Curso: Machine Learning Básico com Python

Aula 02 – Bibliotecas PYTHON para Inteligência Artificial

Roteiro 03 – Bibliotecas para IA

1. NumPy

O NumPy é o pacote fundamental para computação científica em Python. É uma biblioteca Python que fornece um objeto de matriz multidimensional, vários objetos derivados e uma variedade de rotinas para operações rápidas em matrizes, incluindo matemática, lógica, manipulação de forma, classificação, seleção, E / S , álgebra linear básica, operações estatísticas básicas, simulação aleatória entre outros.

No núcleo do pacote NumPy, está o objeto ndarray (N-dimensional array) que encapsula matrizes multidimensionais de tipos de dados homogêneos, com muitas operações sendo realizadas em código compilado para desempenho. Existem várias diferenças importantes entre as matrizes NumPy e as sequências Python padrão:

- Os arrays NumPy têm um tamanho fixo na criação, ao contrário das listas Python (que podem crescer dinamicamente). Alterar o tamanho de um ndarray criará um novo array e excluirá o original.

- Os elementos em uma matriz NumPy devem ser todos do mesmo tipo de dados e, portanto, terão o mesmo tamanho na memória. A exceção: pode-se ter arrays de objetos (Python, incluindo NumPy), permitindo assim arrays de elementos de tamanhos diferentes.

- Os arrays NumPy facilitam operações matemáticas avançadas e outros tipos de operações em um grande número de dados. Normalmente, essas operações são executadas de forma mais eficiente e com menos código do que é possível usando as sequências integradas do Python.

Uma crescente abundância de pacotes científicos e matemáticos baseados em Python está usando matrizes NumPy. Embora eles normalmente suportem entrada de sequência Python, eles convertem essa entrada em matrizes NumPy antes do processamento e frequentemente geram matrizes NumPy.

O uso do NumPy em grandes bases de dados como as que usaremos é imprescindível. Para usar o NumPy no Colab basta fazer o “import” porque ele já está instalado:

```
import numpy as np
```

Note que ao importá-lo demos um apelido a ele de “np”. Este recurso é muito utilizado porque diminui muito a carga de digitação.

Para criar um array simples usando NumPy basta usar o método “array” do NumPy colocando como parâmetro a lista de números que farão parte do array. Veja o exemplo a seguir:

```
a = np.array([12,34,26,18,10])
# cria uma matriz unidimensional
print(a)
print(a.dtype)
```

```
[12 34 26 18 10]
int64
```

Note que pedimos para mostrar o tipo dos elementos do array com o “dtype”. O padrão para inteiros é o inteiro de 64bits e para float 64bits. Se você quiser criar o array com um tipo específico pode informar ao cria-lo. Veja o exemplo:

```
a1 = np.array([1, 2, 3], dtype = np.float64)
# cria o array como float de 64 bits
print(a1)
print(a1.dtype)
a2 = np.array([1, 2, 3], dtype = np.int32)
# cria o array como float de 64 bits
print(a2)
print(a2.dtype)
```

```
[1. 2. 3.]
float64
[1 2 3]
int32
```

Se você quiser você pode também mudar o tipo do array. Basta usar o “astype”. Veja como no exemplo a seguir:

```
a3 = np.array([1.4, 3.6, -5.1, 9.42, 4.999999])
# Podemos transformar tipos de dados de arrays
print(a3)
print(a3.dtype)
# Mostra array([1.4, 3.6, -5.1, 9.42, 4.999999])
a4 = a3.astype(np.int32)
# quando transformamos de float para int os valores são truncados
print(a4)
print(a4.dtype)
# Mostra array([ 1,  3, -5,  9,  4])
a5 = np.array([1, 2, 3, 4])
print(a5)
print(a5.dtype)
a6 = a5.astype(float)
# podemos fazer o inverso também.
print(a6)
print(a6.dtype)
# Mostra array([1., 2., 3., 4.])
```

```
[ 1.4       3.6      -5.1       9.42      4.999999]
float64
[ 1  3 -5  9  4]
int32
[1 2 3 4]
int64
[1. 2. 3. 4.]
float64
```

Você não precisa decorar todos os tipos existentes em NumPy, mas é bom saber que eles existem. Eles estão disponíveis na própria documentação oficial do NumPy neste link:

https://numpy.org/devdocs/user/basics.types.html

Em NumPy também é possível trabalhar com mais de uma dimensão no array. Para criar um array multidimensionais basta separar as diversas dimensões por parênteses conforme o exemplo a seguir:

```
b = np.array([[7,2,23],[12,27,4],[5,34,23]])
# cria um matriz bidimensional
print(b)
```

```
[[ 7  2 23]
 [12 27  4]
 [ 5 34 23]]
```

O NymPy tem uma função para criar arrays vazios já tipificados. O método para tal tarefa é o método “empty”. Veja o exemplo a seguir:

```
import numpy as np
c = np.empty([3,2], dtype = int)
print(c)
```

```
[[27751728        0]
 [       0        0]
 [       0        0]]
```

Os elementos acima mostram valores aleatórios, pois não são inicializados.

Muitas vezes precisamos criar arrays enormes com valores zerados. O método “zeros” facilita esta tarefa. Veja o exemplo a seguir:

```
d = np.zeros([4,3])
# cria uma matriz 4x3 com valores zero
print(d)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

O método “ones” faz o mesmo mas com arras com o valor 1. Veja:

```
e = np.ones([5,7])
# cria uma matriz de 5x7 com valores 1
print(e)
```

```
[[1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1.]]
```

O método “eye” cria matrizes quadaradas com zeros e diagonal principal 1. Veja:

```
f = np.eye(5)
# cria matriz quadrada com diagonal principal com valores 1 e os ou
tros valores zero
print(f)
```

```
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

É possível ainda carregar um array com valores aleatórios entre 0 e 1. Para isso usamos o método “random” do NumPy. Veja o exemplo:

```
g = np.random.random((5))
print(g)
```

```
[0.19367658 0.5184406  0.82765223 0.49897707 0.9167044 ]
```

É possível ainda carregar um array com valores aleatórios também com sinal negativo. Para isso usamos o método “randn” do NumPy. Preenche o array com flutuações aleatórias amostradas a partir de uma distribuição univariada "normal" (Gaussiana) de média 0 e variância 1.Veja o exemplo:

```
g2= np.random.randn((5))
print(g2)
```

```
[ 0.03645266  0.24951722 -0.80311438 -1.25796416  0.81868561]
```

É possível ainda usar o método “random” para criar arrays multidimensionais. No exemplo a seguir fazemos isso criando valores entre 0 e 10 já que multiplicamos por 10 o valor randomizado e usamos ainda o método “floor” que retorna o valor absoluto do número:

```
h = np.floor(10*np.random.random((3,4)))
print(h)
h = (10*np.random.random((3,4)))
print(h)
h = np.floor(h)
print(h)
```

```
[[5. 7. 0. 1.]
 [8. 0. 3. 5.]
 [0. 7. 9. 3.]]
[[6.13964214 2.93186177 6.00449072 5.7609196 ]
 [1.55265622 6.12293642 9.89994306 3.73331244]
 [3.72965958 8.8907206  4.00640842 3.89901736]]
[[6. 2. 6. 5.]
 [1. 6. 9. 3.]
 [3. 8. 4. 3.]]
```

O uso da geração de números aleatórios é uma parte importante da configuração e avaliação de muitos algoritmos numéricos e de aprendizado de máquina. Se você precisa inicializar pesos aleatoriamente em uma rede neural artificial, dividir dados em conjuntos aleatórios ou embaralhar seu conjunto de dados aleatoriamente, ser capaz de gerar números aleatórios é essencial.

Outra forma de gerar valores randômicos é a seguinte:

```
gnr = np.random.default_rng(0)
i = gnr.random(3)
print (i)
```

```
[0.63696169 0.26978671 0.04097352]
```

Com o gerador de inteiros você pode gerar inteiros aleatórios indicando o valor inicial (lembre-se de que isso é inclusivo com NumPy) até o valor final (exclusivo). Você pode definir "endpoint = True" para tornar o valor final inclusivo. Veja o exemplo:

```
i = gnr.integers(10, size=(3, 4))
print(i)
```

```
[[5 9 2 8]
 [6 0 3 8]
 [5 0 7 7]]
```

Já a função "unique" permite obtermos a quantidade de números em um array sem repetições. Veja o exemplo:

```
j = np.array([11, 12, 13, 14, 15, 16, 17, 12, 13, 11, 18, 19, 20])
j = np.unique(j)
print(j)
```

```
[11 12 13 14 15 16 17 18 19 20]
```

Existem ainda uma série de funções matemáticas que podem ser feitas com os array. Veja algumas delas no exemplo a seguir:

```
k = np.array([[17,22,43],[27,25,14],[15,24,32]])
# cria a matriz bidimensional k
print(k)
# Mostra a matriz k
print(k[0][1])
# Mostra um elemento específico da matriz k
print(k.shape)
# Mostra o tamanho das dimensões da matriz k
print(k.max())
# Mostra o maior valor da matriz k
print(k.min())
# Mostra o menor valor da matriz k
print((k.sum()))
# Mostra a soma dos valores da matriz k
```

```
print(k.mean())
# Mostra o valor da média dos valores da matriz k
print(k.std())
# Mostra o valor do desvio padrão (standard deviation) dos valores
da matiz k
```

```
[[17 22 43]
 [27 25 14]
 [15 24 32]]
22
(3, 3)
43
14
219
24.333333333333332
8.615231988880057
```

Das funções citadas somente a última que calcula o desvio padrão que é uma medida de dispersão é que precisa ser explicada. O desvio padrão tem relação com quanto um determinado grupo de dados varia. Ela é calculada a partir raiz quadrada do somatório do quadrado da diferença dos valores à média dividido pelo número de valores:

$$DP = \sqrt{\frac{\sum_{i=1}^{n}(x_i - M_A)^2}{n}}$$

Considere,

∑ : somatório (indica que temos que somar todos os termos, de i=1 até a posição n)
x_i : valor na posição **i** no conjunto de dados
M_A : média aritmética dos dados
n : quantidade de dados

Este é só um dos exemplos de funções estatísticas disponíveis no NumPy. Note que estas funções têm uma série de outros parâmetros não usados neste exemplo. Para ver as variações digite o comando e ele mostra a sintaxe. Vale a pena você explorar um pouco estes parâmetros.

Veja agora mais dois exemplos de funções do NumPy chamadas universais porque são aplicadas a todos os elementos do array, que bastante usadas:

```
k1 = np.array([1, 4, 9, 16, 25, 36])
print(np.sqrt(k1))
# Mostra o valor da raiz quadrada de todos elementos
print(np.exp(k1))
# Mostra o valor do exponencial de todos elementos
```

```
[1. 2. 3. 4. 5. 6.]
[2.71828183e+00 5.45981500e+01 8.10308393e+03 8.88611052e+06
 7.20048993e+10 4.31123155e+15]
```

O NumPy tem inúmeras outras funções e recursos. Não será possível abordarmos todas aqui, mas já foi possível perceber o seu poder computacional e potencial em manipular dados numéricos estruturados. Para consultar mais funções específicas acesse o link a seguir:

https://numpy.org/

Mesmo sem poder explorar toda a enormidade de funções disponíveis do NumPy ainda conheceremos alguns recursos importantes. Um deles é a maneira de extrair linhas/colunas de um array multidimensional. Para isso basta indicar entre colchetes a dimensão que deseja extrair e na outra um ":". Veja o exemplo a seguir:

```
l = np.array([[4, 5], [6, 1], [7, 4]])
print(l)
l_linha_1 = l[0, :]
print(l_linha_1)
l_linha_2 = l[1, :]
print(l_linha_2)
l_linha_3 = l[2, :]
print(l_linha_3)
l_coluna_1 = l[:, 0]
print(l_coluna_1)
l_coluna_2 = l[:, 1]
print(l_coluna_2)
```

```
[[4 5]
 [6 1]
 [7 4]]
[4 5]
[6 1]
[7 4]
[4 6 7]
[5 1 4]
```

Além desse recurso de extração de linhas e colunas é possível realizar com os arrays do NumPy todos os recursos de fatiamento que foram apresentados para listas em Python.

```
m = np.array([1, 2, 3, 4, 5, 6])
print(m[1])
# Mostra o elemento da posição 2
print(m[0:2])
# Mostra o array criado a partir da posição 0, dois elementos
print(m[1:])
# Mostra o array criado a partir da 2a posição
# até todo o restante do array
print(m[-3:])
# Mostra o array criado a partir da antepenúltima
#posição até o final
```

```
2
[1 2]
[2 3 4 5 6]
```

```
[4 5 6]
```

Depois de criar suas matrizes, você pode adicioná-las e multiplicá-las usando operadores aritméticos se tiver duas matrizes do mesmo tamanho. Veja o exemplo:

```
n = np.array([[1, 2], [3, 4]])
o = np.array([[1, 1], [1, 1]])
res1 = n+o
print(res1)
res2 = n*o
print(res2)
```

```
[[2 3]
 [4 5]]
[[1 2]
 [3 4]]
```

Outro exemplo:

```
p = np.array([[1, 2], [3, 4], [5, 6]])
q = np.array([[2, 1]])
print(p+q)
```

```
[[3 3]
 [5 5]
 [7 7]]
```

Transpor e rearranjar eixos é uma tarefa comum em processamento de dados e cálculos numéricos. Por isso o NumPy fornece uma gama de métodos e funções que facilitam estas atividades. Veja este exemplo a seguir que dimensiona uma matriz criada com o “arange” redimensionando-a para 3x5 e depois criando a transposta com o “.T”:

```
r = np.arange(15).reshape((3, 5))
# rearranja um conjunto de 15 elementos de 0 a 14 em 3 linhas e 5 c
olunas.
print(r)
# mostra a matrizes transposta entre linha e coluna
s = r.T
print(s)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
[[ 0  5 10]
 [ 1  6 11]
 [ 2  7 12]
 [ 3  8 13]
 [ 4  9 14]]
```

Uma outra forma de fazê-lo é usando o “transpose”. Veja:

```
r = np.arange(15).reshape((3, 5))
```

```
# rearranja um conjunto de 15 elementos de 0 a 14 em 3 linhas e 5 c
olunas.
print(r)
# mostra a matrizes transposta entre linha e coluna
s = r.transpose((1,0))
print(s)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
[[ 0  5 10]
 [ 1  6 11]
 [ 2  7 12]
 [ 3  8 13]
 [ 4  9 14]]
```

Usando o "transpose" é possível de forma interessante e rápida transpor as diversas dimensões. Olha esta transposição das duas primeiras dimensões de um array de 3 dimensões:

```
t = np.arange(16).reshape((2, 2, 4))
print(t)
# Aqui vamos rearranjar as linhas, colocando-as
# na ordem desejada. Aqui trocamos de posição os eixos 0 e 1 do
# array. O eixo 2 permaneceu no seu lugar.
u = t.transpose((1, 0, 2))
print(u)
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]]

 [[ 8  9 10 11]
  [12 13 14 15]]]
[[[ 0  1  2  3]
  [ 8  9 10 11]]

 [[ 4  5  6  7]
  [12 13 14 15]]]
```

Existe uma forma mais simples para simplesmente trocar eixos de lugar com a função "swapaxes", que troca os eixos de lugar para organizar os dados. Veja:

```
t = np.arange(16).reshape((2, 2, 4))
print(t)
u = t.swapaxes(1, 2)
print(u)
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]]

 [[ 8  9 10 11]
  [12 13 14 15]]]
[[[ 0  4]
  [ 1  5]
```

```
  [ 2  6]
  [ 3  7]]

 [[ 8 12]
  [ 9 13]
  [10 14]
  [11 15]]]
```

Agora apresentaremos como usar expressões lógicas em array. O NumPy nos fornece a função “where” que substitui um laço condicional if-else. Veja o seu uso:

```
v = np.random.randn(4, 4)
print(v)
# criando matriz com valores aleatórios positivos e negativos
x = (v > 0)
# criando matriz com valores booleanos baseado no array v
print(x)
z = np.where(x > 0, 1, -1)
# criando matriz com valores -1 e 1 baseado nos valores do array x
print(z)
```

```
[[ 0.94519168 -0.54177712  0.09565931  0.1858831 ]
 [ 1.50389454  1.86282887 -0.78315929  1.73635337]
 [ 0.73938699  0.13037564  0.99285671 -0.0843752 ]
 [-0.47857647  0.11604454  1.33391729  3.05820096]]
[[ True False  True  True]
 [ True  True False  True]
 [ True  True  True False]
 [False  True  True  True]]
[[ 1 -1  1  1]
 [ 1  1 -1  1]
 [ 1  1  1 -1]
 [-1  1  1  1]]
```