SoSe 2017: Python for Machine Learning : Assignment #1

Due on Tuesday, May 2, 2017, 10:00am

Hauke Lars Iben 337 398, Michael Eric Stenzel 386 258, Ştefan Cobeli 514 7789.

Contents

Problem 1

The code for the programming assignment:

```
#your name and matrikelnr
#337398
#Michael Stenze, 386258
#Stefan Cobeli, 514 7789
import pylab as pl
import scipy as sp
import time
import pdb



''' ---- Task 1 ---- '''

def task1():
    '''
    Task 1
    Generate, transform and plot gaussian data
    '''
    X = generate_data(100)
    X2 = scale_data(X)
    X3 = standardise_data(X)
    # Plot data
    # Your code here
    pl.hold(True)
    pl.xlabel('X axis')
    pl.ylabel('Y axis')
    pl.title('Basic point manipulation')
    pl.xlim(-6, 10)
    pl.ylim(-5, 5)
    original = pl.scatter(X[0],X[1],  marker='.', c='y', label='original')
    scaled = pl.scatter(X2[0],X2[1],  marker='.', c='r', label='scaled')
    standardised = pl.scatter(X3[0],X3[1], marker='x', c='b', label='standardised')
    pl.legend((original, scaled, standardised), ('original', 'scaled', 'standardised'), loc="upper r
    pl.savefig("simple_transformation_of_Gaussian_data.pdf", bbox_inches='tight')
    # Hint: Use the functions pl.scatter(x[0,:],[1,:],c='r'), pl.hold(True),
    # pl.legend, pl.title, pl.xlabel, pl.ylabel

def generate_data(N):
    '''
    Generate N data points form a 2D Gaussian Gaussian distribution
    with mean [1, 2]

    Usage:     x = generate_data(N)

    Returns:   x : a 2xN array

    Instructions: Use sp.random.multivariate_normal
    '''
    # Your code here
    mean = [1, 2]
```

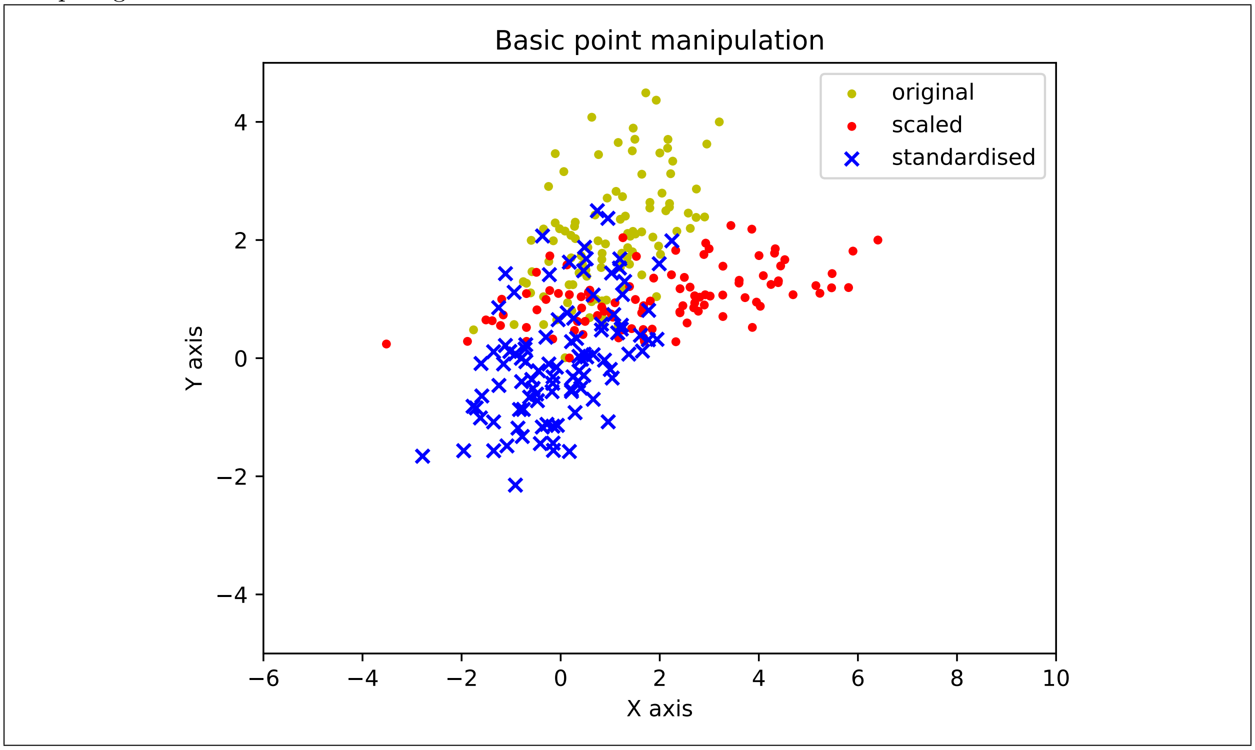
Problem 1 continued on next page. . .

```
    cov = [[1, 0.5], [0.5, 1]]
    return sp.random.multivariate_normal(mean, cov, N).T

def scale_data(X):
    '''
    Scales the data in X by 2 in x-direction and by 0.5 in y-direction

    Usage:     Y = scale_data(X)
    Input:     X : a 2xN array
    Returns:   Y : a 2xN array of scaled data

    '''
    # Your code here
    return [[2*i for i in X[0]],[.5*i for i in X[1]]]
def standardise_data(X):
    ''' Returns a centered, scaled version of X, the same size as X.

    Usage:      Y = standardise_data(X)
    Input:      X : a DxN array
    Returns:    Y : a DxN array of z-scores of X
                    Y[i][n] = (X[i][n] - mean(X[i][:]))/std(X[i][:])

    Instructions: Do not use for-loops. Use sp.mean and sp.std
    '''
    # Your code here
    means2 = sp.mean(X, axis = 1, keepdims=True)
    squares2 = sp.std(X, axis = 1, keepdims=True)
    return (X-means2)/squares2

''' ---- Task 2 ---- '''

def task2():
    '''
    Task 2
    Calculate time demand of different mean calculations
    (for-loop based implementation vs. scipy.mean)
    '''
    dims = [100, 1000, 10**4, 10**5, 10**6]
    for i, d in enumerate(dims):
        x = generate_data(d)
        r1 = timedcall(mean_for, x)
        r2 = timedcall(sp.mean, x,1)
        print 'For N = ' + str(d) + ' scipy.mean is ' + str(r1 / r2) \
            + 's faster than a for-loop implementation'


def mean_for(X):
    ''' Mean of array X along the rows

    Usage:      m = mean_for(X)
    Input:      X : a DxN array
    Returns:    m : a 1-dimensional array of length D, containing the means of each row

    Example: if   X =  [1 5      mean_for(X) = [3 4 5]
```

```
                        2 6
                        3 7]


    Instructions: Use for-loops to replicate sp.mean(X,1)
    Do not use sp.mean or sp.sum
    '''
    # Your code here
    return [1.*sum(x)/len(x) for x in X]

def timedcall(fn, *args):
    '''Call function with args; return the time in seconds and result.
        example:
        You want to time the function call "C = foo(A,B)".
        --> "T, C = timecall(foo, A, B)"
    '''
    t0 = time.clock()
    result = fn(*args)
    t1 = time.clock()
    return t1-t0

''' ---- Function for testing ---- '''

def test_prep():
    a = sp.array([[ 1.,  3.,  4.],[ 2.,  4.,  6.]])
    b = sp.array([[ 2.,  6.,  8.],[ 1.,  2.,  3.]])
    #test scale_data
    assert(sp.all(scale_data(a) == b))
    #test standardise_data
    assert(sp.all(standardise_data(a.T) == sp.array([[-1.,  1.],[-1.,  1.],[-1.,  1.]])))
    assert(sp.all(sp.mean(standardise_data(b),1).round() == sp.zeros((1,2))))
    c = sp.concatenate((a,b),axis=0)
    assert(sp.all(sp.mean(standardise_data(c),1).round() == sp.zeros((1,4))))
    #test mean_for
    assert(sp.all(mean_for(a) == sp.mean(a,1)))
    #test generate_data
    x = generate_data(200)
    assert(x.shape == (2, 200))
    print 'Tests passed'
task1()
task2()
```

Problem 2

The plot generated in Task 1.4:



Problem 3

The Answer to the Task 2.2 is: The function that is faster is **scipy.mean**. The output from the task2 function was:

```
For N = 100 scipy.mean is 1.14423076923s faster than a for-loop implementation
For N = 1000 scipy.mean is 5.00746268656s faster than a for-loop implementation
For N = 10000 scipy.mean is 10.3560732113s faster than a for-loop implementation
For N = 100000 scipy.mean is 12.7634191176s faster than a for-loop implementation
For N = 1000000 scipy.mean is 12.1978746794s faster than a for-loop implementation
```