

DS 440 Data Mining

Lecture 3: Numpy

1. Basics

1.1 Adding collections without using numpy

Data available in list format cannot be directly added pointwise. The + operator concatenates the two lists

```
In [5]: a = [1,2,3,4]
        b = [5,6,7,8,9]
        a+b
```

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

In order we achieve this we need to use the zip() function

```
In [6]: result = []
        for i,j in zip(a,b):
            result.append(i+j)
        result
```

```
Out[6]: [6, 8, 10, 12]
```

Numpy makes it very easy

1.2 Declaring an array

```
In [7]: import numpy as np
```

```
In [8]: a = np.array([1,2,3])
        a
```

```
Out[8]: array([1, 2, 3])
```

```
In [9]: type(a)
```

```
Out[9]: numpy.ndarray
```

```
In [10]: a.dtype
```

```
Out[10]: dtype('int64')
```

```
In [11]: f = np.array([1.2,1,2])  
f.dtype
```

```
Out[11]: dtype('float64')
```

```
In [ ]:
```

```
In [12]: f
```

```
Out[12]: array([1.2, 1. , 2. ])
```

```
In [14]: f[1]
```

```
Out[14]: 1.0
```

```
In [15]: f.ndim
```

```
Out[15]: 1
```

```
In [16]: f.shape
```

```
Out[16]: (3,)
```

```
In [17]: f.size
```

```
Out[17]: 3
```

1.3 Re-shape of an array

```
In [18]: a = np.array([1,2,3])
b = np.array([1,2,3])
print(a)
print(a.shape)
print(b)
print(b.shape)
```

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

```
In [19]: a.dot(b)
```

```
Out[19]: 14
```

```
In [33]: a1 = a.reshape(-1,1)
a1
```

```
Out[33]: array([[1],
               [2],
               [3]])
```

```
In [34]: a1.dot(b)
```

```
-----
-----
ValueError                                Traceback (most recent call
last)
Input In [34], in <cell line: 1>()
----> 1 a1.dot(b)

ValueError: shapes (3,1) and (3,) not aligned: 1 (dim 1) != 3 (dim 0)
```

```
In [35]: b1 = b.reshape(-1,1)
a1.dot(b1)
```

```
-----
ValueError                                Traceback (most recent call
last)
Input In [35], in <cell line: 2>()
      1 b1 = b.reshape(-1,1)
----> 2 a1.dot(b1)

ValueError: shapes (3,1) and (3,1) not aligned: 1 (dim 1) != 3 (dim 0
)
```

```
In [36]: b1 = b.reshape(1,-1)
a1.dot(b1)
```

```
Out[36]: array([[1, 2, 3],
               [2, 4, 6],
               [3, 6, 9]])
```

1.4 Back to elementwise addition

For the following a and f array, elementwise adding is straightforward

```
In [20]: a = np.array([1,2,3,4])
b = np.array([5,6,7,8])
a+b
```

```
Out[20]: array([ 6,  8, 10, 12])
```

```
In [21]: ## for concatenating a and b like for the list case
np.concatenate((a,b),axis = 0)
```

```
Out[21]: array([1, 2, 3, 4, 5, 6, 7, 8])
```

```
In [22]: from numpy import *
concatenate((a,b),axis = 0)
```

```
Out[22]: array([1, 2, 3, 4, 5, 6, 7, 8])
```

```
In [23]: import numpy as s
```

```
In [ ]: s.concatenate
```

```
In [ ]:
```

1.5 Numpy is way faster

```
In [71]: a = [i for i in range(1000000)]  
f = [i for i in range(1000000)]
```

```
In [72]: %%time  
summ = 0  
for i,j in zip(a,f):  
    summ = summ + i*j  
summ
```

CPU times: user 222 ms, sys: 8.49 ms, total: 231 ms
Wall time: 239 ms

```
Out[72]: 333332833333500000
```

```
In [73]: a = np.array([i for i in range(1000000)])  
f = np.array([i for i in range(1000000)])
```

```
In [74]: %%time  
summ = a.dot(f)  
summ
```

CPU times: user 1.42 ms, sys: 9 µs, total: 1.43 ms
Wall time: 1.45 ms

```
Out[74]: 333332833333500000
```

1.6 Other elementwise operations

```
In [25]: a = np.array([1,2,3])  
f = np.array([3,2,3])
```

```
In [26]: # multiplication  
a*f
```

```
Out[26]: array([3, 4, 9])
```

```
In [27]: ## division  
a/f
```

```
Out[27]: array([0.33333333, 1.          , 1.          ])
```

```
In [82]: ## raising to power  
a**f
```

```
Out[82]: array([ 1,  4, 27])
```

```
In [83]: ## multiplying a scalar  
a*10
```

```
Out[83]: array([10, 20, 30])
```

1.7 Universal function

Operate on a numpy array in an element by element fashion

```
In [91]: a = np.array([1,2,3])  
np.sin(a)  ## vectorized operation: faster than a loop
```

```
Out[91]: array([0.84147098, 0.90929743, 0.14112001])
```

```
In [92]: np.log(a)
```

```
Out[92]: array([0.          , 0.69314718, 1.09861229])
```

1.8 arange vs range

range is a standard python function. Arange is a similar function provided by numpy

range

```
In [151]: range(10)
```

```
Out[151]: range(0, 10)
```

```
In [152]: [i for i in range(10)]
```

```
Out[152]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [156]: [i for i in range(2,10)]
```

```
Out[156]: [2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [157]: [i for i in range(2,10,2)]
```

```
Out[157]: [2, 4, 6, 8]
```

```
In [167]: [i for i in range(0,1,0.2)]
```

```
-----  
-----  
TypeError                                Traceback (most recent call  
last)  
Input In [167], in <cell line: 1>()  
----> 1 [i for i in range(0,1,0.2)]  
  
TypeError: 'float' object cannot be interpreted as an integer
```

arange

```
In [159]: np.arange(10)
```

```
Out[159]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [160]: np.arange(2,10)
```

```
Out[160]: array([2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [161]: np.arange(2,10,2)
```

```
Out[161]: array([2, 4, 6, 8])
```

```
In [169]: np.arange(0,1,0.2)
```

```
Out[169]: array([0. , 0.2, 0.4, 0.6, 0.8])
```

time comparison

```
In [165]: %%time  
np.sin(np.arange(1000000))
```

CPU times: user 22.9 ms, sys: 5.53 ms, total: 28.5 ms
Wall time: 28.2 ms

```
Out[165]: array([ 0.          ,  0.84147098,  0.90929743, ...,  0.21429647,  
                -0.70613761, -0.97735203])
```

```
In [166]: %%time  
np.sin(range(1000000))
```

CPU times: user 137 ms, sys: 24.6 ms, total: 162 ms
Wall time: 238 ms

```
Out[166]: array([ 0.          ,  0.84147098,  0.90929743, ...,  0.21429647,  
                -0.70613761, -0.97735203])
```

```
In [ ]:
```

```
In [ ]:
```

2. Moving to 2 dimensions

```
In [28]: a = np.array([[1,2,4,4],[4,3,4,5],[5,3,2,5],[53,4,2,5]])  
a
```

```
Out[28]: array([[ 1,  2,  4,  4],  
                [ 4,  3,  4,  5],  
                [ 5,  3,  2,  5],  
                [53,  4,  2,  5]])
```

```
In [97]: print(a[1,1],a[2,3])
```

3 5

```
In [29]: a.shape
```

```
Out[29]: (4, 4)
```

```
In [30]: a.ndim
```

```
Out[30]: 2
```



```
In [32]: a.size
```

```
Out[32]: 16
```

2.1 Slicing

Slicing refers to accessing specific sections of the array through proper indices

```
In [33]: a = np.array([[1, 2, 3, 4, 5],  
                      [6, 7, 8, 9, 10]])  
a
```

```
Out[33]: array([[ 1,  2,  3,  4,  5],  
               [ 6,  7,  8,  9, 10]])
```

```
In [34]: a.shape
```

```
Out[34]: (2, 5)
```

```
In [35]: a[0,2:3]
```

```
Out[35]: array([3])
```

```
In [36]: a[0]
```

```
Out[36]: array([1, 2, 3, 4, 5])
```

```
In [127]: a[1].shape
```

```
Out[127]: (5,)
```

```
In [128]: b = a[1]  
b
```

```
Out[128]: array([ 6,  7,  8,  9, 10])
```

```
In [112]: ## 1 and 3 both indices are with respect to start of the array  
b[1:3]
```

```
Out[112]: array([7, 8])
```

```
In [38]: b
```

```
Out[38]: array([5, 6, 7, 8])
```

```
In [40]: ## b[lower:upper:step] -- step is optional  
b[0:-1:2]
```

```
Out[40]: array([5, 7])
```

```
In [114]: # start at index 0 and keep on going with an increment of 2  
b[0::2]
```

```
Out[114]: array([ 6,  8, 10])
```

```
In [115]: ## start to end with step of 2  
b[::2]
```

```
Out[115]: array([ 6,  8, 10])
```

```
In [116]: b[1:]
```

```
Out[116]: array([ 7,  8,  9, 10])
```

```
In [129]: ## 3rd last element to end  
b[-3:]
```

```
Out[129]: array([ 8,  9, 10])
```

```
In [130]: b[1:-2]
```

```
Out[130]: array([7, 8])
```

2.2 Striding while slicing

```
In [41]: c = np.array([[12, 3, 4, 5, 5],  
                       [ 4, 3, 4, 5, 6],  
                       [ 6, 4, 5, 6, 3],  
                       [ 6, 4, 6, 7, 4],  
                       [ 6, 4, 5, 7, 8]])  
c
```

```
Out[41]: array([[12, 3, 4, 5, 5],  
               [ 4, 3, 4, 5, 6],  
               [ 6, 4, 5, 6, 3],  
               [ 6, 4, 6, 7, 4],  
               [ 6, 4, 5, 7, 8]])
```

```
In [134]: c[0:-1:2,:]
```

```
Out[134]: array([[12,  3,  4,  5,  5],
                 [ 6,  4,  5,  6,  3]])
```

```
In [135]: c[:,0:-2:2]
```

```
Out[135]: array([[12,  4],
                 [ 4,  4],
                 [ 6,  5],
                 [ 6,  6],
                 [ 6,  5]])
```

```
In [136]: c[0::2,:]
```

```
Out[136]: array([[12,  3,  4,  5,  5],
                 [ 6,  4,  5,  6,  3],
                 [ 6,  4,  5,  7,  8]])
```

```
In [137]: c[0:-1:2,0:-2:2]
```

```
Out[137]: array([[12,  4],
                 [ 6,  5]])
```

2.3 Reshaping

```
In [140]: c.shape
```

```
Out[140]: (5, 5)
```

```
In [139]: d = c.reshape(1,25)
          d
```

```
Out[139]: array([[12,  3,  4,  5,  5,  4,  3,  4,  5,  6,  6,  4,  5,  6,  3,
                  6,
                  4,  6,  7,  4,  6,  4,  5,  7,  8]])
```

```
In [141]: d = c.reshape(10,2)
```

```
-----  
ValueError                                Traceback (most recent call  
last)  
Input In [141], in <cell line: 1>()  
----> 1 d = c.reshape(10,2)  
  
ValueError: cannot reshape array of size 25 into shape (10,2)
```

2.4 Modifying a numpy array

```
In [170]: a = np.array([1,2,3])  
print(a)  
a[0] = -1  
print(a)
```

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

```
In [171]: a[0] = 11.2  
print(a)
```

```
[11  2  3]
```

be careful while modifying arrays

```
In [172]: a = np.array([[1,2,3],[4,5,6],[7,8,9]])  
a
```

```
Out[172]: array([[1, 2, 3],  
                [4, 5, 6],  
                [7, 8, 9]])
```

```
In [173]: b = a[0,:]  
b
```

```
Out[173]: array([1, 2, 3])
```

```
In [174]: b[0] = -10  
b
```

```
Out[174]: array([-10,  2,  3])
```

```
In [175]: a
```

```
Out[175]: array([[ -10,   2,   3],
                 [   4,   5,   6],
                 [   7,   8,   9]])
```

Multiple view of the same arrays. To make a copy do the following

```
In [176]: a
```

```
Out[176]: array([[ -10,   2,   3],
                 [   4,   5,   6],
                 [   7,   8,   9]])
```

```
In [177]: b = a[0,:].copy()
          b
```

```
Out[177]: array([ -10,   2,   3])
```

```
In [ ]:
```

```
In [178]: b[0] = -20
          b
```

```
Out[178]: array([ -20,   2,   3])
```

```
In [179]: a
```

```
Out[179]: array([[ -10,   2,   3],
                 [   4,   5,   6],
                 [   7,   8,   9]])
```

Editing multiple values together

```
In [180]: a
```

```
Out[180]: array([[ -10,   2,   3],
                 [   4,   5,   6],
                 [   7,   8,   9]])
```

```
In [181]: a[0::2,0::2] = 1000
```

In [182]: a

Out[182]: array([[1000, 2, 1000],
[4, 5, 6],
[1000, 8, 1000]])

2.5 Fancy indexing

2.5.1 Indexing by position

In [43]: a = np.array([1,43,55,75,34545,453])
ind = [3,4]
a[ind]

Out[43]: array([75, 34545])

In [44]: c = np.array([[12,3,4,5,5],[4,3,4,5,6],[6,4,5,6,3],[6,4,6,7,4],[6,4,5,
c

Out[44]: array([[12, 3, 4, 5, 5],
[4, 3, 4, 5, 6],
[6, 4, 5, 6, 3],
[6, 4, 6, 7, 4],
[6, 4, 5, 7, 8]])

In [185]: ind1 = [1,3,4]
ind2 = [1,3,4]
c[ind1,ind2]

Out[185]: array([3, 7, 8])

2.5.2 Indexing by masking

In [45]: a = np.arange(5)
a

Out[45]: array([0, 1, 2, 3, 4])

In [46]: mask = np.array([1,0,0,1,0],dtype=bool)
a[mask]

Out[46]: array([0, 3])

```
In [188]: mask = a>1  
mask
```

```
Out[188]: array([False, False,  True,  True,  True])
```

```
In [189]: a[mask]
```

```
Out[189]: array([2, 3, 4])
```

We can also assign values

```
In [190]: a = np.array([10,20,-10,3,4,5,20])  
a
```

```
Out[190]: array([ 10,  20, -10,   3,   4,   5,  20])
```

```
In [191]: a[a<0] =0  
a
```

```
Out[191]: array([10, 20,  0,  3,  4,  5, 20])
```

2.5.3 Multiple dimension masking

```
In [192]: c
```

```
Out[192]: array([[12,  3,  4,  5,  5],  
                [ 4,  3,  4,  5,  6],  
                [ 6,  4,  5,  6,  3],  
                [ 6,  4,  6,  7,  4],  
                [ 6,  4,  5,  7,  8]])
```

```
In [194]: c[c[:,0]>5,:]
```

```
Out[194]: array([[12,  3,  4,  5,  5],  
                [ 6,  4,  5,  6,  3],  
                [ 6,  4,  6,  7,  4],  
                [ 6,  4,  5,  7,  8]])
```

```
In [195]: c[(c[:,0]>5) & (c[:,0]<10),:]
```

```
Out[195]: array([[6, 4, 5, 6, 3],  
                [6, 4, 6, 7, 4],  
                [6, 4, 5, 7, 8]])
```

```
In [196]: c>10
```

```
Out[196]: array([[ True, False, False, False, False],
                 [False, False, False, False, False],
                 [False, False, False, False, False],
                 [False, False, False, False, False],
                 [False, False, False, False, False]])
```

```
In [197]: a = c>10
          c[a]
```

```
Out[197]: array([12])
```

```
In [198]: c
```

```
Out[198]: array([[12,  3,  4,  5,  5],
                 [ 4,  3,  4,  5,  6],
                 [ 6,  4,  5,  6,  3],
                 [ 6,  4,  6,  7,  4],
                 [ 6,  4,  5,  7,  8]])
```

```
In [199]: c%3==0
```

```
Out[199]: array([[ True,  True, False, False, False],
                 [False,  True, False, False,  True],
                 [ True, False, False,  True,  True],
                 [ True, False,  True, False, False],
                 [ True, False, False, False, False]])
```

```
In [202]: c[c%3==0]
```

```
Out[202]: array([12,  3,  3,  6,  6,  6,  3,  6,  6,  6])
```

```
In [ ]:
```

```
In [ ]:
```

3 Higher dimensional arrays


```
In [207]: c = np.array([[[1,2,3],[4,5,6],[7,8,9],[13,14,15]], [[1,2,3],[4,5,6],[13,14,15]]])
```

```
Out[207]: array([[ 1,  2,  3],
                 [ 4,  5,  6],
                 [ 7,  8,  9],
                 [13, 14, 15]],

                [[ 1,  2,  3],
                 [ 4,  5,  6],
                 [10, 11, 12],
                 [16, 17, 18]])
```

```
In [209]: c.shape
```

```
Out[209]: (2, 4, 3)
```

```
In [210]: c[0]
```

```
Out[210]: array([[ 1,  2,  3],
                 [ 4,  5,  6],
                 [ 7,  8,  9],
                 [13, 14, 15]])
```

```
In [213]: c[0,1]
```

```
Out[213]: array([4, 5, 6])
```

4 Calculation with arrays

4.1 Broadcasting

```
In [214]: a = np.array([0,10,20,30])
          a.shape
```

```
Out[214]: (4,)
```

```
In [215]: b = np.array([0,1,2])
          b.shape
```

```
Out[215]: (3,)
```

```
In [216]: a1 = a.reshape(4,1)
          a1.shape
```

```
Out[216]: (4, 1)
```

```
In [235]: # a1: (4,1) and b is (3,)
          a1+b
```

```
Out[235]: array([[ 0,  1,  2],
                 [10, 11, 12],
                 [20, 21, 22],
                 [30, 31, 32]])
```

```
In [237]: b1 = b.reshape(1,3)
          b1.shape
```

```
Out[237]: (1, 3)
```

```
In [238]: # a1: (4,1) and b1 (1,3)
          a1+b1
```

```
Out[238]: array([[ 0,  1,  2],
                 [10, 11, 12],
                 [20, 21, 22],
                 [30, 31, 32]])
```

```
In [239]: a1.dot(b1)
```

```
Out[239]: array([[ 0,  0,  0],
                 [ 0, 10, 20],
                 [ 0, 20, 40],
                 [ 0, 30, 60]])
```

```
In [242]: ## scalar broadcasting over an array
          print(a)
          print(a+2)
```

```
[ 0 10 20 30]
[ 2 12 22 32]
```

4.2 Reduction operations

```
In [280]: c = np.array([[12,3,4,5,5],[4,3,4,5,6],[6,4,5,6,3],[6,4,6,7,4]])  
c
```

```
Out[280]: array([[12,  3,  4,  5,  5],  
                [ 4,  3,  4,  5,  6],  
                [ 6,  4,  5,  6,  3],  
                [ 6,  4,  6,  7,  4]])
```

```
In [228]: ## adds all the values in the array  
c.sum()
```

```
Out[228]: 102
```

```
In [244]: ## compress rows into 1 vector  
c.sum(axis=0)
```

```
Out[244]: array([28, 14, 19, 23, 18])
```

```
In [245]: ## compress columns into 1 vector  
c.sum(axis=1)
```

```
Out[245]: array([29, 22, 24, 27])
```

```
In [246]: c.sum(axis=-1)
```

```
Out[246]: array([29, 22, 24, 27])
```

Others

```
In [247]: c.min()
```

```
Out[247]: 3
```

```
In [248]: c.max()
```

```
Out[248]: 12
```

```
In [253]: c.mean()
```

```
Out[253]: 5.1
```

```
In [254]: c.ptp()
```

```
Out[254]: 9
```

And many others....

In []:

both of the following are okay

```
In [255]: ## method format  
a.sum()
```

```
Out[255]: 60
```

```
In [257]: ## function format  
np.sum(a)
```

```
Out[257]: 60
```

4.3 Other important functions

4.3.1 argmax and argmin

```
In [258]: c
```

```
Out[258]: array([[12,  3,  4,  5,  5],  
                [ 4,  3,  4,  5,  6],  
                [ 6,  4,  5,  6,  3],  
                [ 6,  4,  6,  7,  4]])
```

```
In [259]: c[[2,3],[3,4]]
```

```
Out[259]: array([6, 4])
```

```
In [260]: c.argmax(axis = 0)
```

```
Out[260]: array([1, 0, 0, 0, 2])
```

```
In [261]: c.argmax(axis = 1)
```

```
Out[261]: array([1, 1, 4, 1])
```

```
In [262]: c[c.argmax(axis = 0),np.arange(5)]
```

```
Out[262]: array([4, 3, 4, 5, 3])
```

```
In [146]: c.argmin(axis = 0),np.arange(5)
```

```
Out[146]: (array([1, 0, 0, 0, 2]), array([0, 1, 2, 3, 4]))
```

```
In [263]: c.argmin()
```

```
Out[263]: 1
```

```
In [264]: c[0,1] = 10  
c.argmin()
```

```
Out[264]: 6
```

```
In [265]: c
```

```
Out[265]: array([[12, 10,  4,  5,  5],  
                 [ 4,  3,  4,  5,  6],  
                 [ 6,  4,  5,  6,  3],  
                 [ 6,  4,  6,  7,  4]])
```

To better understand this

```
In [266]: np.unravel_index(c.argmin(),c.shape)  ## to unflatten the index in rec  
## c.argmin(): index to be resolved  
## c.shape: the shape to be resolved into
```

```
Out[266]: (1, 1)
```

Still a problem as there are multiple minima here

```
In [281]: c
```

```
Out[281]: array([[12,  3,  4,  5,  5],  
                 [ 4,  3,  4,  5,  6],  
                 [ 6,  4,  5,  6,  3],  
                 [ 6,  4,  6,  7,  4]])
```

```
In [282]: np.where(c == c.min())
```

```
Out[282]: (array([0, 1, 2]), array([1, 1, 4]))
```

```
In [283]: c[np.where(c == c.min())]  ## returns a tuple with as many elements as
```

```
Out[283]: array([3, 3, 3])
```

Can also use where for many other things

```
In [284]: c[np.where(c>=10)]  ## subsetting based on a rule
```

```
Out[284]: array([12])
```

4.3.2 Some other functions

```
In [285]: np.linspace(0,1,3)
```

```
Out[285]: array([0. , 0.5, 1. ])
```

```
In [286]: import numpy as np
f1 = np.array([5,3,2,5])
np.argsort(f1)
```

```
Out[286]: array([2, 1, 0, 3])
```

```
In [289]: f2 = np.array([[5,3,2,5],[1,2,3,4]])
f2
```

```
Out[289]: array([[5, 3, 2, 5],
                 [1, 2, 3, 4]])
```

```
In [290]: np.argsort(f2)
```

```
Out[290]: array([[2, 1, 0, 3],
                 [0, 1, 2, 3]])
```

```
In [291]: np.argsort(f2,axis=1)
```

```
Out[291]: array([[2, 1, 0, 3],
                 [0, 1, 2, 3]])
```

```
In [292]: np.argsort(f2,axis=0)
```

```
Out[292]: array([[1, 1, 0, 1],
                 [0, 0, 1, 0]])
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

