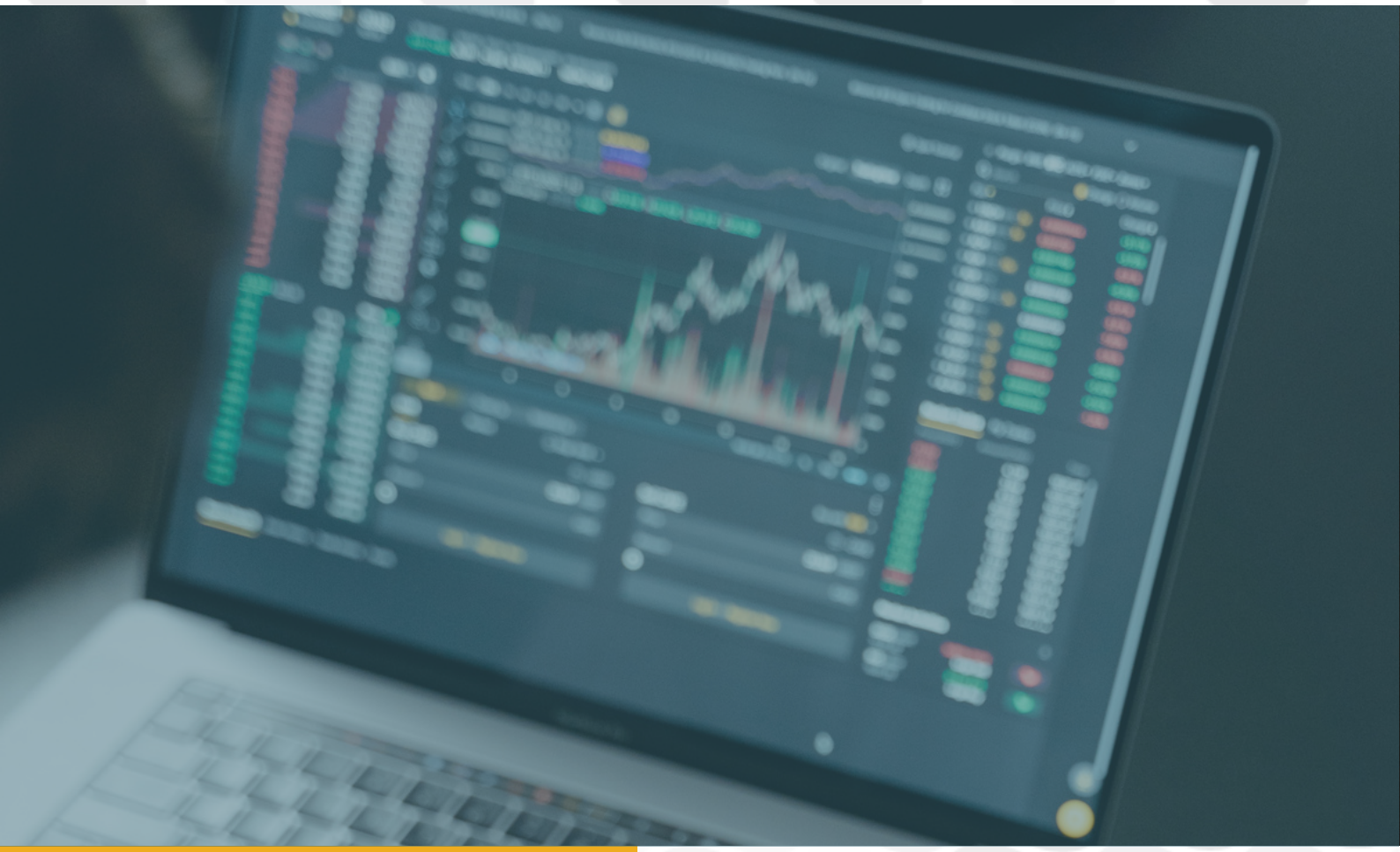


02
Numpy

Linear Algebra

Machine learning and deep learning models are data hungry. The performance of them is highly dependent on the amount of data. Thus, we tend to collect as much data as possible in order to build a robust and accurate model. As the amount of data increases, the operations done with scalars start to be inefficient. We need vectorized or matrix operations to make computations efficiently. That's where linear algebra comes.

Linear algebra: mathematical discipline that deals with vectors and matrices and, more generally, with vector spaces and linear transformations.

There are different types of objects (or structures) in linear algebra:

- Scalar: Single number
- Vector: Array of numbers
- Matrix: 2-dimensional array of numbers
- Tensor: N-dimensional array of numbers where n > 2

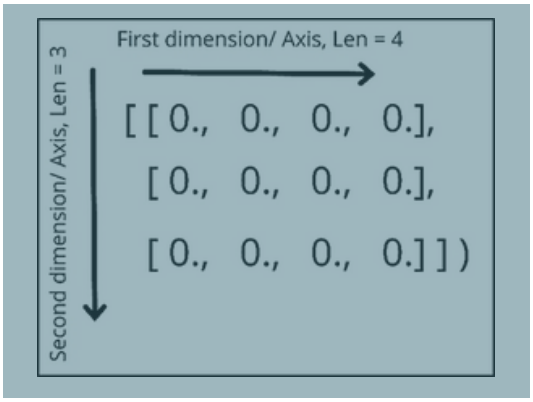
The best way to work with linear algebra in Python is through NumPy library.

What is NumPy?

NumPy is an open source Python library used for working with arrays. It also has functions for working in the domain of linear algebra, Fourier Transform, and matrices. NumPy was created in 2005 by Travis Oliphant It is an open source project and you can use it freely NumPy stands for "Numerical Python". List serve the purpose of arrays, but they are slow to process. NumPy aim to provide an array object that is up to 50x faster than traditional Python lists. The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working wit ndarray very easy. Arrays are very frequently used in machine learning.

Installation of NumPy

- Pip install numpy
- If this command fails, then use a python distribution that already has NumPy installed like Anaconda, Spyder etc or use Google Colab.



The above array has a rank of 2 since it is 2 dimensional.

NumPy– Introduction

NumPy's main object is the homogeneous multidimensional array. It is a table of elements:

- Usually numbers
- Same type
- Indexed by a tuple of positive integers

Dimensions in NumPy are called axes. Number of axes is rank.

Creating NumPy Arrays

- np.array()

Creating NumPy array from Python sequences (List, Tuple, etc.)

```
import numpy as np

a = np.array([1, 2, 3])
b = np.array((3, 4, 5))

if type(a) == type(b):
    print(type(a))
<class 'numpy.ndarray'>
```

- np.ones()

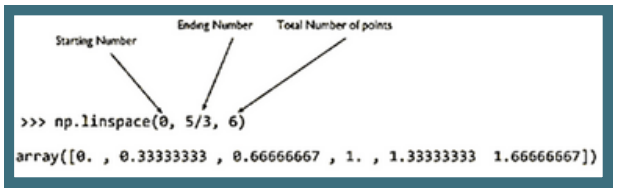
```
x = np.ones((3,4), dtype=np.int16)
array([[0., 0., 0., 0.],
[0., 0., 0., 0.], [0., 0.,
0., 0.]])
```

- np.zeros()

```
x = np.zeros((3,4))
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]], dtype=int16)
```

- np.linspace()

Returns an array having a specific number of points.
Evenly distributed between two values.
The maximum value is included, contrary to arange().



- np.full()

```
x = np.full((3,4),0.11)
array([[0.11, 0.11, 0.11, 0.11],
       [0.11, 0.11, 0.11, 0.11],
       [0.11, 0.11, 0.11, 0.11]])
```

- np.arrange()

```
x = np.arange(10, 12, 0.4)
array([10. , 10.4, 10.8,
11.2, 11.6])
```

- np.random.rand()

Creating an array with random numbers

```
x = np.random.rand(2,3)
array([[0.73893963, 0.66912285, 0.05888493],
[0.69173691, 0.77312144, 0.54114232]])
```

Data Types in NumPy

Numpy provides a large set of numeric datatypes that can be used to construct arrays. At the time of Array creation, Numpy tries to guess a datatype, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype.

Data Type	Description
bool_	Boolean (True or False) stored as a byte
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2.15E-9 to 2.15E+9)
int64	Integer (-9.22E-18 to 9.22E+18)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4.29E+9)
uint64	Unsigned integer (0 to 1.84E+19)
float16	Half precision signed float
float32	Single precision signed float
float64	Double precision signed float
complex64	Complex number: two 32-bit floats (real and imaginary components)
complex128	Complex number: two 64-bit floats (real and imaginary components)

Important Attributes of NumPy

```
x = np.array([[1., 0., 0.],
              [0., 1., 2.]])

print(x.ndim)       2
print(x.shape)      (2, 3)
print(x.size)       6
print(x.dtype)      float64
print(x.itemsize)   8
```

NumPy's array class is called ndarray

- ndarray.ndim: Number of axes (dimensions) of the array
- ndarray.shape: Dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension
- ndarray.size: Total number of elements of the array. Equal to the product of the elements of shape
- ndarray.dtype: Datatype of the elements in the numpy array. (ndarray should have same type of elements)
- ndarray.itemsize: Returns the size (in bytes) of each item

Reshaping Arrays

Reshape() function is used to change the shape of an array while the itemsize remains the same.

```
a = np.arange(6)
print(a)
b = a.reshape(2, 3)
print(b)
```

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

```
a = np.arange(24)
print(a)
b = a.reshape(2,3,4)
print(b)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

Transposing and Flattening Arrays

Transpose() function is used to generate the transposition of an array.

```
a = np.array([[2, 3, 5], [1, 7, 8]])
b = np.transpose(a)
print(a)
print(b)
```

```
[[2 3 5]
 [1 7 8]]

[[2 1]
 [3 7]
 [5 8]]
```

Flatten() method is used to flatten an array to one dimensional array.

```
a = np.array([[2, 3, 5], [1, 7, 8]])
print(a.flatten())
```

```
[2 3 5 1 7 8]
```

Indexing and Slicing 1-D NumPy Arrays

```
a = np.array([1, 5, 3, 19, 13, 7, 3])

a[3]
19

a[2:5]
[ 3 19 13]

a[2::2]
[ 3 13 3]

a[::-1]
[ 3 7 13 19 3 5 1]
```

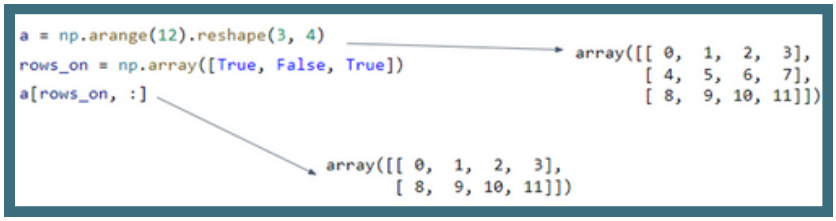
Indexing Multi Dimensional NumPy Arrays

Multi-dimensional arrays can be accessed as

```
Array[row_start_index:row_end_index, column_start_index:  column_end_index]

b[1, 2]   # row 1, col 2
b[1, :]   # row 1, all columns
b[:, 1]   # all rows, column 1
```

We can also use boolean indexing



Joining Arrays

Similar to Python lists and tuples, NumPy arrays can be concatenated together. However, because NumPy's arrays can be multi-dimensional, we can choose the dimension along which arrays are joined.

```
# demonstrating methods for joining arrays
>>> x = np.array([1, 2, 3])
>>> y = np.array([-1, -2, -3])

# stack `x` and `y` "vertically"
>>> np.vstack([x, y])
array([[ 1,  2,  3],
       [-1, -2, -3]])

# stack `x` and `y` "horizontally"
>>> np.hstack([x, y])
array([ 1,  2,  3, -1, -2, -3])
```

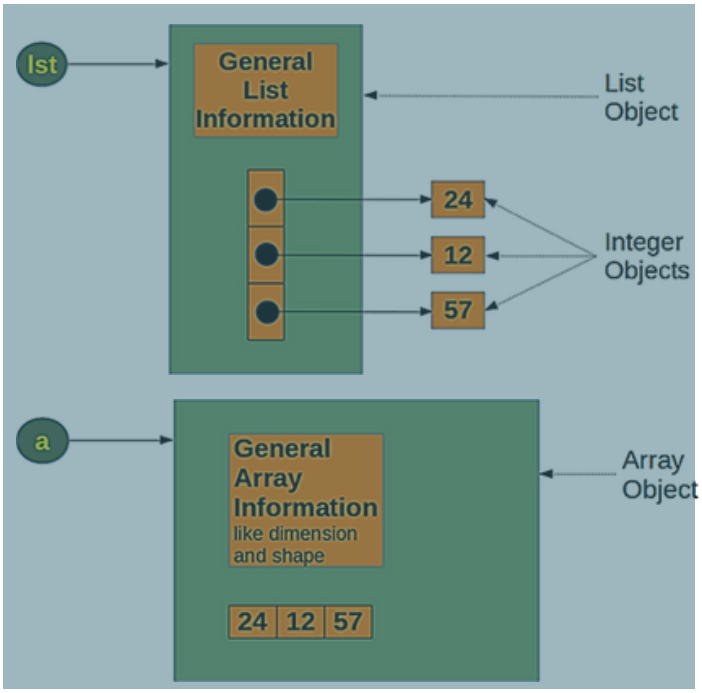
Difference between Python Lists and Numpy Arrays

- Memory consumption

In this example, a Python list and a Numpy array of size 1000 will be created. Let's see the comparison

```
S = range(1000)
print("Size of each element of list in bytes: ",sys.getsizeof(S))
print("Size of the whole list in bytes: ",sys.getsizeof(S)*len(S))
D = np.arange(1000)
print("Size of each element of the Numpy array in bytes: ",D.itemsize)
print("Size of the whole Numpy array in bytes: ",D.size*D.itemsize)

Output:
Size of each element of list in bytes:  48
Size of the whole list in bytes:  48000
Size of each element of the Numpy array in bytes:  4
Size of the whole Numpy array in bytes:  4000
```



- If you assign a single value to an ndarray slice, it is copied across the whole slice :

```
a = np.array([1, 2, 5, 7, 8])
a[1:3] = -1
array([ 1, -1, -1,  7,  8])
b = [1, 2, 5, 7, 8]
b[1:3] = -1
TypeError: can only assign an iterable
```

Basic Mathematical Operations

In general, NumPy implements mathematical functions such that, when a function acts on an array, the mathematical operation is applied to each entry in the array.

```
x = np.array([[ 0.,  1.,  2.],
              [ 3.,  4.,  5.],
              [ 6.,  7.,  8.]])
```

- Time comparison

Let's see the performance of the code

```
list1 = range(1000000)
list2 = range(1000000)
initialTime = time.time()
resultantList = [(a * b) for a, b in zip(list1,
list2)]
print("Time taken by Lists to perform
multiplication:",
      (time.time() - initialTime),
      "seconds")

array1 = np.arange(1000000)
array2 = np.arange(1000000)
initialTime = time.time()
resultantArray = array1 * array2
print("Time taken by NumPy Arrays to perform
multiplication:",
      (time.time() - initialTime),
      "seconds")
```

Output:
Time taken by Lists to perform multiplication: 0.12701106071472168 seconds.
Time taken by NumPy Arrays to perform multiplication: 0.00200653076171875 seconds

- ndarray slices are actually views on the same data buffer. If you modify it, it is going to modify the original ndarray as well.

```
a = np.array([1, 2, 5, 7, 8])
a_slice = a[1:5]
a_slice[1] = 1000
a # original array will also be modified
array([1, 2, 1000, 7, 8])
```

usually we use copy() method to create a copy of ndarray so the original array remains the same.

```
a = np.array([1, 2, 5, 7, 8])
a_slice = a[1:5].copy()
a_slice[1] = 1000
a # original array remains the same
array([1, 2, 5, 7, 8])
```

```
x**2
array([[ 0.,  1.,  4.],
       [ 9., 16., 25.],
       [36., 49., 64.]])
```

```
np.sqrt(x)
array([[0.        , 1.        , 1.41421356],
       [1.73205081, 2.        , 2.23606798],
       [2.44948974, 2.64575131, 2.82842712]])
```

```
1.5 + x[0, :] # operations to
each entry of the sliced array
array([1.5, 2.5, 3.5])
```

Similarly, mathematical operations performed between two arrays are designed to act on the corresponding pairs of entries between the two arrays:

```
x = np.array([[ 0.,  1.,  2.],
              [ 3.,  4.,  5.],
              [ 6.,  7.,  8.]])

y = np.array([[2. , -3.5, -3. ],
              [-2.5, -2. , -1.5],
              [-1. , -0.5, -0. ]])
```

```
x + y
array([[-4. , -2.5, -1. ],
       [ 0.5,  2. ,  3.5],
       [ 5. ,  6.5,  8. ]])
```

```
x * y
array([[-0. , -3.5, -6. ],
       [-7.5, -8. , -7.5],
       [-6. , -3.5, -0. ]])
```

Vectorized Operations

Recall that NumPy's ND-arrays are homogeneous. This restriction on an array's contents comes at a great benefit; in "knowing" that an array's contents are homogeneous in data type, NumPy is able to delegate the task of performing mathematical operations on the array's contents to optimized, compiled C code. This is a process that is referred to as vectorization.

The outcome of this can be a tremendous speedup relative to the analogous computation performed in Python, which must painstakingly check the data type of every one of the items as it iterates over the arrays, since Python typically works with lists with unrestricted contents.

For example summing integers 0-9999 using two methods below:

```
>>> import numpy as np

# sum an array, using NumPy's vectorized 'sum' function
>>> np.sum(np.arange(10000))  # takes 11 microseconds on my computer
49995000
```

```
# sum an array by explicitly looping over the array in Python
# this takes 822 microseconds on my computer
>>> total = 0
>>> for i in np.arange(10000):
...     total = i + total
>>> total
49995000
```

This should make it clear that, whenever computational efficiency is important, one should avoid performing explicit for-loops over long sequences of data in Python, be them lists or NumPy arrays. NumPy provides a whole suite of vectorized functions.

NumPy's Mathematical Functions

- Unary Functions

These familiar functions are defined to work on individual numbers (i.e. "scalars"), not sequences of numbers. Applying unary function will apply f(x) elementwise on the array and producing a new array as a result.

```
import numpy as np
>>> x = np.array([0., 1., 2.])

# produces array([exp(0.), exp(1.), exp(2.)])
# x is not overwritten by this; a new array
# is created
>>> np.exp(x)
array([ 1. ,  2.71828183,  7.3890561 ])
```

Unary Function: $f(x)$	NumPy Function
$\|x\|$	`np.absolute`
$\sqrt{x}$	`np.sqrt`
Trigonometric Functions	
$\sin x$	`np.sin`
$\cos x$	`np.cos`
$\tan x$	`np.tan`
Logarithmic Functions	
$\ln x$	`np.log`
$\log_{10} x$	`np.log10`
$\log_2 x$	`np.log2`
Exponential Functions	
e^x	`np.exp`

- Binary Functions

Applying a binary NumPy-function f(x,y) to two same-shape arrays will apply the function to each of their pairwise elements, producing an array of the same shape as either of the operands.

Binary Function: $f(x, y)$	NumPy Function	Python operator
$x \cdot y$	`np.multiply`	`*`
$x \div y$	`np.divide`	`/`
$x + y$	`np.add`	`+`
$x - y$	`np.subtract`	`-`
x^y	`np.power`	`**`
$x\%y$	`np.mod`	`%`

```
# example of a binary function operating on two 2D arrays
>>> x = np.array([[10,  2],
...               [ 3,  5]])

>>> y = np.array([[ 1,   0],
...               [ -4,  -1]])

>>> np.add(x, y)  # equivalent to `x + y`
array([[11,  2],
       [-1,  4]])

# add column-0 of `x` and row-1 of `y`
>>> x[:, 0] + y[1, :]
array([6, 2])
```

Note that np.multiply() different from performing matrix multiplication. This is elementwise multiplication. Matrix multiplication is implemented using np.matmul()

Notes:
It is important to note that NumPy arrays know how to "override" the standard mathematical operators so that they invoke vectorized functions. For example, suppose arr1 and arr2 are NumPy arrays; calling arr1 + arr2 ends up calling np.add(arr1, arr2) "under the hood". Thus we can safely use the standard math operators + - / * ** between NumPy arrays, and fast vectorized functions will be used for us.

- Sequential Functions

A sequential function expects a variable-length sequence of numbers as an input, and produces a single number as an output:

```
# demonstrating sequential functions
>>> x = np.array([0., 2., 4.])
>>> np.sum(x)  # can also be invoked as `x.sum()`
6.
>>> np.mean(x)  # can also be invoked as `x.mean()`
2.
```

Sequential Function: $f(\{x_i\}_{i=0}^{n-1})$	NumPy Function
Mean of $\{x_i\}_{i=0}^{n-1}$	`np.mean`
Median of $\{x_i\}_{i=0}^{n-1}$	`np.median`
Variance of $\{x_i\}_{i=0}^{n-1}$	`np.var`
Standard Deviation of $\{x_i\}_{i=0}^{n-1}$	`np.std`
Maximum Value of $\{x_i\}_{i=0}^{n-1}$	`np.max`
Minimum Value of $\{x_i\}_{i=0}^{n-1}$	`np.min`
Index of the Maximum Value of $\{x_i\}_{i=0}^{n-1}$	`np.argmax`
Index of the Minimum Value of $\{x_i\}_{i=0}^{n-1}$	`np.argmin`
Sum of $\{x_i\}_{i=0}^{n-1}$	`np.sum`

Sequential Functions - Multidimensional Arrays

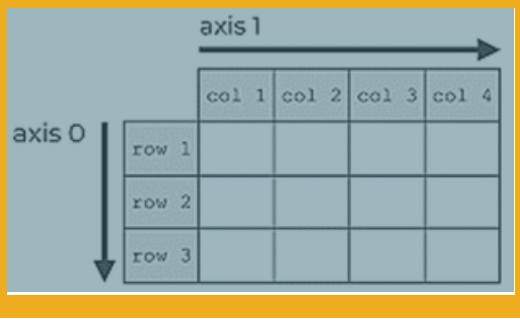
```
>>> x = np.array([[0, 1],
...               [2, 3],
...               [4, 5]])

# `sum`  will treat a multidimensional array
# as if it is a single sequence of numbers, by default
>>> np.sum(x)
15
```

```
# sum over axis-0, within axis-1
# i.e. sum over the rows, within each column
>>> np.sum(x, axis=0)  # equivalent: x.sum(axis=0)
array([6, 9])

# sum over axis-1, within axis-0
# i.e. sum over the columns, within each row
>>> np.sum(x, axis=1)  # equivalent: x.sum(axis=1)
array([1, 5, 9])
```

By default, NumPy's sequential functions treat any multidimensional array as if it had been reshaped to a 1-dimensional array. This default behavior of sequential NumPy functions can be overwritten by specifying the keyword argument axis within the sequential function.



Picture Source

- unsplash.com
- pexels.com