
Instructors: Parth Shah, Riju Pahwa

Lecture 3 Notes

Outline

1. Numpy
    - Introduction
    - Functions and Matrix Manipulation
2. Libraries
    - Theano
    - Tensor Flow
    - Keras
    - Caffe
    - Nodal Datasets

Numpy

Introduction

Its convenient to write machine learning applications in Python, but when working with large datasets that comes at a cost. Python is extremely slow compared to other languages. For this reason, most machine learning applications written in Python use a library known as numpy for the data manipulation. Numpy is written in C so it runs far faster than python and proves highly advantageous for speeding up applications written in Python.

We will start with the basics. The main data structures we will use are numpy arrays and numpy matrices. Numpy matrices are standard matrices which implies in terms of organization they are 2 dimensions (note this is different from the actual dimension of the data vectors these matrices transform). One dimension tells us the number of rows and the other tells us the number of columns. Numpy arrays are not restricted to 2 dimensions. Its easiest to see the difference through example.

Below is a python shell instance. For convenience 'l' is a list object, 'm' is a numpy matrix, and 'a' is a numpy array. The number of dimensions is given by the number of occurrences of the letter. Note how numpy throws an error when it tries to cast the numpy matrix type to the three dimensional list 'lll'.

```
>>> l = [1,2,3,4,5]
>>> ll = [l,l,l]
>>> l
[1, 2, 3, 4, 5]
>>> ll
[[1, 2, 3, 4, 5], [1, 2, 3, 4, 5], [1, 2, 3, 4, 5]]
>>> lll = [ll,ll,ll,ll]
>>> lll
[[[1, 2, 3, 4, 5], [1, 2, 3, 4, 5], [1, 2, 3, 4, 5]], [[1, 2, 3, 4, 5], [1, 2, 3, 4, 5]
, [1, 2, 3, 4, 5]], [[1, 2, 3, 4, 5], [1, 2, 3, 4, 5], [1, 2, 3, 4, 5]], [[1, 2, 3, 4,
5], [1, 2, 3, 4, 5], [1, 2, 3, 4, 5]]]
>>> m = np.matrix(l)
>>> mm = np.matrix(ll)
>>> m
matrix([[1, 2, 3, 4, 5]])
>>> mm
matrix([[1, 2, 3, 4, 5],
        [1, 2, 3, 4, 5],
        [1, 2, 3, 4, 5]])
>>> mmm = np.matrix(lll)

Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    mmm = np.matrix(lll)
  File "/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/n
umpy/matrixlib/defmatrix.py", line 261, in __new__
    raise ValueError, "matrix must be 2-dimensional"
ValueError: matrix must be 2-dimensional
>>> a = np.array(l)
>>> aa = np.array(ll)
>>> a
array([1, 2, 3, 4, 5])
>>> aa
array([[1, 2, 3, 4, 5],
       [1, 2, 3, 4, 5],
       [1, 2, 3, 4, 5]])
>>> aaa = np.array(lll)
>>> aaa
array([[[1, 2, 3, 4, 5],
        [1, 2, 3, 4, 5],
        [1, 2, 3, 4, 5]],

       [[1, 2, 3, 4, 5],
        [1, 2, 3, 4, 5],
        [1, 2, 3, 4, 5]],

       [[1, 2, 3, 4, 5],
        [1, 2, 3, 4, 5],
        [1, 2, 3, 4, 5]],

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

We will see that the choice between using arrays and matrices depends on the operations we plan on conducting on the data. Matrices are convenient for the general matrix operations, but the numpy arrays are far more useful for element wise operations. Furthermore they are also not restricted to the 2 dimensional stucture of matrices.

Some standard matrices we can quickly build are ones array (all ones), zeros array (all zeros), and identity matrices.

Below is a python shell snapshot. Notice to build the ones matrix we use the function np.ones() and pass the shape as an argument. If we wanted a 2 by 3 by 4 ones array np.ones((2,3,4)). Zeros arrays are built similarly using the np.zeros() function. Note creating identity matrices is different because they must be two dimensional and square. Also notice how computation on numpy arrays work. All operations are elementwise.

```
>>> ones = np.ones((3,5))
>>> ones
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])
>>> zeros = np.zeros((3,5))
>>> zeros
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> zeros + ones
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])
>>> zeros*ones
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> zeros**2
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> (2*ones)**2
array([[ 4.,  4.,  4.,  4.,  4.],
       [ 4.,  4.,  4.,  4.,  4.],
       [ 4.,  4.,  4.,  4.,  4.]])
>>> np.eye(5)
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
>>> np.identity(5)
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
```

Another two array initializations that are useful are np.arange() which creates a 1-dimensional numpy array. This works exactly like range() in python. Another is np.diag() which makes a diagonal matrix out of an input data array. Python shell snapshot shown below (we will explain reshape() in the matrix manipulation section):

```
>>> np.arange(10).reshape((2,5))
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> l = [1,2,3,4,5]
>>> np.diag(l)
array([[1, 0, 0, 0, 0],
       [0, 2, 0, 0, 0],
       [0, 0, 3, 0, 0],
       [0, 0, 0, 4, 0],
       [0, 0, 0, 0, 5]])
```

In summary the basic way to create numpy arrays is to create a list and then just type cast. However, there are faster ways to create some simple matrices which proves to be extremely useful.

Functions and Matrix Manipulation

The key functions for matrix manipulations are reshape, vstack, hstack, tile, and repeat.

Reshape is simple. Basically just think of this function as remolding clay. The key is to make sure the shape makes sense. Because if the number of entries is not matching it will throw an error. Also if you want to reshape it into 3 dimensions or order the way its reshaped you need to pass in axis terms. This is something you can experiment with more based on the specific application but the function itself is useful for many numpy implementations of machine learning algorithms.

Numpy repeat is also a straightforward but useful function. It basically repeats the array based on how many times you want it to repeat and along which axis.

```
>>> npb = np.array(b)
>>> npb
array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]])
>>> npbb = npb.repeat(2)
>>> npbb
array([1, 1, 2, 2, 3, 3, 1, 1, 2, 2, 3, 3, 1, 1, 2, 2, 3, 3])
>>> npbb2 = npb.repeat(2, axis = 0)
>>> npbb2
array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3],
       [1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]])
>>> npbb3 = npb.repeat([1,2,3], axis = 1)
>>> npbb3
array([[1, 2, 2, 3, 3, 3],
       [1, 2, 2, 3, 3, 3],
       [1, 2, 2, 3, 3, 3]])
```

Numpy tile works like it sounds. It basically treats your starting array as a tile and use those tiles to create a new numpy array.

```
>>> npbt = np.tile(npb, 2)
>>> npbt
array([[1, 2, 3, 1, 2, 3],
       [1, 2, 3, 1, 2, 3],
       [1, 2, 3, 1, 2, 3]])
>>> npbt = np.tile(npb,(3,1))
>>> npbt
array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3],
       [1, 2, 3],
       [1, 2, 3],
       [1, 2, 3],
       [1, 2, 3],
       [1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]])
>>> a = np.arange(9).reshape((3,3))
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> at = np.tile(a,(3,2))
>>> at
array([[0, 1, 2, 0, 1, 2],
       [3, 4, 5, 3, 4, 5],
       [6, 7, 8, 6, 7, 8],
       [0, 1, 2, 0, 1, 2],
       [3, 4, 5, 3, 4, 5],
       [6, 7, 8, 6, 7, 8],
       [0, 1, 2, 0, 1, 2],
       [3, 4, 5, 3, 4, 5],
       [6, 7, 8, 6, 7, 8]])
```

Numpy vstack and hstack are the final functions. They stack vertically and horizontally. So if you have two different numpy arrays and want to stack them these are the functions to use. However, I find repeat and tile more useful.

Masking

Finally there is masking. Basically you pass in a condition and mask the array based on a condition. This means that if you take averages of the row it only takes in not masked values into account.

Here's how it looks:

```
>>> a = np.array([[1,2,4],[5,-1,0]])
>>> a
array([[ 1,  2,  4],
       [ 5, -1,  0]])
>>> m = np.ma.masked_where(a > 3, a)
>>>
>>> m
masked_array(data =
 [[1 2 --]
 [-- -1 0]],
             mask =
 [[False False  True]
 [ True False False]],
       fill_value = 999999)
```

Libraries

Theano

Theano is a deep learning library. For working with neural networks it proves to be useful. However, writing in Theano requires using Theano's own objects. For most purposes I recommend using Keras with Theano instead of just straight Theano if you are working with neural networks.

Tensor Flow

Tensor Flow is a powerful general purpose Machine Learning library by Google. The resources section contains more of what you can do with Tensor Flow.

Keras

An awesome wrapper that goes on top of Theano and TensorFlow. Keras makes using neural networks extremely easy. The resources section contains more of what you can do with Keras.

Caffe

Caffe is a library for deep learning. It is popular for Convolutional Neural Networks due to its speed. The resources section contains more of what you can do with Caffe.

OpenCV

OpenCV, as per its name, is a popular library for computer vision. The resources section contains more of what you can do with openCV.

Nodal Datasets

The available datasets for this class are Boston crime data and Boston collision data. In this class you will also have access to the Nodal Connect API if needed. Information about Nodal can be found at www.nodal.co. The resources section contains more of what you can do with Nodal.
