, criando esse dataframe 12x12:

```
pd.concat([df1,df2,df3],axis=1)
```

	A	B	C	D	A	B	C	D	A	B	C	D
0	A0	B0	C0	D0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	A2	B2	C2	D2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	A3	B3	C3	D3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	A4	B4	C4	D4	NaN	NaN	NaN	NaN
5	NaN	NaN	NaN	NaN	A5	B5	C5	D5	NaN	NaN	NaN	NaN
6	NaN	NaN	NaN	NaN	A6	B6	C6	D6	NaN	NaN	NaN	NaN
7	NaN	NaN	NaN	NaN	A7	B7	C7	D7	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A8	B8	C8	D8
9	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A9	B9	C9	D9
10	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A10	B10	C10	D10
11	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A11	B11	C11	D11

Mesclar:

Criando 2 novos DataFrames, esses serão os novos dados para usar no método mesclar:

```
esquerda
```

	key	A	B
0	K0	A0	B0
1	K1	A1	B1
2	K2	A2	B2
3	K3	A3	B3

```
direita
```

	key	C	D
0	K0	C0	D0
1	K1	C1	D1
2	K2	C2	D2
3	K3	C3	D3

```
pd.merge(esquerda,direita,how='inner',on='key')
```

	key	A	B	C	D
0	K0	A0	B0	C0	D0
1	K1	A1	B1	C1	D1
2	K2	A2	B2	C2	D2
3	K3	A3	B3	C3	D3

Usando o merge ele tem a finalidade de reunir as tabelas mantendo a coluna key como o elemento em comum entre eles.

Usando outros DataFrames

```
esquerda = pd.DataFrame({'key1': ['K0', 'K0', 'K1', 'K2'],
                         'key2': ['K0', 'K1', 'K0', 'K1'],
                            'A': ['A0', 'A1', 'A2', 'A3'],
                            'B': ['B0', 'B1', 'B2', 'B3']})

direita = pd.DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],
                                'key2': ['K0', 'K0', 'K0', 'K0'],
                                   'C': ['C0', 'C1', 'C2', 'C3'],
                                   'D': ['D0', 'D1', 'D2', 'D3']})
```

```
pd.merge(esquerda, direita, on=['key1', 'key2'])
```

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2

```
pd.merge(esquerda, direita, how='outer', on=['key1', 'key2'])
```

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K0	K1	A1	B1	NaN	NaN
2	K1	K0	A2	B2	C1	D1
3	K1	K0	A2	B2	C2	D2
4	K2	K0	NaN	NaN	C3	D3
5	K2	K1	A3	B3	NaN	NaN

No primeiro merge ele é usado para manter apenas as linhas que possuem correspondência exata nas colunas key1 e key2 em ambos os DataFrames. Já no segundo ele mantém todas as combinações de key1 e key2 dos dois DataFrames, mesmo que não tenham correspondência no outro.

]

Usando os comandos left e right podemos manipular os dados ausentes.

- Left - DataFrame da esquerda dados ausentes na direita
- Right - DataFrame da direita dados ausentes na esquerda

```
pd.merge(esquerda, direita, how='right', on=['key1', 'key2'])
```

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2
3	K2	K0	NaN	NaN	C3	D3

```
pd.merge(esquerda, direita, how='left', on=['key1', 'key2'])
```

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K0	K1	A1	B1	NaN	NaN
2	K1	K0	A2	B2	C1	D1
3	K1	K0	A2	B2	C2	D2
4	K2	K1	A3	B3	NaN	NaN

Juntar:

```
esquerda = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                         'B': ['B0', 'B1', 'B2']},
                         index=['K0', 'K1', 'K2'])

direita = pd.DataFrame({'C': ['C0', 'C2', 'C3'],
                        'D': ['D0', 'D2', 'D3']},
                         index=['K0', 'K2', 'K3'])
```

```
esquerda
```

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

```
direita
```

	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

```
esquerda.join(direita)
```

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2

Quando usamos o “juntar” é para colocar os valores do dataframe ‘direita’ sempre que o índice dele for igual ao índice do dataframe ‘esquerda’. Como no da direita não tem o K1 ele fica NaN e devido a esquerda não ter o K3 então nem aparece no juntar.

Outro exemplo é quando acrescentamos a palavra “outer”, onde ele pega todos os índices existentes dos dataframes e os preenche com valores não existentes caso não exista.

```
esquerda.join(direita, how='outer')
```

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2
K3	NaN	NaN	C3	D3

Aula 34

Na aula 34 é mostrado operações dentro da própria biblioteca dos pandas. Onde o método de seleção de dados únicos retorna os valores únicos que não se repetem dentro de uma coluna específica.

```
import pandas as pd
df = pd.DataFrame({'col1':[1,2,3,4],'col2':[444,555,666,444],'col3':['abc','def','ghi','xyz']})
df.head()
```

	col1	col2	col3
0	1	444	abc
1	2	555	def
2	3	666	ghi
3	4	444	xyz

```
df['col2'].unique()
```

```
array([444, 555, 666], dtype=int64)
```

Também pode se obter essa lista utilizando o numpy, exemplo:

```
import numpy as np
```

```
np.unique(df['col2'])
```

```
array([444, 555, 666], dtype=int64)
```

Usando o nunique() podemos saber o tamanho da lista:

```
df['col2'].nunique()
```

```
3
```

Usando o value_counts() podemos saber a quantidade de vezes que um número da lista apareceu, no caso [e especificado na coluna 2:

```
df['col2'].value_counts()
```

```
col2
444    2
555    1
666    1
Name: count, dtype: int64
```

Como na imagem abaixo é possível filtrar usando parâmetro da lista, seja numero da tabela ou até por valor.

```
df[df['col1'] > 2]
```

	col1	col2	col3
2	3	666	ghi
3	4	444	xyz

```
df[(df['col1'] > 2) & (df['col2'] == 444)]
```

	col1	col2	col3
3	4	444	xyz

```
def vezes2(x):
    return x*2
```

```
df.sum()
```

```
col1              10
col2            2109
col3    abcdefghixyz
dtype: object
```

```
df['col1'].sum()
```

```
10
```

```
df['col1'].apply(vezes2)
```

```
0    2
1    4
2    6
3    8
Name: col1, dtype: int64
```

A função retorna o dobro dos valores, o sum realiza a soma dos valores de cada coluna 1 + 4 + 2 + 3 = 10. Já no apply ele aplica a função vezes2 em cada elemento da coluna col1, ou seja, multiplica cada valor.

Caso queira poupar código e não queira utilizar a função vezes2 pode ser trocado pelo lambda:

```
df['col1'].apply(lambda x: x*x)
```

```
0     1
1     4
2     9
3    16
Name: col1, dtype: int64
```

A função len retorna o comprimento (número de caracteres) de cada string na coluna 'col3':

```
df['col3'].apply(len)
```

```
0    3
1    3
2    3
3    3
Name: col3, dtype: int64
```

Para deletar uma coluna basta usar o comando del:

```
del df['col2']
```

```
df
```

	col1	col3
0	1	abc
1	2	def
2	3	ghi
3	4	xyz

Para visualizar quais colunas ou índices estão presentes no dataframe, exemplo:

```
df.columns
```

```
Index(['col1', 'col3'], dtype='object')
```

```
df.index
```

```
RangeIndex(start=0, stop=4, step=1)
```

Conseguimos ordenar os valores das colunas, como no exemplo abaixo:

```
df.sort_values(by='col2')
```

	col1	col2	col3
0	1	444	abc
3	4	444	xyz
1	2	555	def
2	3	666	ghi

Caso queira que alteração fique salva é necessário usar inplace=True:

```
df.sort_values(by='col2', inplace=True)
```

```
df
```

	col1	col2	col3
0	1	444	abc
3	4	444	xyz
1	2	555	def
2	3	666	ghi

Outro método é a verificação de elementos nulos dentro do dataframe:

```
df.isnull()
```

	col1	col2	col3
0	False	False	False
3	False	False	False
1	False	False	False
2	False	False	False

Caso houvesse NaN, seria possivl usar o dropna para remover linhas que contêm valores nulos (NaN) em qualquer coluna do DataFrame.

```
df.dropna()
```

	col1	col2	col3
0	1	444	abc
3	4	444	xyz
1	2	555	def
2	3	666	ghi

Uma alternativa para tornar os dados mais legíveis é transformar valores iguais de uma coluna em níveis de índice, criando uma estrutura de índice multinível, como no exemplo:

```
data = {'A':['foo','foo','foo','bar','bar','bar'],
     'B':['one','one','two','two','one','one'],
       'C':['x','y','x','y','x','y'],
       'D':[1,3,2,5,4,1]}

df = pd.DataFrame(data)
```

```
df
```

	A	B	C	D
0	foo	one	x	1
1	foo	one	y	3
2	foo	two	x	2
3	bar	two	y	5
4	bar	one	x	4
5	bar	one	y	1

```
df.pivot_table(values='D',index=['A', 'B'],columns=['C'])
```

		C	x	y
A	**B**			
bar	**one**	4.0	1.0
	two	NaN	5.0
foo	**one**	1.0	3.0
	two	2.0	NaN

Aqui ele transforma os valores em comum facilitando a visualização da tabela.

Aula 35:

Na aula 35 é ensinado como que o pandas pode ser útil na entrada e na saída de dados facilitando muito.

```
df = pd.read_csv('exemplo.csv', sep = ',')
```

```
df
```

	Unnamed: 0	a	b	c	d
0	0	0	1	2	3
1	1	4	5	6	7
2	2	8	9	10	11
3	3	12	13	14	15

```
df = df + 1
```

```
df
```

	Unnamed: 0	a	b	c	d
0	1	1	2	3	4
1	2	5	6	7	8
2	3	9	10	11	12
3	4	13	14	15	16

```
df.to_csv("exemplo.csv", sep = ';', decimal = ',')
```

Na imagem acima, foi importado um arquivo chamado ‘exemplo’ e criei um dataframe com os dados dentro dele, após isso somei um a todos os dados do dataframe e foi exportado como .csv e transformando sua separação em (;) como separador de colunas.

```
df = pd.read_excel('Exemplo_Excel.xlsx', sheet_name= 'Sheet1')
```

```
df
```

	Unnamed: 0	a	b	c	d
0	0	0	1	2	3
1	1	4	5	6	7
2	2	8	9	10	11
3	3	12	13	14	15

No exemplo acima, foi feito o carregamento de uma planilha Excel chamada "Exemplo_Excel.xlsx", especificamente da aba "Sheet1", usando o comando pd.read_excel(). O conteúdo foi armazenado em um DataFrame, exibindo os dados organizados por colunas.

```
df.to_excel("Exemplo_Excel.xlsx", sheet_name= 'Sheet1')
```

Depois é salva o DataFrame df em um arquivo Excel chamado Exemplo_Excel.xlsx, na planilha chamada Sheet1.

```
df.to_excel("Exemplo_Excel.xlsx", sheet_name= 'Sheet1')
```

```
df = pd.read_html('http://www.fdic.gov/bank/individual/failed/banklist.html')
```

```
df[0]
```

	Bank Name	City	State	Cert	Acquiring Institution	Closing Date	Fund Sort ascending
0	Pulaski Savings Bank	Chicago	Illinois	28611	Millennium Bank	January 17, 2025	10548
1	The First National Bank of Lindsay	Lindsay	Oklahoma	4134	First Bank & Trust Co., Duncan, OK	October 18, 2024	10547
2	Republic First Bank dba Republic Bank	Philadelphia	Pennsylvania	27332	Fulton Bank, National Association	April 26, 2024	10546
3	Citizens Bank	Sac City	Iowa	8758	Iowa Trust & Savings Bank	November 3, 2023	10545
4	Heartland Tri-State Bank	Elkhart	Kansas	25851	Dream First Bank, N.A.	July 28, 2023	10544
5	First Republic Bank	San Francisco	California	59017	JPMorgan Chase Bank, N.A.	May 1, 2023	10543
6	Signature Bank	New York	New York	57053	Flagstar Bank, N.A.	March 12, 2023	10540
7	Silicon Valley Bank	Santa Clara	California	24735	First Citizens Bank & Trust Company	March 10, 2023	10539
8	Almena State Bank	Almena	Kansas	15426	Equity Bank	October 23, 2020	10538
9	First City Bank of Florida	Fort Walton Beach	Florida	16748	United Fidelity Bank, fsb	October 16, 2020	10537

Na imagem acima o pandas permite acessar dados através de um link html, ele retorna uma lista, porém dá para fazer um tratamento de dados com mais facilidade.

Conclusões

A prática neste relatório proporcionou uma introdução aplicada ao uso das bibliotecas ”Python NumPy e Pandas”, essenciais na análise de dados e aprendizado de máquina. Foram exploradas funções como criação e manipulação de arrays, operações matemáticas e estatísticas com NumPy, além do uso de DataFrames no Pandas para indexação, filtragem, tratamento de dados, agrupamentos e importação/exportação. Essa experiência contribuiu para uma base sólida no uso dessas ferramentas em projetos de ciência de dados.

Referencias

Introdução ao NumPy

Disponível em: http://www.opl.ufc.br/post/numpy/Tutorial_numpy.pdf

Acesso em: 15 jul. 2025.

Introdução ao Pandas

Disponível em: https://medium.com/tech-grupozap/introdu%C3%A7%C3%A3o-abiblioteca-pandas-89fa8ed4fa38

Acesso em: 15 jul. 2025.