Lecture Notes 2: Numpy, Timing, Plotting

Numpy Basics

```
In [1]: import numpy
```

Numpy arrays

```
In [2]: X = numpy.array([1,2,3,4])
        Y = numpy.array([5,6,7,8])
```

Operations between arrays

```
In [3]: A = X+Y            # element-wise addition
        M = X*Y            # element-wise multiplication
        D = numpy.dot(X,Y) # dot product

        A,M,D

Out[3]: (array([ 6,  8, 10, 12]), array([ 5, 12, 21, 32]), 70)
```

Equivalent operations with lists

```
In [4]: X = [1,2,3,4]
        Y = [5,6,7,8]

In [5]: A = [x+y for x,y in zip(X,Y)]      # element-wise addition
        M = [x*y for x,y in zip(X,Y)]      # element-wise multiplication
        D = sum([x*y for x,y in zip(X,Y)]) # dot product

        A,M,D

Out[5]: ([6, 8, 10, 12], [5, 12, 21, 32], 70)
```

Observation: Results are the same, but the Numpy syntax is terser (i.e. more compact) than the Python syntax for the same vector operations.

Matrix Multiplication

```
In [6]: A = numpy.array([[1,2],[3,4]])

In [7]: A*A

Out[7]: array([[ 1,  4],
               [ 9, 16]])

In [8]: numpy.dot(A,A)

Out[8]: array([[ 7, 10],
               [15, 22]])
```

Observation: Unlike Matlab, “*” denotes an element-wise multiplication. Matrix multiplication is instead implemented by the function “dot”.

Performance evaluation

To verify that in addition to the terser syntax, Numpy also provides a computational benefit over standard Python, we compare the running time of a similar computation performed in Python and in Numpy. The module "time" provides a function "clock" to measure the current time.

```
In [9]: import time
        time.clock()
```

```
Out[9]: 0.275665
```

we now wait a little bit...

```
In [10]: time.clock()
```

```
Out[10]: 0.279728
```

and can observed that the value is higher than before (time has passed). We now define two functions to test the speed of matrix multiplication for two $n \times n$ matrices.

```
In [11]: # pure Python implementation

         def benchmark_python(n):

             # initialization
             X = numpy.ones([n,n])
             Y = numpy.ones([n,n])
             Z = numpy.zeros([n,n])

             # actual matrix multiplication
             start = time.clock()
             for i in range(n):
                 for j in range(n):
                     for k in range(n):
                         Z[i,j] += X[i,k]*Y[k,j]
             end = time.clock()

             return end-start
```

```
In [12]: # Numpy implementation

         def benchmark_numpy(n):

             # initialization
             X = numpy.ones([n,n])
             Y = numpy.ones([n,n])
             Z = numpy.zeros([n,n])

             # actual matrix multiplication
             start = time.clock()
             Z = numpy.dot(X,Y)
             end = time.clock()

             return end-start
```

Trying this function for $n = 100$, we can observe that numpy is much faster than pure Python.

```
In [13]: benchmark_python(100),benchmark_numpy(100)
```

```
Out[13]: (0.570958, 0.0005460000000000464)
```

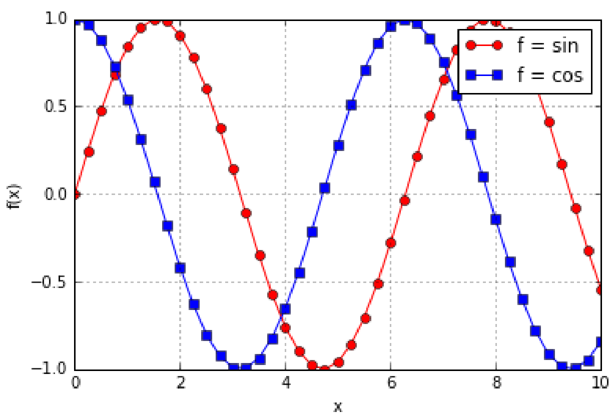
Plotting

In machine learning, it is often necessary to visualize the data, or to plot properties of algorithms such as their accuracy or their speed. For this, we can make use of the matplotlib library, which we load with the following sequence of commands.

In [14]:
```
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
```

Creating a Basic Plot

In [15]:
```
x = numpy.arange(0,10.001,0.25)
y = numpy.sin(x)
z = numpy.cos(x)

plt.plot(x,y,'o-',color='red',label='f = sin')
plt.plot(x,z,'s-',color='blue',label='f = cos')

plt.legend()

plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid(True)
```



Plotting a performance curve for matrix multiplication

We run the computation with different parameters (e.g. size of input arrays)

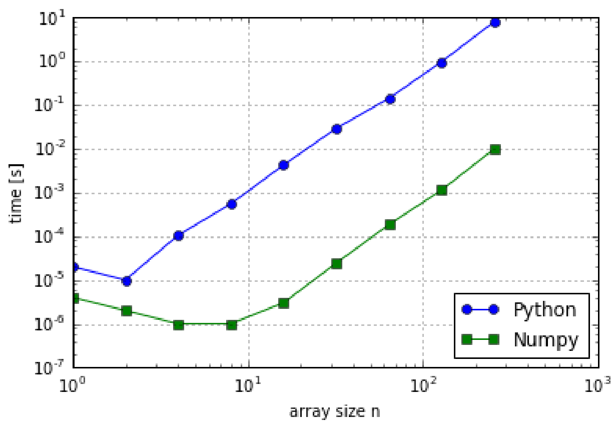
In [16]:
```
N = [1,2,4,8,16,32,64,128,256]

py_t = [benchmark_python(n) for n in N]
np_t = [benchmark_numpy(n) for n in N]
```

Then, we render the plot

In [18]:
```
plt.plot(N,py_t,'o-',label='Python')
plt.plot(N,np_t,'s-',label='Numpy')
plt.grid(True)
plt.xscale('log')
plt.yscale('log')
```

```
        plt.xlabel('array size n')
        plt.ylabel('time [s]')
        plt.legend(loc = 'lower right')
```

Out[18]: <matplotlib.legend.Legend at 0x7f05b7546950>



Advanced Numpy

Special Array Initializations

Numpy arrays can be initialized to specific values (numpy.zeros, numpy.ones, ...). Special numpy arrays (e.g. diagonal, identity) can be created easily.

```
In [19]: A = numpy.zeros([3,3])          # array of size 2x2 filled with zeros
         B = numpy.ones([3,3])           # same, but filled with ones
         C = numpy.diag([1.0,2.0,3.0])   # diagonal matrix
         D = numpy.eye(3)                # identity matrix

         print(A)
         print(B)
         print(C)
         print(D)
```

```
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
[[ 1.  0.  0.]
 [ 0.  2.  0.]
 [ 0.  0.  3.]]
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

Array type

```
In [23]: A = numpy.ones([2,2])
         type(A),A.shape,A.size,A.ndim,A.dtype
```

```
Out[23]: (numpy.ndarray, (2, 2), 4, 2, dtype('float64'))
```

```
In [24]: A = numpy.ones([3,3,3])
         type(A),A.shape,A.size,A.ndim,A.dtype
```

```
Out[24]: (numpy.ndarray, (3, 3, 3), 27, 3, dtype('float64'))
```

Casting

An array can be explicitly forced to have elements of a certain type (e.g. half-precision). When applying an operator to two arrays of different types, the returned array retains the type of the highest-precision input array (here, float64).

```
In [25]: E = A.astype('float32')
         A.dtype,E.dtype,(A+E).dtype
```

```
Out[25]: (dtype('float64'), dtype('float32'), dtype('float64'))
```

Reshaping and transposing

```
In [26]: A = numpy.array([[1,2,3],[4,5,6]])

         print(A)
         print(A.reshape([3,2]))
         print(A.T)
```

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

Broadcasting

See also https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html

```
In [27]: numpy.ones([3,2])+1
```

```
Out[27]: array([[ 2.,  2.],
                [ 2.,  2.],
                [ 2.,  2.]])
```

```
In [28]: numpy.ones([3,2])+numpy.ones([3,2])
```

```
Out[28]: array([[ 2.,  2.],
                [ 2.,  2.],
                [ 2.,  2.]])
```

```
In [29]: numpy.ones([3,1])+numpy.ones([1,2])
```

```
Out[29]: array([[ 2.,  2.],
                [ 2.,  2.],
                [ 2.,  2.]])
```

```
In [30]: numpy.ones([3,1])+numpy.ones([2])
```

```
Out[30]: array([[ 2.,  2.],
                [ 2.,  2.],
                [ 2.,  2.]])
```

Indexing
See also https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html

```
In [31]: A = numpy.arange(30).reshape(6,5)
         print(A)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]
 [25 26 27 28 29]]
```

Select rows/columns

```
In [32]: print(A[3,:])
         print(A[:,3])
```

```
[15 16 17 18 19]
[ 3  8 13 18 23 28]
```

Select window

```
In [33]: print(A[1:5,1:4])
```

```
[[ 6  7  8]
 [11 12 13]
 [16 17 18]
 [21 22 23]]
```

Select even rows and odd columns

```
In [34]: print(A[::2,1::2])
```

```
[[ 1  3]
 [11 13]
 [21 23]]
```

Select last two columns

```
In [35]: print(A[:,-2:])
```

```
[[ 3  4]
 [ 8  9]
 [13 14]
 [18 19]
 [23 24]
 [28 29]]
```

Select column 1 and 4

```
In [36]: print(A[:,[1,4]])
```

```
[[ 1  4]
 [ 6  9]
 [11 14]
 [16 19]
 [21 24]
 [26 29]]
```

Analyzing a Dataset

Let's load the Boston dataset (506 examples composed of 13 features each).

```
In [38]: # extract two interesting features of the data
         from sklearn.datasets import load_boston
         boston = load_boston()

         X = boston['data']
         F = boston['feature_names']

         print(F)
```

```
['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO'
 'B' 'LSTAT']
```

Reduce-type operations

```
In [39]: print(X.mean())          # Global dataset mean feature value
         print(X[:,0].mean())     # Mean of first feature (CRIM)
         print(X.mean(axis=0))    # Mean of all features
         print(X.std(axis=0))     # Standard deviation of all features
```

```
70.0724468258
3.59376071146
[  3.59376071e+00   1.13636364e+01   1.11367787e+01   6.91699605e-02
   5.54695059e-01   6.28463439e+00   6.85749012e+01   3.79504269e+00
   9.54940711e+00   4.08237154e+02   1.84555336e+01   3.56674032e+02
   1.26530632e+01]
[  8.58828355e+00   2.32993957e+01   6.85357058e+00   2.53742935e-01
   1.15763115e-01   7.01922514e-01   2.81210326e+01   2.10362836e+00
   8.69865112e+00   1.68370495e+02   2.16280519e+00   9.12046075e+01
   7.13400164e+00]
```

```
In [40]: # Show the feature name along with the mean and standard deviation
         zip(F,X.mean(axis=0),X.std(axis=0))
```

```
Out[40]: [('CRIM', 3.5937607114624508, 8.5882835476535533),
          ('ZN', 11.363636363636363, 23.299395694766027),
          ('INDUS', 11.136778656126504, 6.8535705833908729),
          ('CHAS', 0.069169960474308304, 0.25374293496034855),
          ('NOX', 0.55469505928853724, 0.11576311540656153),
          ('RM', 6.2846343873517867, 0.7019225143345692),
          ('AGE', 68.574901185770784, 28.121032570236885),
          ('DIS', 3.795042687747034, 2.1036283563444589),
          ('RAD', 9.5494071146245059, 8.6986511177906447),
          ('TAX', 408.23715415019763, 168.37049503938141),
          ('PTRATIO', 18.455533596837967, 2.1628051914821418),
          ('B', 356.67403162055257, 91.204607452172723),
          ('LSTAT', 12.653063241106723, 7.1340016366504848)]
```

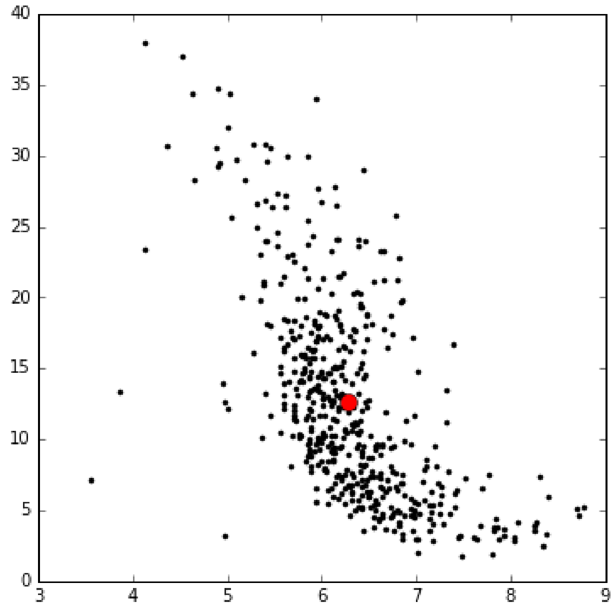
Retain two interesting features (5 and 12)

```
In [41]: X = X[:,[5,12]]
```

Scatter-plot the first two dimensions

In [42]:
```
plt.figure(figsize=(6,6))
plt.plot(X[:,0],X[:,1],'o',color='black',ms=3)
plt.plot(X[:,0].mean(),X[:,1].mean(),'o',color='red',ms=10)
```

Out[42]: [<matplotlib.lines.Line2D at 0x7f05afd75950>]



Normalize the data

In [43]:
```
X = X - X.mean(axis=0) # substract mean
X = X / X.std(axis=0)  # rescale features so that they have standard deviation 1
```

Computing a distance matrix

In [44]:
```
import scipy
import scipy.spatial

D = scipy.spatial.distance.cdist(X,X)
```

alternative way of computing a distance matrix:

In [45]:
```
Dalt = ((X**2).sum(axis=1).reshape([1,len(X)]) \
      + (X**2).sum(axis=1).reshape([len(X),1]) \
      - 2*numpy.dot(X,X.T))**.5

((Dalt-D)**2).mean()
```

Out[45]: 1.301244933328759e-31

Highlighting nearby data points

In [46]:
```
plt.figure(figsize=(6,6))

ind = numpy.where(D < 0.2)
```

```
plt.plot(X[:,0],X[:,1],'o',color='black',ms=3)

for i1,i2 in zip(*ind):
    plt.plot([X[i1,0],X[i2,0]],[X[i1,1],X[i2,1]],color='red',alpha=0.25)
```