week1_v2

December 3, 2019

1 Mean/Covariance of a data set and effect of a linear transformation

We are going to investigate how the mean and (co)variance of a dataset changes when we apply affine transformation to the dataset.

1.1 Learning objectives

1. Get Farmiliar with basic programming using Python and Numpy/Scipy.
2. Learn to appreciate implementing functions to compute statistics of dataset in vectorized way.
3. Understand the effects of affine transformations on a dataset.
4. Understand the importance of testing in programming for machine learning.

First, let's import the packages that we will use for the week

```
In [1]: # PACKAGE: DO NOT EDIT THIS CELL
        import numpy as np
        import matplotlib
        matplotlib.use('Agg')
        import matplotlib.pyplot as plt
        matplotlib.style.use('fivethirtyeight')
        from sklearn.datasets import fetch_lfw_people, fetch_olivetti_faces
        import time
        import timeit
```

```
In [2]: %matplotlib inline
        from ipywidgets import interact
```

Next, we are going to retrieve Olivetti faces dataset.

When working with some datasets, before digging into further analysis, it is almost always useful to do a few things to understand your dataset. First of all, answer the following set of questions:

1. What is the size of your dataset?
2. What is the dimensionality of your data?

The dataset we have are usually stored as 2D matrices, then it would be really important to know which dimension represents the dimension of the dataset, and which represents the data points in the dataset.

When you implement the functions for your assignment, make sure you read the docstring for what each dimension of your inputs represents the data points, and which represents the dimensions of the dataset!. For this assignment, our data is organized as **(D,N)**, where D is the dimensionality of the samples and N is the number of samples.

In [34]:
```
image_shape = (64, 64)
# Load faces data
dataset = fetch_olivetti_faces('./')
faces = dataset.data.T

print('Shape of the faces dataset: {}'.format(faces.shape))
print('{} data points'.format(faces.shape[1]))
abc = np.array([[5,7],[8,10]])
#print(abc)
sum = np.sum(abc,axis=1)
#print (sum)
#print(sum.shape)
sum = sum.reshape(2,1)
#print(sum)
#print(sum.shape)
abc = np.array([[5,7],[8,10]])
print(abc)
print(abc.shape)
mean = np.mean(abc,axis=1).reshape(2,1)
print(mean)
print (mean.shape)
```

```
Shape of the faces dataset: (4096, 400)
400 data points
[[ 5  7]
 [ 8 10]]
(2, 2)
[[ 6.]
 [ 9.]]
(2, 1)
```

When your dataset are images, it's a really good idea to see what they look like.

One very convenient tool in Jupyter is the `interact` widget, which we use to visualize the images (faces). For more information on how to use interact, have a look at the documentation here.

We have created two function which help you visuzlie the faces dataset. You do not need to modify them.

In [9]:
```
def show_face(face):
    plt.figure()
    plt.imshow(face.reshape((64, 64)), cmap='gray')
    plt.show()
```

```
In [12]: @interact(n=(0, faces.shape[1]-1))
         def display_faces(n=0):
             plt.figure()
             plt.imshow(faces[:,n].reshape((64, 64)), cmap='gray')
             plt.show()
```

```
interactive(children=(IntSlider(value=0, description='n', max=399), Output()), _dom_classes=('
```

1.2 1. Mean and Covariance of a Dataset

In this week, you will need to implement functions in the cell below which compute the mean and covariance of a dataset.

You will implement both mean and covariance in two different ways. First, we will implement them using Python's for loops to iterate over the entire dataset. Later, you will learn to take advantage of Numpy and use its library routines. In the end, we will compare the speed differences between the different approaches.

```
In [51]: # GRADED FUNCTION: DO NOT EDIT THIS LINE
         def mean_naive(X):
             "Compute the mean for a dataset X nby iterating over the data points"
             # X is of size (D,N) where D is the dimensionality and N the number of data point
             D, N = X.shape
             mean = np.zeros((D,1))
             ### Edit the code; iterate over the dataset and compute the mean vector.
             ###Nothing to change.. It is already doing the sum


             mean = (np.sum(X,axis=1)/N).reshape(D,1)
             return mean

         def cov_naive(X):
             """Compute the covariance for a dataset of size (D,N)
             where D is the dimension and N is the number of data points"""
             D, N = X.shape
             ### Edit the code below to compute the covariance matrix by iterating over the da
             covariance = np.zeros((D, D))
             ### Update covariance
             diff = X - mean_naive(X)
             covariance = (diff @ diff.T)/N

             ###
             return covariance


         def mean(X):
             "Compute the mean for a dataset of size (D,N) where D is the dimension and N is t
             # given a dataset of size (D, N), the mean should be an array of size (D,1)
```

```
    # you can use np.mean, but pay close attention to the shape of the mean vector yo
    D, N = X.shape
    ### Edit the code to compute a (D,1) array `mean` for the mean of dataset.
    mean = np.zeros((D,1))
    ### Update mean here
    mean = np.mean(X,axis=1).reshape(D,1)
    ###
    return mean

def cov(X):
    "Compute the covariance for a dataset"
    # X is of size (D,N)
    # It is possible to vectorize our code for computing the covariance with matrix m
    # i.e., we do not need to explicitly
    # iterate over the entire dataset as looping in Python tends to be slow
    # We challenge you to give a vectorized implementation without using np.cov, but
    # be sure to pass in bias=True.
    D, N = X.shape
    ### Edit the code to compute the covariance matrix
    covariance_matrix = np.zeros((D, D))
    ### Update covariance_matrix here
    diff = X - mean(X)
    covariance_matrix = (diff @ diff.T)/N
    #covariance_matrix = cov_naive(X)
    ###
    return covariance_matrix
```

Now, let's see whether our implementations are consistent

In [52]:
```
# Let's first test the functions on some hand-crafted dataset.

X_test = np.arange(6).reshape(2,3)
expected_test_mean = np.array([1., 4.]).reshape(-1, 1)
expected_test_cov = np.array([[2/3., 2/3.], [2/3.,2/3.]])
print('X:\n', X_test)
print('Expected mean:\n', expected_test_mean)
print('Expected covariance:\n', expected_test_cov)

np.testing.assert_almost_equal(mean(X_test), expected_test_mean)
np.testing.assert_almost_equal(mean_naive(X_test), expected_test_mean)

np.testing.assert_almost_equal(cov(X_test), expected_test_cov)
np.testing.assert_almost_equal(cov_naive(X_test), expected_test_cov)
```

```
X:
 [[0 1 2]
 [3 4 5]]
Expected mean:
```

```
 [[ 1.]
 [ 4.]]
Expected covariance:
 [[ 0.66666667  0.66666667]
 [ 0.66666667  0.66666667]]
```

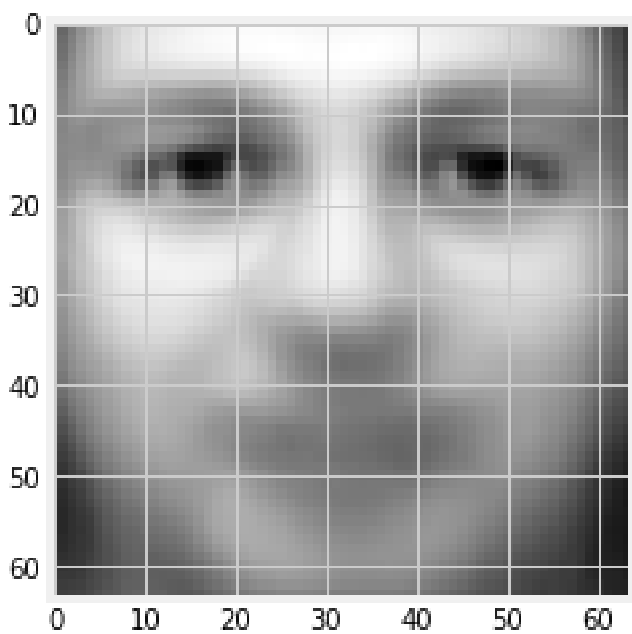
We now test that both implementation should give identical results running on the faces dataset.

```
In [53]: np.testing.assert_almost_equal(mean(faces), mean_naive(faces), decimal=6)
         np.testing.assert_almost_equal(cov(faces), cov_naive(faces))
```

With the `mean` function implemented, let's take a look at the *mean* face of our dataset!

```
In [54]: def mean_face(faces):
             return faces.mean(axis=1).reshape((64, 64))

         plt.imshow(mean_face(faces), cmap='gray');
```



Loops in Python are slow, and most of the time you want to utilise the fast native code provided by Numpy without explicitly using for loops. To put things into perspective, we can benchmark the two different implementation with the `%time` function in the following way:

```
In [55]: # We have some HUUUGE data matrix which we want to compute its mean
         X = np.random.randn(20, 1000)
         # Benchmarking time for computing mean
```

```
%time mean_naive(X)
%time mean(X)
pass
```

```
CPU times: user 70 ţs, sys: 22 ţs, total: 92 ţs
Wall time: 96.8 ţs
CPU times: user 447 ţs, sys: 0 ns, total: 447 ţs
Wall time: 403 ţs
```

```
In [56]: # Benchmarking time for computing covariance
         %time cov_naive(X)
         %time cov(X)
         pass
```

```
CPU times: user 1.01 ms, sys: 316 ţs, total: 1.33 ms
Wall time: 58.6 ms
CPU times: user 182 ţs, sys: 57 ţs, total: 239 ţs
Wall time: 245 ţs
```

As you can see, using Numpy's functions makes the code much faster! Therefore, whenever you can use something that's implemented in Numpy, be sure that you take advantage of that.

1.3 2. Affine Transformation of Datasets

In this week we are also going to verify a few properties about the mean and covariance of affine transformation of random variables.

Consider a data matrix $\boldsymbol{X}$ of size (D, N). We would like to know what is the covariance when we apply affine transformation $\boldsymbol{A}\boldsymbol{x}_i + \boldsymbol{b}$ for each datapoint $\boldsymbol{x}_i$ in $\boldsymbol{X}$, i.e., we would like to know what happens to the mean and covariance for the new dataset if we apply affine transformation.

For this assignment, you will need to implement the `affine_mean` and `affine_covariance` in the cell below.

```
In [57]: # GRADED FUNCTION: DO NOT EDIT THIS LINE
         def affine_mean(mean, A, b):
             """Compute the mean after affine transformation
             Args:
                 mean: ndarray, the mean vector
                 A, b: affine transformation applied to x
             Returns:
                 mean vector after affine transformation
             """
             ### Edit the code below to compute the mean vector after affine transformation
             affine_m = np.zeros(mean.shape) # affine_m has shape (D, 1)
             ### Update affine_m
             affine_m = A@mean+b

             ###
```