Lecture 14: Analyzing data V – More numpy

Topic – np.nan

NaN is short for **Not a number**. It is used to represent entries that are undefined. It is also used for representing missing values in a dataset. Sometimes when you collect data, there are errors that occur that prevent you from collecting data. For example, your sensor might fail (or partially fail), resulting in no data or data you know to be incorrect. Or maybe you were sick one day and failed to make observations in the field.

You also might make post-hoc analysis of your dataset and want to indicate that the data that you collected is incorrect. For example, maybe you made a field observation, but afterwords you found out that there were a large number of boats or other people in the area, making your observation biased. You might want to indicate that the data isn't trustworthy by changing it to NaN data.

np.nan is a built-in float object that represents NaN data:

```
>>> np.nan
nan
>>> type(np.nan)
<class 'float'>
```

Note that it is a float object – you cannot use np.nan in integer arrays.

```
>>> a
array([1, 2, 3, 4, 5])
>>> a[2] = np.nan
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot convert float NaN to integer
```

It can be pretty easy to change data to np.nan if it violates some criteria. For example, if data is more than some threshold value (> 5 or < 0), you might want to mask it out. To do this in numpy, you simply:

```
>>> a = np.array([1,5,-3,4,1,16.0, -30, 4])
>>> a
array([  1.,   5.,  -3.,   4.,   1.,  16., -30.,   4.])
>>> a[(a<0) | (a > 5)] = np.nan
>>> a
array([ 1.,  5., nan,  4.,  1., nan, nan,  4.])
```

np.nan also has the unusual feature that it is not equal to itself.

```
>>> np.nan == np.nan
False
```

This means you must use special functions to test whether a data value is equal to np.nan.

```
>>> np.isnan(np.nan)
True
>>> a = np.array([1,2,3.0,np.nan,5])
>>> a
array([ 1.,  2.,  3., nan,  5.])
>>> a == 1
array([ True, False, False, False, False])
>>> a == np.nan
array([False, False, False, False, False])
>>> np.isnan(a)
array([False, False, False,  True, False])
>>> ~np.isnan(a)
array([ True,  True,  True, False,  True])
```

Finally, when you use summary functions on data that contains NaN values, you also need to use special functions, as the normal summary functions cannot handle np.nan values (or at least they simply return np.nan). For example:

```
>>> a = np.array([1,5,-3,4,1,16.0, -30, 4])
>>> a[(a<0) | (a > 5)] = np.nan
>>> a
array([ 1.,  5., nan,  4.,  1., nan, nan,  4.])
>>> np.mean(a)
nan
>>> a.mean()
nan
>>> np.nanmean(a)
3.0
```

Topic – Concatenating/stacking arrays

Often you might want to combine data from multiple arrays. There are a number of different built in functions that can do this for you, but it's important to understand the distinctions between them. In this lecture, we will cover 1 approach to concatenate data by creating and replacing an array, as well as two functions that can do this for you.

Approach 1 – Create a new array and replace the data with the existing data. If you know ahead of time how large the data will be, you can create a new array large enough to hold all the existing data, initially setting the data as zero.

1D example: combine two 1D arrays of shape (4,) and (4,) to create a 2D array of shape (4,2):

```
>>> a = np.array([5,4,3,6])
>>> b = np.array([2,3,4,1])
>>> a.shape
(4,)
>>> b.shape
(4,)
>>> c = np.zeros(shape = (4,2))
>>> c
array([[0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.]])
>>> c[0]
array([0., 0.])
>>> c[:,0]
array([0., 0., 0., 0.])
>>> c[:,0] = a.copy()
>>> c[:,1] = b.copy()
>>> c
array([[5., 2.],
       [4., 3.],
       [3., 4.],
       [6., 1.]])
>>> c.shape
(4, 2)
```

Approach 2 – Combine new data using the np.concatenate function. This function works well to concatenate data along existing axis, but cannot concatenate along a new axis. For example:

```
>>> a
array([5, 4, 3, 6])
>>> b
array([2, 3, 4, 1])
>>> np.concatenate([a,b])
```

```
array([5, 4, 3, 6, 2, 3, 4, 1])
>>> np.concatenate([a,b], axis = 0)
array([5, 4, 3, 6, 2, 3, 4, 1])
>>> np.concatenate([a,b], axis = 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in concatenate
numpy.AxisError: axis 1 is out of bounds for array of dimension 1
>>> c = np.array([[1,2,3],[4,5,6]])
>>> d = np.array([[6,5,4],[3,2,1]])
>>> np.concatenate([c,d], axis = 0)
array([[1, 2, 3],
       [4, 5, 6],
       [6, 5, 4],
       [3, 2, 1]])
>>> np.concatenate([c,d], axis = 1)
array([[1, 2, 3, 6, 5, 4],
       [4, 5, 6, 3, 2, 1]])
>>> np.concatenate([c,d], axis = 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in concatenate
numpy.AxisError: axis 2 is out of bounds for array of dimension 2
```

Approach 3 – Combine new data along a new axis using the np.stack function. This function works well to concatenate data along a new axis. For example:

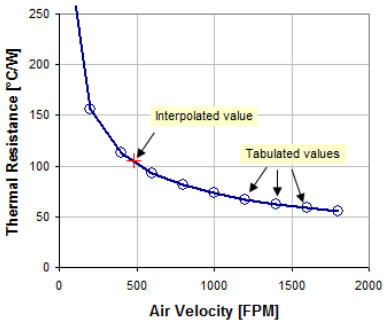
```
>>> a
array([5, 4, 3, 6])
>>> b
array([2, 3, 4, 1])
>>> np.stack([a,b])
array([[5, 4, 3, 6],
       [2, 3, 4, 1]])
>>> c
array([[1, 2, 3],
       [4, 5, 6]])
>>> d
array([[6, 5, 4],
       [3, 2, 1]])
>>> np.stack([c,d])
array([[[1, 2, 3],
        [4, 5, 6]],

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

Topic – Interpolation

The NumPy library has a large number of functions to perform data analysis. One useful example of this is interpolation. Interpolation is used when you want to estimate the value of data that you don't have observations for. For example, let's say you measure the temperature of an area every other day (i.e. the 1st, 3rd, 5th, 7th, etc). You might want to know the temperature on one of the days that you did not measure it (e.g. the 2nd). Interpolation allows you to estimate the temperature from other values:

Air Velocity	Thermal Resistance
0	373.0
200	156.1
400	113.6
600	93.1
800	81.5
1000	73.7
1200	67.2
1400	62.4
1600	58.3
1800	55.0



NumPy library provides a function for interpolating data.

```
numpy.interp(x, xp, fp, left=None, right=None, period=None)
```

x = The x values of where you want to estimate new values (e.g. the 2nd)

xp = The x values of the 'good' data (eg. the (1st, 3rd, 5th, 7th, etc))

fp = The y values of the 'good' data (e.g. (55F, 61F, 70F, 66F, 48F))

Let's see this in action. First we will create two arrays, one containing the x values and one containing the y values:

```
>>> x = np.arange(0,1,0.1)
>>> x
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
>>> y = np.sin(x)
>>> y
array([0.        , 0.09983342, 0.19866933, 0.29552021, 0.38941834,
       0.47942554, 0.56464247, 0.64421769, 0.71735609, 0.78332691])
```

Now if you want to estimate the value at one of the intermediate values, you can like so:

```
>>> np.interp(.55, x, y)
0.5220340059996192
>>> np.interp([0.05, 0.15, .55], x, y)
array([0.04991671, 0.14925137, 0.52203401])
```

Interpolation is also useful to replace bad data (e.g. nan data) with real values. Let's modify the y array so a few of the values are NaN:

```
>>> y[4] = np.nan
>>> y[6] = np.nan
>>> y
array([0.        , 0.09983342, 0.19866933, 0.29552021,        nan,
       0.47942554,        nan, 0.64421769, 0.71735609, 0.78332691])
```

Interpolation is also useful to replace bad data (e.g. nan data) with real values. Let's modify the y array so a few of the values are NaNs:

```
>>> y[4] = np.nan
>>> y[6] = np.nan
>>> y
array([0.        , 0.09983342, 0.19866933, 0.29552021,        nan,
       0.47942554,        nan, 0.64421769, 0.71735609, 0.78332691])
```

If we want to 'fix' these values we can use interpolation to do so.

1st – Get the x values of the bad data

```
>>> x[np.isnan(y)]
array([0.4, 0.6])
```

2nd – Get the x values of the good data

```
>>> x[~np.isnan(y)]
array([0. , 0.1, 0.2, 0.3, 0.5, 0.7, 0.8, 0.9])
```

3rd – Get the y values of the good data

```
>>> y[~np.isnan(y)]
array([0.        , 0.09983342, 0.19866933, 0.29552021, 0.47942554,
       0.64421769, 0.71735609, 0.78332691])
```

4th – Interpolate

```
>>> np.interp(x[np.isnan(y)], x[~np.isnan(y)], y[~np.isnan(y)])
array([0.38747287, 0.56182161])
>>> y[np.isnan(y)] = np.interp(x[np.isnan(y)], x[~np.isnan(y)], y[~np.isnan(y)])>>> y
array([0.        , 0.09983342, 0.19866933, 0.29552021, 0.38747287,
       0.47942554, 0.56182161, 0.64421769, 0.71735609, 0.78332691])
```