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

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

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.

Here are a few links for your reference. You may want to refer back to them throughout the whole course.

- If you are less comfortable with programming in Python, have a look at this Coursera course https://www.coursera.org/learn/python (https://www.coursera.org/learn/python).
- To learn more about using Scipy/Numpy, have a look at the Getting Started Guide (https://scipy.org/getting-started.html). You should also refer to the numpy documentation (https://docs.scipy.org/doc/) for references of available functions.
- If you want to learn more about creating plots in Python, checkout the tutorials found on matplotlib's website https://matplotlib.org/tutorials/index.html (https://matplotlib.org/tutorials/index.html). Once you are more familiar with plotting, check out this excellent blog post http://pbpython.com/effective-matplotlib.html (http://pbpython.com/effective-matplotlib.html).
- There are more advanced libraries for interactive data visualization. For example, bqplot (https://github.com/bloomberg/bqplot) or d3.js (https://d3js.org/). You may want to check out other libraries if you feel adventurous.
- Although we use Jupyter notebook for these exercises, you may also want to check out Jupyter Lab (https://github.com/jupyterlab/jupyterlab) when you want to work on your own projects.

First, let's import the packages that we will use for the week. Run the cell below to import the packages.

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

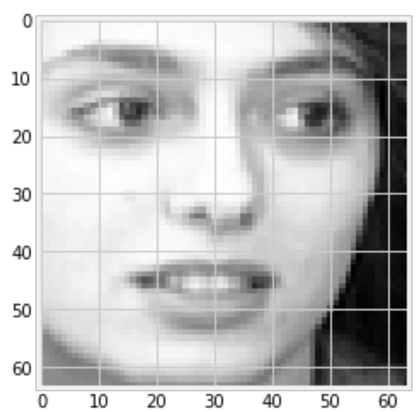
In [3]:
```
image_shape = (64, 64)
# Load faces data
dataset = fetch_olivetti_faces()
faces = dataset.data

print('Shape of the faces dataset: {}'.format(faces.shape))
print('{} data points'.format(faces.shape[0]))
```

```
downloading Olivetti faces from http://cs.nyu.edu/~roweis/data/olivettifaces.mat (
http://cs.nyu.edu/~roweis/data/olivettifaces.mat) to /home/jovyan/scikit_learn_dat
a
Shape of the faces dataset: (400, 4096)
400 data points
```

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 (http://ipywidgets.readthedocs.io/en/stable/examples/Using%20Interact.html).

In [4]:
```
@interact(n=(0, len(faces)-1))
def display_faces(n=0):
    plt.figure()
    plt.imshow(faces[n].reshape((64, 64)), cmap='gray')
    plt.show()
```



1. Mean and Covariance of a Dataset

You will now need to implement functions to which compute the mean and covariance of a dataset.

There are two ways to compute the mean and covariance. The naive way would be to iterate over the dataset to compute them. This would be implemented as a `for` loop in Python. However, computing them for large dataset would be slow. Alternatively, you can use the functions provided by numpy to compute them, these are much faster as numpy uses machine code to compute them. You will implment function which computes mean and covariane both in the naive way and in the fast way. Later we will compare the performance between these two approaches. If you need to find out which numpy routine to call, have a look at the documentation https://docs.scipy.org/doc/numpy/reference/ (https://docs.scipy.org/doc/numpy/reference/). It is a good exercise to refer to the official documentation whenever you are not sure about something.

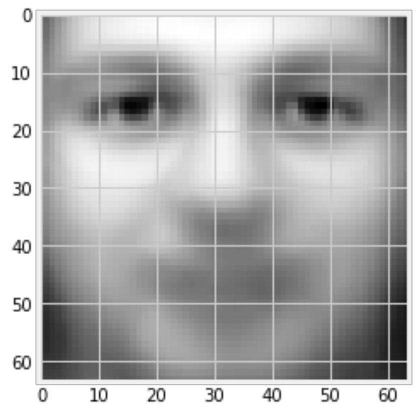
When you implement the functions for your assignment, make sure you read the docstring which dimension of your inputs corresponds to the number of data points and which corresponds to the dimension of the dataset.

In [29]:
```
# ===YOU SHOULD EDIT THIS FUNCTION===
import statistics as st

def mean_naive(X):
    """Compute the mean for a dataset by iterating over the dataset

    Arguments
    ---------
    X: (N, D) ndarray representing the dataset.

    Returns
    -------
    mean: (D, ) ndarray which is the mean of the dataset.
    """
    N, D = X.shape
    mean = np.zeros(D)
    for m in range(D):
        k=0
        for n in range(N):
            k+=X[n,m]
        smean=k/N
        mean[m]=smean
    return mean

# ===YOU SHOULD EDIT THIS FUNCTION===
def cov_naive(X):
    N, D = X.shape
    covariance = np.zeros((D, D))
    for i in range (D):
        eDi=sum(X[:,i])/N
        for j in range (D):
            eDj=sum(X[:,j])/N
            m=0
            for k in range(N):
                m+=(X[k,i]-eDi)*(X[k,j]-eDj)
            co=m/N
            covariance[i,j]=co
    return covariance
```

In [39]:
```
# GRADED FUNCTION: DO NOT EDIT THIS LINE

# ===YOU SHOULD EDIT THIS FUNCTION===
def mean(X):
    """Compute the mean for a dataset

    Arguments
    ---------
    X: (N, D) ndarray representing the dataset.

    Returns
    -------
    mean: (D, ) ndarray which is the mean of the dataset.
    """
    mean = np.zeros(X.shape[1]) # EDIT THIS
    mean = np.mean(X, axis=0)
    return mean

# ===YOU SHOULD EDIT THIS FUNCTION===
def cov(X):
    """Compute the covariance for a dataset
    Arguments
    ---------
    X: (N, D) ndarray representing the dataset.

    Returns
    -------
    covariance_matrix: (D, D) ndarray which is the covariance matrix of the dataset.

    """
    # It is possible to vectorize our code for computing the covariance, i.e. we do r
    # iterate over the entire dataset as looping in Python tends to be slow
    N, D = X.shape
    covariance_matrix = np.zeros((D, D)) # EDIT THIS
    Y = np.transpose(X)
    covariance_matrix = np.cov(Y)
    return covariance_matrix
```

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

In [17]:
```
def mean_face(faces):
    """Compute the mean of the `faces`

    Arguments
    ---------
    faces: (N, 64 * 64) ndarray representing the faces dataset.

    Returns
    -------
    mean_face: (64, 64) ndarray which is the mean of the faces.
    """
    mean_face = mean(faces)
    return mean_face

plt.imshow(mean_face(faces).reshape((64, 64)), cmap='gray');
```



To put things into perspective, we can benchmark the two different implementation with the `%time` function in the following way:

In [19]:
```
# We have some huge data matrix, and we want to compute its mean
X = np.random.randn(100000, 20)
# Benchmarking time for computing mean
%time mean_naive(X)
%time mean(X)
pass
```

```
CPU times: user 528 ms, sys: 0 ns, total: 528 ms
Wall time: 617 ms
CPU times: user 4 ms, sys: 0 ns, total: 4 ms
Wall time: 4.16 ms
```

In [27]:
```
# Benchmarking time for computing covariance
%time cov_naive(X)
%time cov(X)
pass
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-8d0079a6e13f> in <module>()
      1 # Benchmarking time for computing covariance
      2
----> 3 get_ipython().magic('time cov(X)')
      4 pass

/opt/conda/lib/python3.6/site-packages/IPython/core/interactiveshell.py in magic(s
elf, arg_s)
   2156         magic_name, _, magic_arg_s = arg_s.partition(' ')
   2157         magic_name = magic_name.lstrip(prefilter.ESC_MAGIC)
-> 2158         return self.run_line_magic(magic_name, magic_arg_s)
   2159
   2160     #-------------------------------------------------------------------------
----

/opt/conda/lib/python3.6/site-packages/IPython/core/interactiveshell.py in run_lin
e_magic(self, magic_name, line)
   2077                 kwargs['local_ns'] = sys._getframe(stack_depth).f_locals
   2078             with self.builtin_trap:
-> 2079                 result = fn(*args,**kwargs)
   2080             return result
   2081

<decorator-gen-59> in time(self, line, cell, local_ns)

/opt/conda/lib/python3.6/site-packages/IPython/core/magic.py in <lambda>(f, *a, **
k)
    186     # but it's overkill for just that one bit of state.
    187     def magic_deco(arg):
--> 188         call = lambda f, *a, **k: f(*a, **k)
    189
    190         if callable(arg):

/opt/conda/lib/python3.6/site-packages/IPython/core/magics/execution.py in time(se
lf, line, cell, local_ns)
   1174         if mode=='eval':
   1175             st = clock2()
-> 1176             out = eval(code, glob, local_ns)
   1177             end = clock2()
   1178         else:

<timed eval> in <module>()

<ipython-input-18-28ba6bde1451> in cov(X)
     32     # iterate over the entire dataset as looping in Python tends to be slo
w
     33     N, D = X.shape
---> 34     covariance_matrix = np.zeros(D, D) # EDIT THIS
     35     Y = np.transpose(X)
     36     covariance_matrix = np.cov(Y)

TypeError: data type not understood
```

Alternatively, we can also see how running time increases as we increase the size of our dataset. In the following cell, we run `mean`, `mean_naive` and `cov`, `cov_naive` for many times on different sizes of the dataset and collect their running time. If you are less familiar with Python, you may want to spend some time understanding what the code does. **Understanding how your code scales with the size of your dataset (or dimensionality of the dataset) is crucial** when you want to apply your algorithm to larger dataset. This is really important when we propose alternative methods a more efficient algorithms to solve the same problem. We will use these techniques again later in this course to analyze the running time of our code.

In [30]:

```
def time(f, repeat=100):
    """A helper function to time the execution of a function.

    Arguments
    ---------
    f: a function which we want to time it.
    repeat: the number of times we want to execute `f`

    Returns
    -------
    the mean and standard deviation of the execution.
    """
    times = []
    for _ in range(repeat):
        start = timeit.default_timer()
        f()
        stop = timeit.default_timer()
        times.append(stop-start)
    return np.mean(times), np.std(times)
```

In [31]:

```
fast_time = []
slow_time = []

for size in np.arange(100, 5000, step=100):
    X = np.random.randn(size, 20)
    f = lambda : mean(X)
    mu, sigma = time(f)
    fast_time.append((size, mu, sigma))

    f = lambda : mean_naive(X)
    mu, sigma = time(f)
    slow_time.append((size, mu, sigma))

fast_time = np.array(fast_time)
slow_time = np.array(slow_time)
```

In [ ]:

```
fig, ax = plt.subplots()
ax.errorbar(fast_time[:,0], fast_time[:,1], fast_time[:,2], label='fast mean', linewi
ax.errorbar(slow_time[:,0], slow_time[:,1], slow_time[:,2], label='naive mean', linew
ax.set_xlabel('size of dataset')
ax.set_ylabel('running time')
plt.legend();
```

In [ ]:
```
## === FILL IN THIS, follow the approach we have above ===
fast_time_cov = []
slow_time_cov = []
for size in np.arange(100, 5000, step=100):
    X = np.random.randn(size, 20)
    f = None               # EDIT THIS
    mu, sigma = None, None # EDIT THIS
    fast_time_cov.append((size, mu, sigma))

    f = None         # EDIT THIS
    mu, sigma = None # EDIT THIS
    slow_time_cov.append((size, mu, sigma))

fast_time_cov = np.array(fast_time_cov)
slow_time_cov = np.array(slow_time_cov)
```

In [ ]:
```
fig, ax = plt.subplots()
ax.errorbar(fast_time_cov[:,0], fast_time_cov[:,1], fast_time_cov[:,2], label='fast c
ax.errorbar(slow_time_cov[:,0], slow_time_cov[:,1], slow_time_cov[:,2], label='naive
ax.set_xlabel('size of dataset')
ax.set_ylabel('running time')
plt.legend();
```

2. Affine Transformation of Dataset

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 (N, D). We would like to know what is the covariance when we apply an affine transformation $\boldsymbol{A}\boldsymbol{x}_i + \boldsymbol{b}$ with a matrix $\boldsymbol{A}$ and a vector $\boldsymbol{b}$ to 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.

In [57]:
```
# GRADED FUNCTION: DO NOT EDIT THIS LINE

# ===YOU SHOULD EDIT THIS FUNCTION===
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
    """
    affine_m = A@mean+b
    return affine_m

# ===YOU SHOULD EDIT THIS FUNCTION===
def affine_covariance(S, A, b):
    v=A@S
    y=np.transpose(A)
    affine_cov=v@y
    return affine_cov
```

Once the two functions above are implemented, we can verify the correctness our implementation. Assuming that we have some matrix $\boldsymbol{A}$ and vector $\boldsymbol{b}$.

In [33]:
```
random = np.random.RandomState(42)
A = random.randn(4,4)
b = random.randn(4)
```

Next we can generate some random dataset X

In [34]:
```
X = random.randn(100, 4)
```

Assuming that for some dataset X, the mean and covariance are m, S, and for the new dataset after affine transformation X', the mean and covariance are m' and S', then we would have the following identity:

$$m' = \text{affine\_mean}(m, A, b)$$

$$S' = \text{affine\_covariance}(S, A, b)$$

In [35]:
```
X1 = ((A @ (X.T)).T + b)  # applying affine transformation once
X2 = ((A @ (X1.T)).T + b) # and again
```

One very useful way to compare whether arrays are equal/similar is use the helper functions in `numpy.testing`. the functions in `numpy.testing` will throw an `AssertionError` when the output does not satisfy the assertion.

In [58]:
```
np.testing.assert_almost_equal(mean(X1), affine_mean(mean(X), A, b))
np.testing.assert_almost_equal(cov(X1),  affine_covariance(cov(X), A, b))
print('correct')
```

```
correct
```

Fill in the `???` below

In [32]:
```
np.testing.assert_almost_equal(mean(X2), affine_mean(mean(???), A, b))
np.testing.assert_almost_equal(cov(X2),  affine_covariance(cov(???), A, b))
print('correct')
```

```
  File "<ipython-input-32-9ffc006f59b0>", line 1
    np.testing.assert_almost_equal(mean(X2), affine_mean(mean(???), A, b))
                                                              ^
SyntaxError: invalid syntax
```

Check out the numpy documentation (https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.testing.html) for details.

If you are interested in learning more about floating point arithmetic, here is a good paper (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.6768).