Linear algebra in Python with NumPy

Numpy is one of the most used libraries in Python for arrays manipulation. It adds to Python a set of functions that allows us to operate on large multidimensional arrays with just a few lines.

In [1]:

```
import numpy as np
```

Defining lists and numpy arrays

In [2]:

```
alist = [1, 2, 3, 4, 5]   # Define a python list. It looks like an np array
narray = np.array([1, 2, 3, 4]) # Define a numpy array
```

Note the difference between a Python list and a NumPy array.

In [3]:

```
print(alist)
print(narray)

print(type(alist))
print(type(narray))
```

```
[1, 2, 3, 4, 5]
[1 2 3 4]
<class 'list'>
<class 'numpy.ndarray'>
```

Algebraic operators on NumPy arrays vs. Python lists

Note that the '+' operator on NumPy arrays perform an element-wise addition, while the same operation on Python lists results in a list concatenation.

In [4]:

```
print(narray + narray)
print(alist + alist)
```

```
[2 4 6 8]
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

It is the same as with the product operator, `*`. In the first case, we scale the vector, while in the second case, we concatenate three times the same list.

In [5]:

```
print(narray * 3)
print(alist * 3)
```

```
[ 3  6  9 12]
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

Matrix or Array of Arrays

With NumPy, we have two ways to create a matrix:

- **Creating an array of arrays using `np.array` (recommended)**

- Creating an array of arrays using `np.array` (recommended).
- Creating a matrix using `np.matrix` (still available but might be removed soon).

In [6]:

```
npmatrix1 = np.array([narray, narray, narray]) # Matrix initialized with NumPy arrays
npmatrix2 = np.array([alist, alist, alist]) # Matrix initialized with lists
npmatrix3 = np.array([narray, [1, 1, 1, 1], narray]) # Matrix initialized with both type
s

print(npmatrix1)
print(npmatrix2)
print(npmatrix3)
```

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

However, when defining a matrix, be sure that all the rows contain the same number of elements. Otherwise, the linear algebra operations could lead to unexpected results.

Analyze the following two examples:

In [7]:

```
# Example 1:

okmatrix = np.array([[1, 2], [3, 4]]) # Define a 2x2 matrix
print(okmatrix) # Print okmatrix
print(okmatrix * 2) # Print a scaled version of okmatrix
```

```
[[1 2]
 [3 4]]
[[2 4]
 [6 8]]
```

In [8]:

```
# Example 2:

badmatrix = np.array([[1, 2], [3, 4], [5, 6, 7]]) # Define a matrix. Note the third row
contains 3 elements
print(badmatrix) # Print the malformed matrix
print(badmatrix * 2) # It is supposed to scale the whole matrix
```

```
[list([1, 2]) list([3, 4]) list([5, 6, 7])]
[list([1, 2, 1, 2]) list([3, 4, 3, 4]) list([5, 6, 7, 5, 6, 7])]
```

Scaling and translating matrices

Now that you know how to build correct NumPy arrays and matrices, let us see how easy it is to operate with them in Python using the regular algebraic operators like + and -.

Operations can be performed between arrays and arrays or between arrays and scalars.

In [9]:

```
# Scale by 2 and translate 1 unit the matrix
result = okmatrix * 2 + 1 # For each element in the matrix, multiply by 2 and add 1
print(result)
```

```
[[3 5]
 [7 9]]
```

In [10]:

```
# Add two sum compatible matrices
result1 = okmatrix + okmatrix
print(result1)

# Subtract two sum compatible matrices. This is called the difference vector
result2 = okmatrix - okmatrix
print(result2)
```

```
[[2 4]
 [6 8]]
[[0 0]
 [0 0]]
```

The product operator `*` when used on arrays or matrices indicates element-wise multiplications. Do not confuse it with the dot product.

In [11]:

```
result = okmatrix * okmatrix # Multiply each element by itself
print(result)
```

```
[[ 1  4]
 [ 9 16]]
```

Transpose a matrix

In linear algebra, the transpose of a matrix is an operator that flips a matrix over its diagonal, i.e., the transpose operator switches the row and column indices of the matrix producing another matrix. If the original matrix dimension is n by m, the resulting transposed matrix will be m by n.

T denotes the transpose operations with NumPy matrices.

In [12]:

```
matrix3x2 = np.array([[1, 2], [3, 4], [5, 6]]) # Define a 3x2 matrix
print('Original matrix 3 x 2')
print(matrix3x2)
print('Transposed matrix 2 x 3')
print(matrix3x2.T)
```

```
Original matrix 3 x 2
[[1 2]
 [3 4]
 [5 6]]
Transposed matrix 2 x 3
[[1 3 5]
 [2 4 6]]
```

However, note that the transpose operation does not affect 1D arrays.

In [13]:

```
nparray = np.array([1, 2, 3, 4]) # Define an array
print('Original array')
print(nparray)
print('Transposed array')
print(nparray.T)
```

```
Original array
[1 2 3 4]
Transposed array
[1 2 3 4]
```

perhaps in this case you wanted to do:

In [14]:

```
nparray = np.array([[1, 2, 3, 4]]) # Define a 1 x 4 matrix. Note the 2 level of square b
rackets
print('Original array')
print(nparray)
print('Transposed array')
print(nparray.T)
```

```
Original array
[[1 2 3 4]]
Transposed array
[[1]
 [2]
 [3]
 [4]]
```

Get the norm of a nparray or matrix

In linear algebra, the norm of an n-dimensional vector $\vec a$ is defined as:

$$norm(\vec a)$$
$$= ||\vec a||$$
$$= \sqrt{\sum_{i=1}^{n} a_i^2}$$

Calculating the norm of vector or even of a matrix is a general operation when dealing with data. Numpy has a set of functions for linear algebra in the subpackage linalg, including the norm function. Let us see how to get the norm a given array or matrix:

In [15]:

```
nparray1 = np.array([1, 2, 3, 4]) # Define an array
norm1 = np.linalg.norm(nparray1)

nparray2 = np.array([[1, 2], [3, 4]]) # Define a 2 x 2 matrix. Note the 2 level of squar
e brackets
norm2 = np.linalg.norm(nparray2)

print(norm1)
print(norm2)
```

```
5.477225575051661
5.477225575051661
```

Note that without any other parameter, the norm function treats the matrix as being just an array of numbers. However, it is possible to get the norm by rows or by columns. The axis parameter controls the form of the operation:

- **axis=0 means get the norm of each column**
- **axis=1 means get the norm of each row.**

In [16]:

```
nparray2 = np.array([[1, 1], [2, 2], [3, 3]]) # Define a 3 x 2 matrix.

normByCols = np.linalg.norm(nparray2, axis=0) # Get the norm for each column. Returns 2 e
lements
normByRows = np.linalg.norm(nparray2, axis=1) # get the norm for each row. Returns 3 elem
ents

print(normByCols)
print(normByRows)
```

```
[3.74165739 3.74165739]
[1.41421356 2.82842712 4.24264069]
```

The dot product between arrays: All the flavors

The dot product or scalar product or inner product between two vectors $\vec{a}$ and $\vec{a}$ of the same size is defined as:

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i$$

The dot product takes two vectors and returns a single number.

In [17]:

```
nparray1 = np.array([0, 1, 2, 3]) # Define an array
nparray2 = np.array([4, 5, 6, 7]) # Define an array

flavor1 = np.dot(nparray1, nparray2) # Way-1
print(flavor1)

flavor2 = np.sum(nparray1 * nparray2) # Way-2
print(flavor2)

flavor3 = nparray1 @ nparray2         # Way-3
print(flavor3)

# As you never should do:        #Way-4
flavor4 = 0
for a, b in zip(nparray1, nparray2):
    flavor4 += a * b

print(flavor4)
```

```
38
38
38
38
```

Recommend np.dot: since it is the only method that accepts arrays and lists without problems

In [18]:

```
norm1 = np.dot(np.array([1, 2]), np.array([3, 4])) # Dot product on nparrays
norm2 = np.dot([1, 2], [3, 4]) # Dot product on python lists

print(norm1, '=', norm2 )
```

```
11 = 11
```

Finally, note that the norm is the square root of the dot product of the vector with itself.

$$norm(\vec{a}) = ||\vec{a}|| = \sqrt{\sum_{i=1}^{n} a_i^2} = \sqrt{a \cdot a}$$

Sums by rows or columns

Another general operation performed on matrices is the sum by rows or columns. Just as we did for the function norm, the axis parameter controls the form of the operation:

- **axis=0 means to sum the elements of each column together.**
- **axis=1 means to sum the elements of each row together.**

In [ ]:

```
nparray2 = np.array([[1, -1], [2, -2], [3, -3]]) # Define a 3 x 2 matrix.

sumByCols = np.sum(nparray2, axis=0) # Get the sum for each column. Returns 2 elements
sumByRows = np.sum(nparray2, axis=1) # get the sum for each row. Returns 3 elements

print('Sum by columns: ')
print(sumByCols)
print('Sum by rows:')
print(sumByRows)
```

```
Sum by columns:
[ 6 -6]
Sum by rows:
[0 0 0]
```

Get the mean by rows or columns

As with the sums, one can get the mean by rows or columns using the axis parameter. Just remember that the mean is the sum of the elements divided by the length of the vector

$$mean(\vec a) = \frac{\sqrt{\sum_{i=1}^{n} a_i}}{n}$$

In [ ]:

```
nparray2 = np.array([[1, -1], [2, -2], [3, -3]]) # Define a 3 x 2 matrix. Chosen to be a
matrix with 0 mean

mean = np.mean(nparray2) # Get the mean for the whole matrix
meanByCols = np.mean(nparray2, axis=0) # Get the mean for each column. Returns 2 elements
meanByRows = np.mean(nparray2, axis=1) # get the mean for each row. Returns 3 elements

print('Matrix mean: ')
print(mean)
print('Mean by columns: ')
print(meanByCols)
print('Mean by rows:')
print(meanByRows)
```

```
Matrix mean:
0.0
Mean by columns:
[ 2. -2.]
Mean by rows:
[0. 0. 0.]
```

Center the columns of a matrix

Centering the attributes of a data matrix is another essential preprocessing step. Centering a matrix means to remove the column mean to each element inside the column. The sum by columns of a centered matrix is always 0.

With NumPy, this process is as simple as this:

In [ ]:

```
nparray2 = np.array([[1, 1], [2, 2], [3, 3]]) # Define a 3 x 2 matrix.

nparrayCentered = nparray2 - np.mean(nparray2, axis=0) # Remove the mean for each column

print('Original matrix')
print(nparray2)
print('Centered by columns matrix')
print(nparrayCentered)
print('New mean by column')
print(nparrayCentered.mean(axis=0))
```

```
Original matrix
[[1 1]
 [2 2]
 [3 3]]
Centered by columns matrix
[[-1. -1.]
 [ 0.  0.]
 [ 1.  1.]]
New mean by column
[0. 0.]
```

Warning: This process does not apply for row centering. In such cases, consider transposing the matrix, centering by columns, and then transpose back the result.

See the example below:

In [ ]:

```
nparray2 = np.array([[1, 3], [2, 4], [3, 5]]) # Define a 3 x 2 matrix.

nparrayCentered = nparray2.T - np.mean(nparray2, axis=1) # Remove the mean for each row
nparrayCentered = nparrayCentered.T # Transpose back the result

print('Original matrix')
print(nparray2)
print('Centered by columns matrix')
print(nparrayCentered)

print('New mean by rows')
print(nparrayCentered.mean(axis=1))
```

```
Original matrix
[[1 3]
 [2 4]
 [3 5]]
Centered by columns matrix
[[-1.  1.]
 [-1.  1.]
 [-1.  1.]]
New mean by rows
[0. 0. 0.]
```

Exercise:

1) Create Two numpy array of size 3 X 2 and 2 X 3

2) Randomly Initalize that array

3) Perform matrix multiplication

4) Perform elementwise matrix multiplication

5) Find mean of first matrix

6) Convert Numeric entries(columns) of mtcars.csv to Mean Centered Version