CS 482/682 MACHINE LEARNING

NUMPY TUTORIAL

Go through the tutorial and complete all exercises.
Make a single python file with ALL of the solution.

For each Exercise, add comment line as follows
<A Blank Line>
# Code for Exercise 1
(Followed by the code)
<A Blank Line>
# Code for Exercise 2
(Followed by the code)
etc
etc

All of the outputs must be labelled as shown the examples. In the output make sure the printout contains

<A Blank Line>
Exercise 1
(Followed by the results of Exercise 1 all appropriately labelled)
<A Blank Line>
Exercise 2
(Followed by the results of Exercise 2 all appropriately labelled)
tc
etc

Turn in two files:
one python file called <lastnameNumPyTutorial.py>
and
one text file called <lastnameNumPyTutorialResults>.txt

Any code that says "Run the following code and observe the printed results" need NOT be turned in.

Introduction

This tutorial will provide an introduction to NumPy and its uses. You must explore various additional functions within NumPy throughout the term.

We will be working on this tutorial using the *Spyder through Anaconda*.

More documentation on NumPy can be found from the [NumPy website]()

Notation

In these notes `this font` indicates something that appears on a computer screen or which should be typed in to the computer and *`this font`* indicates where you should substitute something. For example:

```
edit filename
```

means type edit followed by the name of a file. Comments in square brackets are asides or explanatory notes often these are intended for experts who might want to know more about the subject.

What to do if you are stuck in an exercise

Try rereading the previous bits of the worksheet or consult NumPy documentation.

Installing NumPy

NumPy should be installed when Anaconda and its packages are installed. So should have to nothing additional to get NumPy.

To check what version of NumPy you have, type the following into the command line at the Spyder's IPython Console:

```
pip show numpy
```

You should see version. Otherwise it can be installed in the command line with the following:

```
pip install numpy
```

Introduction to Arrays

NumPy's main object is the homogeneous multidimensional **array**. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of non-negative integers. In NumPy dimensions are called **axes**.

For example, the coordinates of a point in 3D space [1, 2, 1] has one axis. That axis has 3 elements in it, so we say it has a length of 3. In the example pictured below, the array has 2 axes. The first axis has a length of 2, the second axis has a length of 3.

```
 [[ 1., 0., 0.],
 [ 0., 1., 2.]]
```

NumPy's array class is called **ndarray**. It is also known by the alias array. Note that **numpy.array** is not the same as the Standard Python Library class **array.array**, which only handles one-dimensional arrays and offers less functionality.

1. Creating Arrays

There are several ways to create arrays. For example, you can create an array from a regular Python list or tuple using the array function. The type of the resulting array is deduced from the type of the elements in the sequences. The object **ndarray.dtype** describes the type of the elements in the array.

Run the following code and observe the printed results (no need to turn anything in for this)

```
import numpy as np
a = np.array([2,3,4])
print("\nArray A:\n",a)
print("\nArray A Data Type:\n", a.dtype)

b = np.array([1.2, 3.5, 5.1])
print("\nArray B Data Type:\n", b.dtype)
```

Exercise 1:
Correct the following line.
`e1 = np.array(1,2,3,4)`
Print the array and data type.

The function **zeros** creates an array full of zeros, the function **ones** creates an array full of ones, and the function **empty** creates an array whose initial content is random and depends on the state of the memory. By default, the dtype of the created array is **float64**.

Run the following code and observe the printed results: (no need to turn anything in for this)

```
c = np.zeros((2,3))
```

```
d = np.ones((3,4))
e = np.empty((9))

print("\nArray C:\n",c)
print("\nArray C Data Type:\n", c.dtype)
print("\nArray D:\n",d)
print("\nArray D Data Type:\n", d.dtype)
print("\nArray E:\n",e)
print("\nArray E Data Type:\n", e.dtype)
```

Exercise 2:
Create a 2X7 array of zeros. Name the array **b**.
Print the array.

To create sequences of numbers, NumPy provides the **arange** function, which is analogous to the Python built-in **range**, but returns an array. Arange function takes `beginning`, `end` and `step` as arguments. *The last value included in the array is less than the end.* The function **linspace** receives beginning, end, and the number of elements that we want, instead of the step. It spaces them out equally

<u>Run the following code and observe the printed results:</u>

```
f = np.arange( 10, 30, 5 )
g = np.linspace( 0, 2, 9 )

print("\nArray f:\n",f)
print("\nArray f Data Type:\n", f.dtype)
print("\nArray g:\n",g)
print("\nArray gData Type:\n", g.dtype)
```

Exercise 3:
Use either **arrange** or **linspace** to create an array that includes the values between 1 and 23 in increments of 2.5. Name the array **c**.
Print the array and data type.

Multidimensional arrays can have one index per axis. These indices are given in a tuple separated by commas

<u>Run the following code and observe the printed results:</u>

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

print("\nArray H:\n",h)
print("\nValue at (1,1):\n",h[1,1])
```

Shape Manipulation

The shape of an array can be changed with various commands. Note that the following three commands all return a modified array, but do not change the original array:

- **ndarray.ravel():** returns the array, flattened to shape (n,1)
- **ndarray.reshape(int, int):** returns the array with a modified shape
- **ndarray.T:** returns the array, transposed

Run the following code and observe the printed results:

```
i = np.array([[1,2,3,4],
              [5,6,7,8],
              [9,10,11,12]])

print("\nArray I:\n",i)
print("\nArray I - ravel:\n",i.ravel())
print("\nArray I - reshaped 2X6:\n",i.reshape(2,6))
print("\nArray I - transposed:\n",i.T)
print("\nArray I Shape:\n",i.shape)
print("\nArray I.T Shape:\n",i.T.shape)
```

Exercise 4:
Create an array using `arange` that resembles the following table of values. Name the array **d**.
1 … 10
11 … 20
21 … 30
31 … 40
Reshape the array so that there are 5 rows and 8 columns.
Print the value at (5,8) of the reshaped array.(I,e 5th row – index 4 and 8th column index 7)

Other important attributes of an **ndarray** object are:

- **ndarray.ndim**
  - the number of axes (dimensions) of the array.
- **ndarray.size**
  - the total number of elements of the array
- **ndarray.itemsize**
  - the size in bytes of each element of the array.

Run the following code and observe the printed results:

```
j = np.arange(15).reshape(3, 5)

print("\nArray J:\n",j)
print("\nArray J Size:\n", j.shape)
print("\nArray J Dimensions:\n", j.ndim)
print("\nArray J Data Type:\n", j.dtype.name)
print("\nArray J Item Size:\n", j.itemsize)
print("\nArray J Size:\n", j.size)
print("\nArray J Type:\n", type(j))
```

This should give you an understanding of important **ndarray** attributes. You will need to recall these attributes when creating and modifying arrays.

Exercise 5:
Create a NumPy array called `arr` whose `ndim, shape, size and dtype` are
3
(3, 3, 2)
18
int32

2. Basic Operations on arrays

Arithmetic operators on arrays apply *elementwise*. A new array is created and filled with the result.
Run the following code and observe the printed results:

```
a = np.array([20, 30, 40, 50])
b = np.arange(4)
c = a – b
print("\n Array C\n", c)
print(("\n Array B**2\n", b**2)
print(("\n Array 10*sin(A)\n", 10*np.sin(a))
print(("\n a<35\n", a<35)
```

Unlike in many matrix languages, the product operator * *operates elementwise* in NumPy arrays.

The *matrix product* can be performed using the @ operator (in python >=3.5) or the dot function or method.

Universal Functions

NumPy provides familiar mathematical functions such as sin, cos, and exp. In NumPy, these are called "universal functions" (`ufunc`). Within NumPy, these functions operate elementwise on an array, producing an array as output.

Run the following code and observe the printed results:

```
n = np.array([[1,1],[0,1]])
o = np.array([[2,0],[3,4]])
print("\nProduct N * O:\n",n*o)
print("\nMatrix Product N @ O:\n",n@o)
print("\nMatrix Product N.dot(O):\n",n.dot(o))
```

Many unary operations, such as computing the sum of all the elements in the array, are implemented as methods of the ndarray class.

See also:

`all`, `any`, `apply_along_axis`, `argmax`, `argmin`, `argsort`, `average`, `bincount`, `ceil`, `clip`, `conj`, `corrcoef`, `cov`, `cross`, `cumprod`, `cumsum`, `diff`, `dot`, `floor`, `inner`, `invert`, `lexsort`, `max`, `maximum`, `mean`, `median`, `min`, `minimum`, `nonzero`, `outer`, `prod`, `re`, `round`, `sort`, `std`, `sum`, `trace`, `transpose`, `var`, `vdot`, `vectorize`, `where`

Run the following code and observe the printed results:*describe the example and give output*

```
import numpy.random as rg
p = rg.random((2,3))
print("\nArray P:\n",p)
print("\nSum of P:\n",p.sum())
print("\nMin of P:\n",p.min())
print("\nMax of P:\n",p.max())
print("\nexp of P:\n",p.exp())
print("\nSqrt of P:\n",p.sqrt())
print("\nAdd of P:\n",p+p)
```

By default, these operations apply to the array as though it were a list of numbers, regardless of its shape. However, by specifying the axis parameter you can apply an operation along the specified axis of an array:

Run the following code and observe the printed results:

```
r = np.arange(6).reshape(2,3)
print("\nArray R:\n",r)
print("\nColumn Sum of R:\n",r.sum(axis=0))
print("\nRow Sum of R:\n",r.min(axis=1))
print("\nCumulative Row Sum of R:\n", r.cumsum(axis=1))
```

Exercise 6: Create an array m and p of shape (4,4) with random numbers between 0 to 1. Compute the following

sin(m) @ cos(p) + k

where is k is column vector with sum of each row of m.

3. Adding, removing, and sorting elements

Sorting an element is simple with np.sort(). You can specify the axis, kind, and order when you call the function. If you start with this array:

Run the following code and see the printed results:

You can quickly sort the numbers in ascending order.

<u>Run the following code and observe the printed results:</u>

```
arr = np.array([2, 1, 5, 3, 7, 4, 6,8])
print( “\n Array arr sorted\n”, np.sort(arr))
```

Concatenating is done similarly.
<u>Run the following code and observe the printed results:</u>

```
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])
print(“\n Array A and B concatenated\n”, np.concatenate((a, b)))
```

You can concatenate them with np.concatenate().

<u>Run the following code and observe the printed results:</u>

```
x = np.array([[1, 2], [3, 4]])
y = np.array([[5, 6]])
print("\n Array X and Y Concatenated\n", np.concatenate((x, y),
axis=0))
```

```
k = np.array([2, 1, 5, 3, 7, 4, 6, 8])
print("\nArray K:\n",k)
print("\nArray K - sorted:\n",np.sort(k))

l = np.array([1, 2, 3, 4])
m = np.array([5, 6, 7, 8])
print("\nArray L:\n",l)
print("\nArray M:\n",m)
print("\nArray L+M:\n",np.concatenate((l, m)))

#add comment: concatenated 2 dim…
n = np.array([[1, 2], [3, 4]])
o = np.array([[5, 6], [7, 8]])
print("\nArray N:\n",n)
print("\nArray O:\n",o)
print("\nArray N+O (by col):\n",np.concatenate((n, o),
axis=0))
print("\nArray N+O (by row):\n",np.concatenate((n, o),
axis=1))
```

Exercise 7:
Create an array`f1` containing random 10 elements. Sort the array and print the smallest and largest by accessing first and last element. Reshape array into `f2` to have two rows. Concatenate two copies of `f2`. Print the results

4. Indexing, Slicing and Iterating

One-dimensional arrays can be indexed, sliced and iterated over, much like lists and other Python sequences.

```
a = np.arange(10)**3
print("\n Array A\n", a)
print("\n Array A columns 2 to 5\n", a[2:5])


#from start to position 6, exclusive, set every 2nd element to 1000
a[:6:2] = 1000
print("\n Array a start to position 6, every second element is 1000\n", a)

a[::-1]  # reversed a
```

```
print("\n Array A reveresed\n", a)

for i in a:
  print(("\n Array a's elements times 1/3\n",i**(1 / 3.))
```

Multidimensional arrays can have one index per axis. These indices are given in a tuple separated by commas:

```
 def f(x, y):
     return 10 * x + y


# fromfunction - Constructs an array by executing a function over each
#coordinate.
 b = np.fromfunction(f, (5, 4), dtype=int)
 print((("\n Array B\n",b)
 print(b[2, 3])
 print("each row in the second column of b :\n",b[0:5, 1])
# each row in the second column of b
 print(b[:, 1])    # equivalent to the previous example
 print(b[1:3, :])  # each column in the second and third row of b
```

When fewer indices are provided than the number of axes, the missing indices are considered complete slices:

```
 b[-1]   # the last row. Equivalent to b[-1, :]
```

The expression within brackets in `b[i]` is treated as an `i` followed by as many instances of `:` as needed to represent the remaining axes. NumPy also allows you to write this using dots as `b[i, ...]`.

The **dots** (`...`) represent as many colons as needed to produce a complete indexing tuple. For example, if `x` is an array with 5 axes, then

- `x[1, 2, ...]` is equivalent to `x[1, 2, :, :, :]`,
- `x[..., 3]` to `x[:, :, :, :, 3]` and
- `x[4, ..., 5, :]` to `x[4, :, :, 5, :]`.

```
 c = np.array([[[  0,  1,  2],  ...,[ 10, 12, 13]],...,[[100, 101, 102],...
,[110, 112, 113]]])
 print(c.shape)
 print(c[1, ...])  # same as c[1, :, :] or c[1]
 print(c[..., 2])  # same as c[:, :, 2]
```

Iterating over multidimensional arrays is done with respect to the first axis:

```
 for row in b:
    print(row)
```

However, if one wants to perform an operation on each element in the array, one can use the flat attribute which is an iterator over all the elements of the array:

```
 for element in b.flat:
      print(element)
```

Exercise 8: Use indexing and slicing for each task below:

```
A1 = np.arange(10)
print(A1)
```

Print the first and the last element. Print each element with even index. Print the first four elements. Print the last three elements. Print element four to eight.

```
np.random.seed(42)
M1 = 100*np.random.rand(9,7).round(2)
print(M1)
```

Print the first and the last row with one slice operation. Print all elements less than 10.Print each even row. Print each odd column. Print the elements of the even rows and the odd columns. Print all elements from the following matrix with a pair of even indices ([0,2],[2, 2], [4,2],.. etc.).

5. Functions and Methods Overview

Here is a list of some useful NumPy functions and methods names ordered in categories. See Routines for the full list.

Array Creation

arange, array, copy, empty, empty_like, eye, fromfile, fromfunction, identity, linspace, logspace, mgrid, ogrid, ones, ones_like, r_, zeros, zeros_like

Conversions

ndarray.astype, atleast_1d, atleast_2d, atleast_3d, mat

Manipulations

array_split, column_stack, concatenate, diagonal, dsplit, dstack, hsplit, hstack, ndarray.item, newaxis, ravel, repeat, reshape, resize, squeeze, swapaxes, take, transpose, vsplit, vstack

Questions

`all`, `any`, `nonzero`, `where`

Ordering

`argmax`, `argmin`, `argsort`, `max`, `min`, `ptp`, `searchsorted`, `sort`

Operations

`choose`, `compress`, `cumprod`, `cumsum`, `inner`, `ndarray.fill`, `imag`, `prod`, `put`, `putmask`, `real`, `sum`

Basic Statistics

`cov`, `mean`, `std`, `var`