Machine Learning Diploma

Session 2: Numpy

Agenda:

1	Numpy Basics
2	Quick Array Creation
3	NPZ Files
4	Stacking
5	Comparison
6	Conditional Access & Modification
7	Array BroadCasting
8	Sparse Matrices

1. Numpy Basics

What is Numpy?

- One of the main primarily used data structures to represent & process data.
- Numpy is about creating N-dimensional Tensors, these Tensors are called Numpy Arrays or ndarray.
- Numpy arrays could be 0-dimensional(scalars), 1-dimensional(vectors), 2-dimensional(matrices), 3-dimensional(Images), etc.

Import:

```
import numpy as np
```

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

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

Create Numpy Arrays:

- **Create scalars (0-dimensional)**

```
s = np.array(5)
print(s)
```

```
5
```

- **Create vectors (1-dimensional)**

```
v = np.array([1, 2, 3])
print(v)
```

```
[1 2 3]
```

- **Create Matrices (2-dimensional)**

```
M = np.array([[1, 2, 3],
              [4, 5, 6]])
print(v)
```

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

Dimensions & shape:

- **Scalars**

```
s = np.array(5)
print(s.ndim)
print(s.shape)
```

```
0
()
```

- **Vectors**

```
v = np.array([1, 2, 3])
print(v.ndim)
print(v.shape)
```

```
1
(3,)
```

- **Matrices**

```
M = np.array([[1, 2, 3],
              [4, 5, 6]])
print(M.ndim)
print(M.shape)
```

```
2
(2, 3)
```

Reshape:

```
vec = np.array([1, 2, 3, 4, 5, 6])
mat = vec.reshape((3, 2))
mat
```

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

Add Dimension:

<table>
<tr>
<td>

- add dimension using `reshape`

```
vec = np.array([1, 2, 3, 4])
mat = vec.reshape((vec.shape[0], 1))
print(mat)
```

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

</td>
<td>

- add dimension using `None`

```
vec = np.array([1, 2, 3, 4])
mat = vec[:, None]
mat
```

```
array([[1],
       [2],
       [3],
       [4]])
```

</td>
</tr>
</table>

Dtype:

- Numpy array can only carry one data type.
- If you pass elements with different datatypes, Numpy will change all of them into one datatype (the most general dtype).
- Here is the order of general datatypes:
  - Object > String > Float > Int > Bool.

Dtype examples:

```
mat = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
print(mat.dtype)
```

```
int32
```

```
mat = np.array([[.1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
print(mat.dtype)
```

```
float64
```

```
mat = np.array([[1, 2, 3],
                [True, 5, 6],
                [7, 8, 9]])
print(mat.dtype)
```

```
int32
```

```
mat = np.array([[1, 2, 3],
                [4, "5", 6],
                [7, 8, 9]])
print(mat.dtype)
```

```
<U11
```

```
class C:
    x = 3
o1 = C()
mat = np.array([[o1, 1],
                [3, 2]])
print(mat.dtype)
```

```
object
```

Change Dtypes:

- Numpy allow you to Change the Datatype of ndarrays.
- Examples:

```python
Mat = np.array([[1, 2, 3],
                [4, "5", 6],
                [7, 8, 9]])
fMat = Mat.astype(float)
print(fMat)
print(fMat.dtype)
```

```
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
float64
```

```python
Mat = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
fMat = Mat.astype(str)
print(fMat)
print(fMat.dtype)
```

```
[['1' '2' '3']
 ['4' '5' '6']
 ['7' '8' '9']]
<U11
```

Indexing:

➢ Means accessing one element in Numpy array using its index.

- **Vector index**

```
v = np.array([3, 2, 5, 1])
v[1]
```

2

- **Matrix index**

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

6

- **Vector inverse index**

```
v = np.array([3, 2, 5, 1])
v[-2]
```

5

- **Matrix inverse index**

```
M = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9],
              [0, 0, 0]])
M[1, -2]
```

5

Slicing:

- Means accessing many elements in an array using index range.

- **Vector slicing**

```
v = np.array([3, 2, 5, 1, 2, 3, 0])
v[2:5]
```

```
array([5, 1, 2])
```

- **Matrix slicing**

```
M = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9],
              [0, 0, 0]])
M[1:3, 0:2]
```

```
array([[4, 5],
       [7, 8]])
```

- **Vector inverse slicing**

```
v = np.array([3, 2, 5, 1, 2, 3, 0])
v[-3:-1]
```

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

- **Matrix inverse index**

```
M = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9],
              [0, 0, 0]])
M[-3:-1, -3:-1]
```

```
array([[4, 5],
       [7, 8]])
```

Transopose:

➢ Make matrix columns become rows and rows become columns.

```
Mat = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
MatT = Mat.T
print(MatT)
```

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

2. Quick Array Creation

Quick arrays Methods :

- Numpy provides you some built-in methods that helps you create arrays quickly.
- In the coming slides, you will find the most popular methods.

Zeros():

- Creates an array of the specified size with the contents filled with zero values.

```
mat = np.zeros((3,4))
mat
```

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

Ones():

- Creates an array of the specified size with the contents filled with one values.

```
mat = np.ones((3, 4))
mat
```

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

Full():

- Creates a new array of the specified shape with the contents filled with a specified value.

```
mat = np.full((3, 3),5)
mat
```

```
array([[5, 5, 5],
       [5, 5, 5],
       [5, 5, 5]])
```

Empty():

- Allocates space of a numpy array in the memory without initializing array elements values.

```
mat = np.empty((3,3))
mat
```

```
array([[0.1, 2. , 3. ],
       [4. , 5. , 6. ],
       [7. , 8. , 9. ]])
```

- Use it when you want to create arrays quickly if you plan to fill them with meaningful values later.

arange():

- Generates values starting from a given start value, incrementing by a step.
- It takes three parameters; (start, stop, step).

```python
angles = np.arange(0, 361, 90)
angles
```

```
array([  0,  90, 180, 270, 360])
```

Linspace():

- Is used to create an array of evenly spaced values within a specified range, For example, np.linspace(0, 10, num=20) will create an array of 20 evenly spaced values between 0 and 10, including both 0 and 10.

```
vec = np.linspace(0, 10, num=10)
vec
```

```
array([ 0.        ,  1.11111111,  2.22222222,  3.33333333,  4.44444444,
        5.55555556,  6.66666667,  7.77777778,  8.88888889, 10.        ])
```

Linspace():

- To create 10X10 Matrix using Linspace:

```
vec = np.linspace(0, 10, num=25)
Mat = vec.reshape(5, 5)
Mat
```

```
array([[ 0.        ,  0.41666667,  0.83333333,  1.25      ,  1.66666667],
       [ 2.08333333,  2.5       ,  2.91666667,  3.33333333,  3.75      ],
       [ 4.16666667,  4.58333333,  5.        ,  5.41666667,  5.83333333],
       [ 6.25      ,  6.66666667,  7.08333333,  7.5       ,  7.91666667],
       [ 8.33333333,  8.75      ,  9.16666667,  9.58333333, 10.        ]])
```

Identity ():

- Creates an identity matrix with a given shape.

```
I = np.identity(3, dtype="float32")
I
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]], dtype=float32)
```

Random.random():

- Creates an array of a given shape with random values between 0 & 1.

```python
np.random.random((3, 3))
```

```
array([[0.09722172, 0.83765299, 0.54428848],
       [0.81425031, 0.99812168, 0.96502651],
       [0.32159268, 0.19904197, 0.77736707]])
```

Random.randint():

- Creates an array of a given shape with random values between a & b, where a & b are boundaries that you specify.

```
1  np.random.randint(0, 11, (5, 5))
```

```
array([[ 3,  6, 10,  1,  4],
       [ 7,  3,  1,  1,  7],
       [ 7,  3,  9,  2,  6],
       [ 0,  9,  7,  2,  1],
       [ 9,  6,  2,  9,  8]])
```

Random.choice():

- Creates an array of a given shape with random values sampled from elements of another array.

```
a = ["ali", 3, "omar", False]
np.random.choice(a, size=(3, 5))
```

```
array([['omar', 'ali', 'ali', 'ali', 'False'],
       ['omar', 'ali', 'False', 'ali', 'ali'],
       ['3', '3', 'omar', 'omar', '3']], dtype='<U11')
```

3. NPZ files

What are NPZ files?

- Numpy provides a way to save your numpy arrays as files called npz files which helps you to save the data you want in npz format.
- You can save the files using a built-in method called savez, and you can load using a built-in method called load.

Save Numpy arrays:

```python
import numpy as np
Mat1 = np.ones((3, 4))
Mat2 = np.zeros((5, 3))

np.savez('file.npz', Ones_Mat=Mat1, Zeros_Mat=Mat2)
```

Load Numpy arrays:

```python
with np.load('file.npz') as file:
    Mat1 = file['Ones_Mat']
    Mat2 = file['Zeros_Mat']

print(Mat1)
print("-------")
print(Mat2)
```

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

4. Stacking

What Is Stacking?

- Stacking means Concatenating two matrices together.
- There are two types of Stacking:

Vertical Stacking

- The two matrices must have the same number of columns

```python
m1 = np.array([[1, 2, 3],
               [0, 0, 0]])

m2 = np.array([[4, 5, 6],
               [1, 1, 1]])

mat = np.vstack((m1, m2))
mat
```

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

Horizontal Stacking

- The two matrices must have the same number of rows.

```python
m1 = np.array([[1, 2, 3],
               [0, 0, 0],
               [-1, -1, -1]])
m2 = np.array([[4, 5, 6],
               [1, 1, 1],
               [3, 3, 3]])
mat = np.hstack((m1, m2))
mat
```

```
array([[ 1,  2,  3,  4,  5,  6],
       [ 0,  0,  0,  1,  1,  1],
       [-1, -1, -1,  3,  3,  3]])
```

Why Stacking?

- Sometimes you collect datasets from different sources, and you might want to concatenate them into one dataset.

Vertical Stacking

Key Variable	Variable A	Variable B	Variable C	Variable D
1	3.1	7.3	1	23
2	4.5	9.9	0	21
3	5.0	8.5	0	44
4	1.0	8.4	1	50

+

Key Variable	Variable A	Variable B	Variable C	Variable D
5	5.0	8.7	0	33
6	5.0	9.1	1	25
7	3.7	6.9	1	23
8	4.8	9.4	1	45

=

Key Variable	Variable A	Variable B	Variable C	Variable D
1	3.1	7.3	1	23
2	4.5	9.9	0	21
3	5.0	8.5	0	44
4	1.0	8.4	1	50
5	5.0	8.7	0	33
6	5.0	9.1	1	25
7	3.7	6.9	1	23
8	4.8	9.4	1	45

Horizontal Stacking

Key Variable	Variable A	Variable B	Variable C	Variable D
1	3.1	7.3	1	23
2	4.5	9.9	0	21
3	5.0	8.5	0	44
4	1.0	8.4	1	50

+

Key Variable	Variable E	Variable F	Variable G	Variable H
1	86	Red	4.9	19
2	95	Green	5.0	20
3	78	Red	5.0	14
4	91	Blue	4.1	13

=

Key Variable	Variable A	Variable B	Variable C	Variable D	Variable E	Variable F	Variable G	Variable H
1	3.1	7.3	1	23	86	Red	4.9	19
2	5.0	8.5	0	44	95	Green	5.0	20
3	5.0	8.5	0	44	78	Red	5.0	14
4	1.0	8.4	1	50	91	Blue	4.1	13

5. Comparison

Comparison using ‘==’:

- Is about elementwise comparison between two numpy arrays.
- The result is a Boolean numpy array with the same shape.
- Examples:

- **Compare an array to a scalar**

```
s = 4
v = np.array([2, 4, 3, 4, 10])
v == s
```

```
array([False,  True, False,  True, False])
```

- **Compare two vectors**

```
v1 = np.array([1, 2, 3, 4, 5])
v2 = np.array([2, 4, 3, 4, 10])
v1 == v2
```

```
array([False, False,  True,  True, False])
```

- **Compare two Matrices**

```
mat1 = np.array([[1, 2, 3],
                 [4, 2, 1]])
mat2 = np.array([[1, 4, 3],
                 [3, 2, 3]])
mat1 == mat2
```

```
array([[ True, False,  True],
       [False,  True, False]])
```

Comparison using all():

- Returns True if the all elements follow the condition

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

```
False
```

Comparison using any():

- Returns True if any element follows the condition

```
v = np.array([1, 2, 3, 4, 5])
np.any(v==1)
```

```
True
```

Comparison using isclose():

- It applies elementwise comparison with tolerance, where we check if every element is equal or close to its corresponding element in another array.
- In the following example, tolerance = 1.

```
v1 = np.array([1, 2, 3, 4, 5])
v2 = np.array([2, 3, 0, 5, 3])
np.isclose(v1, v2, atol=1)
```

```
array([ True,  True, False,  True, False])
```

Comparison using allclose():

- It applies comparison with tolerance, where we check if all elements in an array is equal or close to its corresponding element in another array.
- In the following example, tolerance = 1.

```python
v1 = np.array([1, 2, 3, 4, 5])
v2 = np.array([2, 3, 4, 5, 6])
np.allclose(v1, v2, atol=1)
```

```
True
```

6. Conditional Access & Modification

Conditional Access:

- It means getting elements of an array that satisfy a specific condition.
- A Condition means the result of elementwise comparison, we saw in the previous section.

Condition

```python
v = np.array([2, 4, 3, 4, 10])
condition = v >= 4
print(condition)
```

```
[False  True False  True  True]
```

Conditional Access

```python
v = np.array([2, 4, 3, 4, 10])
condition = v >= 4
conditional_access = v[condition]
print(conditional_access)
```

```
[ 4  4 10]
```

Conditional Modification using:

- It means applying modification to elements of an array that satisfy a specific condition.

Condition

```python
v = np.array([2, 4, 3, 4, 10])
condition = v >= 4
print(condition)
```

```
[False  True False  True  True]
```

Conditional Modification using ‘==’

```python
v = np.array([2, 4, 3, 4, 10])
condition = v >= 4
v[condition] = -1
print(v)
```

```
[ 2 -1  3 -1 -1]
```

Conditional Modification using where()

```python
v = np.array([2, 4, 3, 4, 10])
condition = v >= 4
new_arr = np.where(condition, -1, v)
print(new_arr)
```

```
[ 2 -1  3 -1 -1]
```

7. Array BroadCasting

Array BroadCasting:

- Is the application of arithmetic operations between arrays with a different shape.

Vector/Scalar BroadCasting

```
v1 = np.array([1, 4, 3])
c = 2
summation = v1 + c
multiplication = v1 * c
print(summation)
print(multiplication)
```

```
[3 6 5]
[2 8 6]
```

Matrix/Vector BroadCasting

```
Mat1 = np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9]])
v = np.array([.5, .2, .1])
Mat2 = Mat1 + v
print(Mat2)
```

```
[[1.5 2.2 3.1]
 [4.5 5.2 6.1]
 [7.5 8.2 9.1]]
```

8. Sparse Matrices

Sparse Matrix:

- Sparse matrix is a matrix that contain mostly zero values.
- The opposite of sparse matrix is dense matrix, which is a matrix where most of the values are non-zero.
- Sparse matrices has a problem related to waste of memory resources as those zero values do not contain any information.

Sparse Matrix:

- The solution is to transform it into another data structure. Where the zero values can be ignored.
- There are two techniques:

- **Dense Matrix to CSR Sparse Matrix**

```
from scipy.sparse import csr_matrix, csc_matrix
Mat = np.array([[1, 0, 0, 1, 0, 0],
                [0, 0, 2, 0, 0, 1],
                [0, 0, 0, 2, 0, 0]])

sMat = csr_matrix(Mat)
print(sMat)
```

```
(0, 0)        1
(0, 3)        1
(1, 2)        2
(1, 5)        1
(2, 3)        2
```

Dense Matrix to CSC Sparse Matrix

```
from scipy.sparse import csr_matrix, csc_matrix
Mat = np.array([[1, 0, 0, 1, 0, 0],
                [0, 0, 2, 0, 0, 1],
                [0, 0, 0, 2, 0, 0]])

sMat = csc_matrix(Mat)
print(sMat)
```

```
(0, 0)        1
(1, 2)        2
(0, 3)        1
(2, 3)        2
(1, 5)        1
```

- These two techniques are doing the same thing, but in different ways, which we are not interested in.

Thank You