Lecture Notes 2: Timing, Numpy, Plotting

Limitation of pure Python:

- Python is a simple and compact scripting language, but relatively slow

Machine learning problems require specific data structures:

- Vector spaces
- Data matrices
- Linear projections
- Distance matrices

Numpy provides fast methods to manipulate such data structures:

- **In surface:** intuitive user interface for manipulating arrays
- **Under the hood:** optimized code based on high performance libraries (BLAS, Lapack, etc)

Performance evaluation

To be convinced that Numpy provides a computational benefit over standard Python, we should be able to compare the running time of a similar computation performed in Python and in Numpy.

```
In [1]: import time
```

```
In [2]: ## Adding two vectors in python
        a = [i for i in range(1000000)]
        b = [1 for i in range(1000000)]
        c = [0 for i in range(1000000)] # output vector (initialized to zero)

        # Start the computation
        start = time.clock()
        for i in range(1000000):
            c[i] = a[i] + b[i]
        end = time.clock()

        print('%.3f seconds'%(end-start))
```

```
0.311 seconds
```

```
In [3]: ## Adding two vectors in numpy
        import numpy
        a = numpy.arange(1000000)
        b = numpy.ones(1000000)
        c = numpy.zeros(1000000)

        # Start the computation
        start = time.clock()
        numpy.add(a,b,out=c)
        end = time.clock()

        print('%.3f seconds'%(end-start))
```

```
0.006 seconds
```

The TimeIt "Magic Command"

```
In [4]: A = [i for i in range(100000)]
        B = [1 for i in range(100000)]

        %timeit [a+b for a,b in zip(A,B)]

        a = numpy.arange(100000)
        b = numpy.ones(100000)

        %timeit a+b
```

```
10 loops, best of 3: 23.4 ms per loop
1000 loops, best of 3: 239 µs per loop
```

Exercise 1: Plotting Performance

First, we load some modules for plotting in IPython

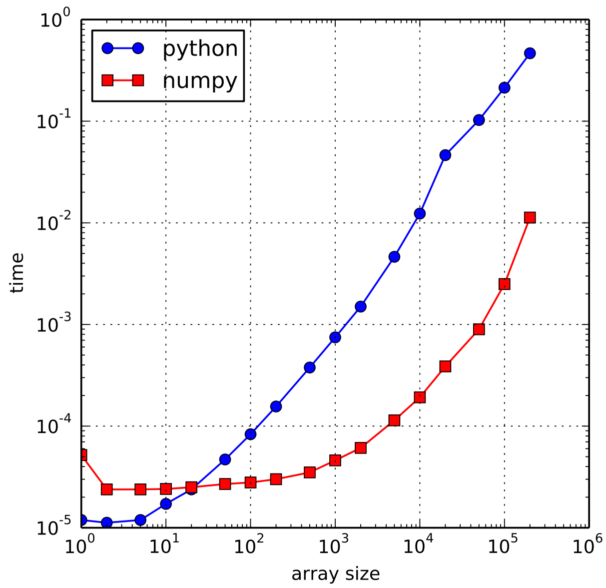
```
In [5]: import matplotlib
        from matplotlib import pyplot as plt

        # Allows to render plots directly within the notebook
        %matplotlib inline

        # Makes plots look nicer in the final PDF
        from IPython.display import set_matplotlib_formats
        set_matplotlib_formats('pdf', 'png')
        plt.rcParams['savefig.dpi'] = 90
```

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

```
In [6]: import timeit,numpy

        sizes = [1,2,5,10,20,50,100,200,500,1000,2000,5000,10000,20000,50000,100000,200000]

        pyexpr = lambda s: '[a+b for a,b in zip(range(%d),range(%d))]'%(s,s)
        npexpr = lambda s: 'numpy.arange(%d)+numpy.arange(%d)'%(s,s)

        pytime = [timeit.timeit(pyexpr(s),number=5) for s in sizes]
        nptime = [timeit.timeit(npexpr(s),setup='import numpy;',number=5) for s in sizes]
```

Then, we render the plot

```
In [7]: plt.figure(figsize=(5,5))
        plt.plot(sizes,pytime,'o-',color='blue',label='python')
        plt.plot(sizes,nptime,'s-',color='red',label='numpy')
        plt.xlabel('array size'); plt.ylabel('time')
        plt.xscale('log'); plt.yscale('log')
        plt.legend(loc='upper left'); plt.grid(True)
```



Numpy basics

Numpy arrays can be directly initialized by the function `numpy.array`

```
In [8]: numpy.array([[1.0,2.0],[3.0,4.0]])

Out[8]: array([[ 1.,  2.],
               [ 3.,  4.]])
```

Numpy arrays can be initialized to specific values (`numpy.zeros`, `numpy.ones`, ...)

```
In [9]: numpy.ones([2,2])

Out[9]: array([[ 1.,  1.],
               [ 1.,  1.]])
```

Special numpy arrays (e.g. identity) can be created easily

```
In [10]: numpy.identity(2)

Out[10]: array([[ 1.,  0.],
                [ 0.,  1.]])

In [11]: numpy.diag([1.0,2.0,3.0])

Out[11]: array([[ 1.,  0.,  0.],
                [ 0.,  2.,  0.],
                [ 0.,  0.,  3.]])
```

Numpy arrays can be initialized randomly following a probability distribution

```
In [12]: numpy.random.uniform(0,1,[3,3])
```

```
Out[12]: array([[ 0.36957268,  0.24852523,  0.84041215],
                [ 0.34741939,  0.98912633,  0.89519385],
                [ 0.94520512,  0.76769106,  0.09574112]])
```

```
In [13]: numpy.random.exponential(1,[3,3])
```

```
Out[13]: array([[ 0.22617325,  1.08486358,  0.32758166],
                [ 1.06398485,  1.23876862,  1.1873128 ],
                [ 0.38338351,  2.00111191,  1.03153185]])
```

Multidimensional arrays can be created

```
In [14]: numpy.ones([2,2,2,2])
```

```
Out[14]: array([[[[ 1.,  1.],
                  [ 1.,  1.]],

                 [[ 1.,  1.],
                  [ 1.,  1.]]],


                [[[ 1.,  1.],
                  [ 1.,  1.]],

                 [[ 1.,  1.],
                  [ 1.,  1.]]]])
```

The properties of an array

```
In [15]: a = numpy.ones([2,2])
         print type(a), a.shape, a.size, a.ndim, a.dtype
```

```
<type 'numpy.ndarray'> (2, 2) 4 2 float64
```

```
In [16]: a = numpy.ones([3,3,3],dtype='float32')
         print type(a), a.shape, a.size, a.ndim, a.dtype
```

```
<type 'numpy.ndarray'> (3, 3, 3) 27 3 float32
```

Casting

Explicit Casting

```
In [17]: a = numpy.ones([2,2])
         print a
         b = a.astype('int16')
         print b
```

```
[[ 1.  1.]
 [ 1.  1.]]
[[1 1]
 [1 1]]
```

Automatic Casting

```
In [18]: a = numpy.array([[1.0,2.0],[3.0,4.0]],dtype='float64')
         b = numpy.array([[2.0,3.0],[4.0,5.0]],dtype='float32')
         a*b
```

```
Out[18]: array([[  2.,   6.],
                [ 12.,  20.]])
```

```
In [19]: a.dtype,b.dtype
```

```
Out[19]: (dtype('float64'), dtype('float32'))
```

```
In [20]: (a*b).dtype
```

```
Out[20]: dtype('float64')
```

- Output array is assigned precision as high as the most precise input array (here, float64).
- **Warning for Matlab users:** Multiplication and division operators apply element-wise.

Reshaping

Explicit Reshaping

```
In [21]: a = numpy.array([[1.0,2.0],[3.0,4.0],[5.0,6.0]])
         a
```

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

```
In [22]: a.flatten()
```

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

```
In [23]: a.reshape([2,3])
```

```
Out[23]: array([[ 1.,  2.,  3.],
                [ 4.,  5.,  6.]])
```

Automatic Reshaping

```
In [24]: a = numpy.array([[1.0,2.0],[3.0,4.0],[5.0,6.0]])
         a
```

```
Out[24]: array([[ 1.,  2.],
                [ 3.,  4.],
                [ 5.,  6.]])
```

```
In [25]: a+3
```

```
Out[25]: array([[ 4.,  5.],
                [ 6.,  7.],
                [ 8.,  9.]])
```

```
In [26]: a*numpy.array([1.0,0.0])
```

```
Out[26]: array([[ 1.,  0.],
                [ 3.,  0.],
                [ 5.,  0.]])
```

```
In [27]: shape1 = (3,1,2)
         shape2 = (1,4,1)
         (numpy.zeros(shape1)+numpy.zeros(shape2)).shape
```

```
Out[27]: (3, 4, 2)
```

Numpy Reduce-type Functions

```
In [28]: a = numpy.array([[1.0,2.0],[3.0,4.0],[5.0,6.0]])
         a

Out[28]: array([[ 1.,  2.],
                [ 3.,  4.],
                [ 5.,  6.]])

In [29]: a.sum(axis=1)

Out[29]: array([  3.,   7.,  11.])

In [30]: a.sum(axis=0)

Out[30]: array([  9.,  12.])

In [31]: a.sum()

Out[31]: 21.0
```

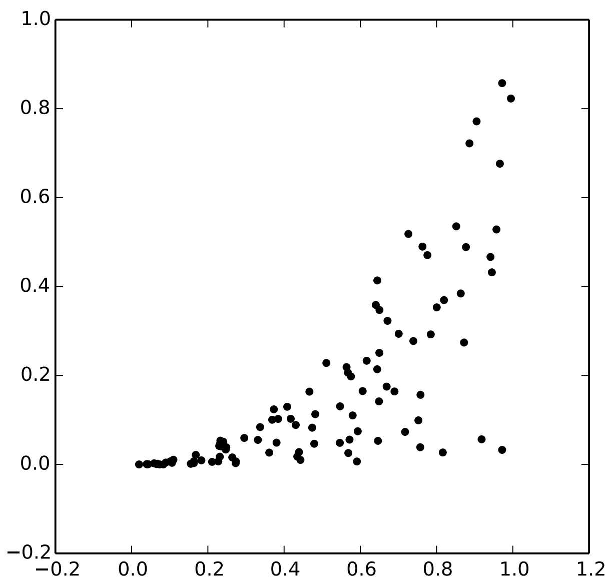
Datasets and scatter plots

Create a dataset of 100 randomly sampled data points

```
In [33]: import numpy.random
         x1 = numpy.random.uniform(0,1,[100,1])
         x2 = numpy.random.uniform(0,1,[100,1]) * x1**2
         X = numpy.concatenate([x1,x2],axis=1)
```

Plot the dataset

```
In [34]: plt.figure(figsize=(5,5))
         plt.scatter(X[:,0],X[:,1],color='black',s=10)

Out[34]: <matplotlib.collections.PathCollection at 0x7f1182341690>
```



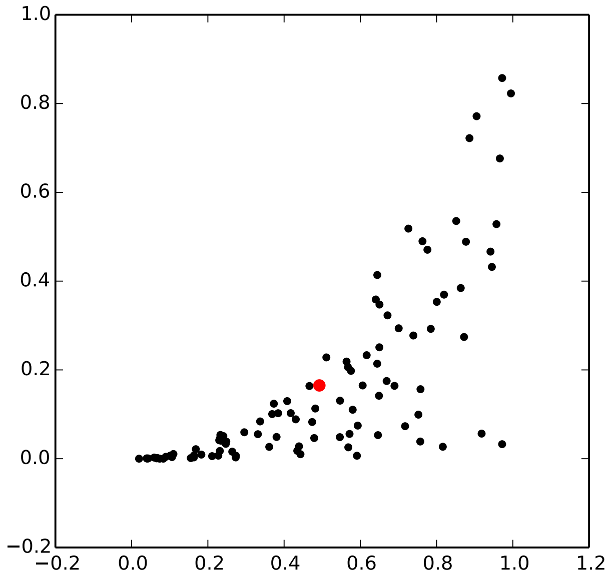
Exercise 2: Computing and plotting the mean of a dataset

In [35]:
```
m = X.mean(axis=0)
m
```

Out[35]:
```
array([ 0.49252741,  0.16503012])
```

In [36]:
```
plt.figure(figsize=(5,5))
plt.scatter(*X.T,color='black',s=10)
plt.scatter([m[0]],[m[1]],color='red',s=30)
```

Out[36]:
```
<matplotlib.collections.PathCollection at 0x7f11827bedd0>
```



Selection on Arrays

In [37]:
```
a = numpy.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]],dtype='float')
a
```

Out[37]:
```
array([[  1.,   2.,   3.],
       [  4.,   5.,   6.],
       [  7.,   8.,   9.],
       [ 10.,  11.,  12.]])
```

In [38]:
```
a[0]
```

Out[38]:
```
array([ 1.,  2.,  3.])
```

In [39]:
```
a[1,0]
```

Out[39]:
```
4.0
```

In [40]:
```
a[:2]
```

Out[40]:
```
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

```
In [41]: a[:,:2]
```

```
Out[41]: array([[  1.,   2.],
                [  4.,   5.],
                [  7.,   8.],
                [ 10.,  11.]])
```

```
In [42]: a[1:3,1:2]
```

```
Out[42]: array([[ 5.],
                [ 8.]])
```

```
In [43]: a[::2]
```

```
Out[43]: array([[ 1.,  2.,  3.],
                [ 7.,  8.,  9.]])
```

```
In [44]: a[:,::-1]
```

```
Out[44]: array([[  3.,   2.,   1.],
                [  6.,   5.,   4.],
                [  9.,   8.,   7.],
                [ 12.,  11.,  10.]])
```

```
In [45]: a[[0,3],:]
```

```
Out[45]: array([[  1.,   2.,   3.],
                [ 10.,  11.,  12.]])
```

```
In [46]: a[[0,3]][:,[0,2]]
```

```
Out[46]: array([[  1.,   3.],
                [ 10.,  12.]])
```

Matrix multiplication

```
In [47]: a = numpy.array([[1.0,2.0],[3.0,4.0],[5.0,6.0]])
         b = numpy.array([[1.0,2.0,1.0,2.0],[3.0,4.0,2.0,1.0]])

         a.shape,b.shape
```

```
Out[47]: ((3, 2), (2, 4))
```

```
In [48]: numpy.dot(a,b).shape
```

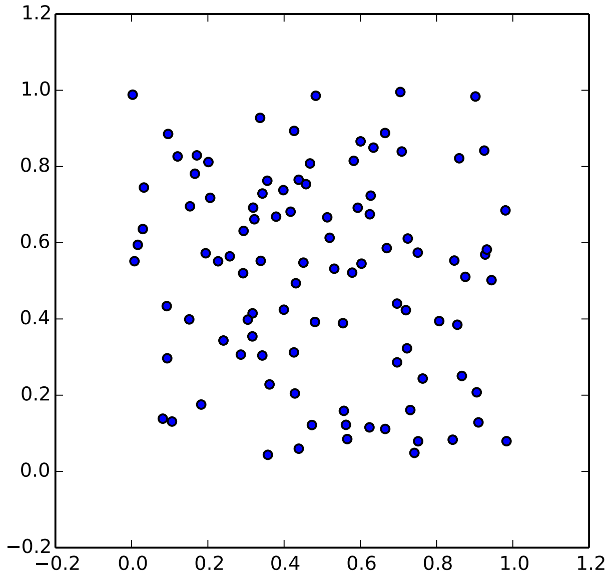
```
Out[48]: (3, 4)
```

Datasets and distance matrices

```
In [49]: X = numpy.random.mtrand.RandomState(123).uniform(0,1,[100,2])
```

```
In [50]: plt.figure(figsize=(5,5))
         plt.scatter(*X.T)
```

```
Out[50]: <matplotlib.collections.PathCollection at 0x7f11827b06d0>
```



Compute distance matrix (square Euclidean)

```
In [51]: import scipy,scipy.spatial
         D = scipy.spatial.distance.cdist(X,X,'sqeuclidean')
```

Distance matrices can also be written with numpy.dot

```
In [52]: Dalt = (X**2).sum(axis=1).reshape([1,100]) \
              + (X**2).sum(axis=1).reshape([100,1]) \
              - 2*numpy.dot(X,X.T)
```

We can verify that both computations are equivalent

```
In [53]: ((Dalt-D)**2).mean()

Out[53]: 1.6221555628483391e-32
```

Exercise 3: Finding Points that are Furthest Apart

Max distance between data points

```
In [54]: D.max()

Out[54]: 1.7882784080338652
```

Which pair of points has max distance

```
In [55]: numpy.argmax(D)

Out[55]: 5967
```

We need to convert the index of the flattened array to the index of the original array

```
In [56]: a,b = numpy.unravel_index(numpy.argmax(D),D.shape)
         print a,X[a]
         print b,X[b]
```

```
59 [ 0.00268806  0.98834542]
67 [ 0.98352161  0.07936579]
```

Verify that these two points are indeed the ones with maximum distance

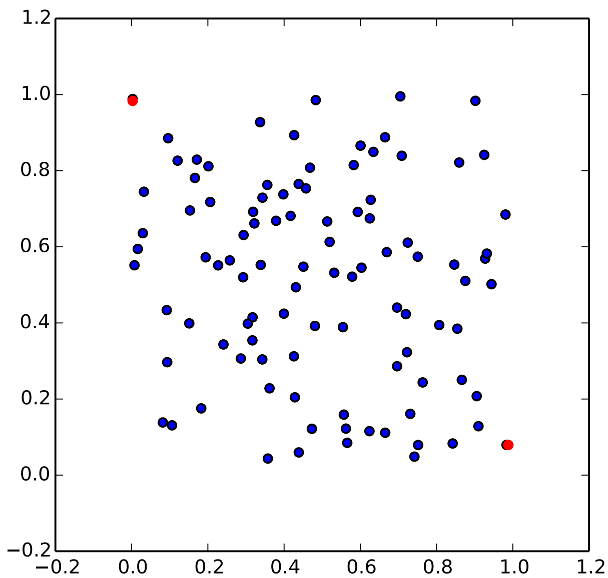
```
In [57]: ((X[a]-X[b])**2).sum() - D.max()
```

```
Out[57]: -2.2204460492503131e-16
```

Plotting these two distant points

```
In [58]: plt.figure(figsize=(5,5))
         plt.scatter(*X.T)
         plt.scatter(*X[[a,b]],color='red')
```

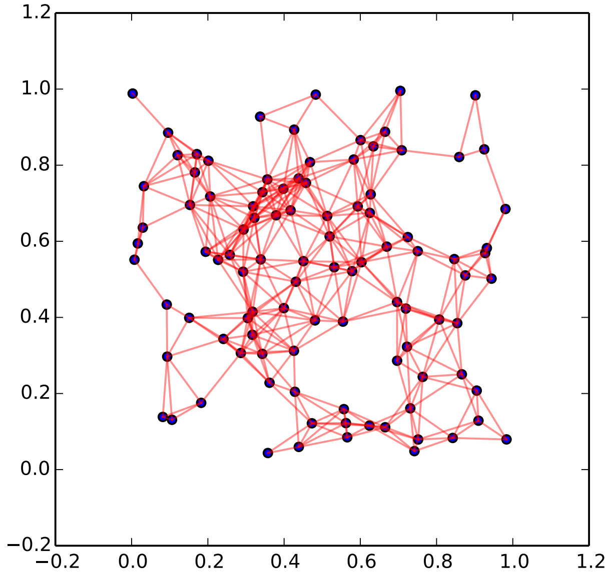
```
Out[58]: <matplotlib.collections.PathCollection at 0x7f1182895fd0>
```



Exercise 4: Building a Nearest Neighbor Graph

Plotting pairs of points that are at a smaller distance than a tenth of the average

```
In [59]: m = D.sum() / (len(D) * (len(D)-1))
         ind = numpy.where(D < 0.1*m)
         plt.figure(figsize=(5,5))
         plt.scatter(*X.T)
         for i,j in zip(*ind): plt.plot(*X[[i,j]].T,color='red',alpha=0.25)
```



1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2