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

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

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

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!

In [2]:
```
## PLEASE DO NOT EDIT THIS CELL

image_shape = (64, 64)
# Load faces data
dataset = fetch_olivetti_faces(data_home='./')
faces = dataset.data

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

```
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.

In [3]:
```python
## PLEASE DO NOT EDIT THIS CELL

from ipywidgets import interact
```

In [4]:
```python
## PLEASE DO NOT EDIT THIS CELL

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

In [5]:
```python
## PLEASE DO NOT EDIT THIS CELL

@interact(n=(0, len(faces)-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=('widget-interact'…
```

The purpose of the following block is to compute the mean and covariance of a dataset of size (N,D), where N is the number of data points and D is the dimensionality of each data point.

1. Mean and Covariance of a Dataset

In [8]:
```python
# GRADED FUNCTION: DO NOT EDIT THIS LINE
def mean_naive(X):
    """Compute the sample mean for a dataset by iterating over the dataset.

    Args:
        X: `ndarray` of shape (N, D) representing the dataset. N
        is the size of the dataset (the number of data points)
        and D is the dimensionality of each data point.
    Returns:
        mean: `ndarray` of shape (D, ), the sample mean of the dataset `X`.
    """
    # YOUR CODE HERE
    ### Uncomment and edit the code below
#     iterate over the dataset and compute the mean vector.
    mean = np.zeros((X.shape[1]))
    N = X.shape[0]
    for n in range(X.shape[0]):
        # Update the mean vector
        mean += X[n]
    mean = mean / N
    return mean
```

In [9]:
```python
## PLEASE DO NOT EDIT THIS CELL

from numpy.testing import assert_allclose

# Test case 1
X = np.array([[0., 1., 1.],
              [1., 2., 1.]])
expected_mean = np.array([0.5, 1.5, 1.])
assert_allclose(mean_naive(X), expected_mean, rtol=1e-5)

# Test case 2
X = np.array([[0., 1., 0.],
              [2., 3., 1.]])
expected_mean = np.array([1., 2., 0.5])
assert_allclose(mean_naive(X), expected_mean, rtol=1e-5)

# Test covariance is zero
X = np.array([[0., 1.],
              [0., 1.]])
expected_mean = np.array([0., 1.])
assert_allclose(mean(X), expected_mean, rtol=1e-5)

### Some hidden tests below
### ...
```

In [12]:
```
# GRADED FUNCTION: DO NOT EDIT THIS LINE
def cov_naive(X):
    """Compute the sample covariance for a dataset by iterating over the dataset.

    Args:
        X: `ndarray` of shape (N, D) representing the dataset.
        N is the size of the dataset (the number of data points)
        and D is the dimensionality of each data point.
    Returns:
        ndarray: ndarray with shape (D, D), the sample covariance of the dataset `X`.
    """
    # YOUR CODE HERE
    ### Uncomment and edit the code below
    N, D = X.shape
    ### Edit the code below to compute the covariance matrix by iterating over the dataset.
    covariance = np.zeros((D, D))
    ### Update covariance
    mean = np.zeros(D)
    for i in range(N):
        mean += X[i]
    mean /= N
    for i in range(N):
        diff = X[i] - mean
        covariance += np.outer(diff, diff)
    covariance /= N
    ###
    return covariance
```

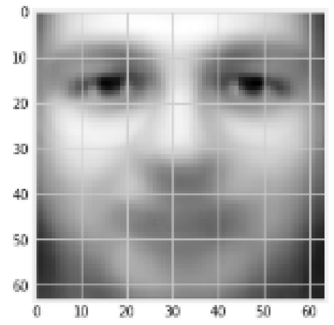
In [13]:
```
## PLEASE DO NOT EDIT THIS CELL

from numpy.testing import assert_allclose

# Test case 1
X = np.array([[0., 1.],
              [1., 2.],
     [0., 1.],
     [1., 2.]])
expected_cov = np.array(
    [[0.25, 0.25],
    [0.25, 0.25]])

assert_allclose(cov_naive(X), expected_cov, rtol=1e-5)

# Test case 2
X = np.array([[0., 1.],
              [2., 3.]])
expected_cov = np.array(
    [[1., 1.],
    [1., 1.]])

assert_allclose(cov_naive(X), expected_cov, rtol=1e-5)

# Test covariance is zero
X = np.array([[0., 1.],
              [0., 1.],
              [0., 1.]])
expected_cov = np.zeros((2, 2))

assert_allclose(cov_naive(X), expected_cov, rtol=1e-5)
```

In [14]:
```
# GRADED FUNCTION: DO NOT EDIT THIS LINE
def cov(X):
    """Compute the sample covariance for a dataset.

    Args:
        X: `ndarray` of shape (N, D) representing the dataset.
        N is the size of the dataset (the number of data points)
        and D is the dimensionality of each data point.
    Returns:
        ndarray: ndarray with shape (D, D), the sample covariance of the dataset `X`.
    """
    # YOUR CODE HERE

    # It is possible to vectorize our code for computing the covariance with matrix multiplications,
    # 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 if you choose to use np.cov,
    # be sure to pass in bias=True.
    ### Uncomment and edit the code below
    N, D = X.shape
#     ### Edit the code to compute the covariance matrix
    covariance_matrix = np.zeros((D, D))
    ### Update covariance_matrix here
    mean = np.mean(X, axis=0)
    diff = X - mean
    covariance_matrix = (diff.T @ diff) / N
    ###
    return covariance_matrix
```

In [15]:
```
## PLEASE DO NOT EDIT THIS CELL

from numpy.testing import assert_allclose

# Test case 1
X = np.array([[0., 1.],
              [1., 2.],
     [0., 1.],
     [1., 2.]])
expected_cov = np.array(
    [[0.25, 0.25],
    [0.25, 0.25]])

assert_allclose(cov(X), expected_cov, rtol=1e-5)

# Test case 2
X = np.array([[0., 1.],
              [2., 3.]])
expected_cov = np.array(
    [[1., 1.],
    [1., 1.]])

assert_allclose(cov(X), expected_cov, rtol=1e-5)

# Test covariance is zero
X = np.array([[0., 1.],
              [0., 1.],
              [0., 1.]])
expected_cov = np.zeros((2, 2))

assert_allclose(cov(X), expected_cov, rtol=1e-5)

### Some hidden tests below
### ...
```

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

In [16]:
```python
## PLEASE DO NOT EDIT THIS CELL

def mean_face(faces):
    return faces.mean(axis=0).reshape((64, 64))

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



One of the advantage of writing vectorized code is speedup gained when working on larger dataset. 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 [17]:
```python
# We have some HUUUGE data matrix which we want to compute its mean
X = np.random.randn(1000, 20)
# Benchmarking time for computing mean
%time mean_naive(X)
%time mean(X)
pass
```

```
CPU times: user 736 µs, sys: 325 µs, total: 1.06 ms
Wall time: 1.07 ms
CPU times: user 719 µs, sys: 317 µs, total: 1.04 ms
Wall time: 1.04 ms
```

In [18]:
```python
# Benchmarking time for computing covariance
%time cov_naive(X)
%time cov(X)
pass
```

```
CPU times: user 7.86 ms, sys: 0 ns, total: 7.86 ms
Wall time: 7.82 ms
CPU times: user 391 µs, sys: 171 µs, total: 562 µs
Wall time: 282 µs
```

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 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.

In [19]:
```python
# GRADED FUNCTION: DO NOT EDIT THIS LINE

def affine_mean(mean, A, b):
    """Compute the mean after affine transformation
    Args:
        mean: `ndarray` of shape (D,), the sample mean vector for some dataset.
        A, b: `ndarray` of shape (D, D) and (D,), affine transformation applied to x
    Returns:
        sample mean vector of shape (D,) after affine transformation.
    """
    # YOUR CODE HERE
    ### Uncomment and edit the code below
#     ### Edit the code below to compute the mean vector after affine transformation
    affine_m = np.zeros(mean.shape) # affine_m has shape (D,)
    ### Update affine_m
    affine_m = A @ mean + b
    ###
    return affine_m
```

In [20]:

```
# GRADED FUNCTION: DO NOT EDIT THIS LINE
def affine_covariance(S, A, b):
    """Compute the covariance matrix after affine transformation

    Args:
        S: `ndarray` of shape (D, D), the sample covariance matrix for some dataset.
        A, b: `ndarray` of shape (D, D) and (D,), affine transformation applied to x

    Returns:
        the sample covariance matrix of shape (D, D) after the transformation
    """
    # YOUR CODE HERE
    ### Uncomment and edit the code below
    ### EDIT the code below to compute the covariance matrix after affine transformation
    affine_cov = np.zeros(S.shape) # affine_cov has shape (D, D)
    ### Update affine_cov
    affine_cov = A @ S @ A.T
    ###
    return affine_cov
```

In [21]:

```
## PLEASE DO NOT EDIT THIS CELL

from numpy.testing import assert_allclose

A = np.array([[0, 1], [2, 3]])
b = np.ones(2)
m = np.full((2,), 2)
S = np.eye(2)*2

expected_affine_mean = np.array([ 3., 11.])
expected_affine_cov = np.array(
    [[ 2.,  6.],
    [ 6., 26.]])

assert_allclose(affine_mean(m, A, b), expected_affine_mean, rtol=1e-4)
### Some hidden tests below
### ...
```

```
In [22]: ## PLEASE DO NOT EDIT THIS CELL

         from numpy.testing import assert_allclose

         A = np.array([[0, 1], [2, 3]])
         b = np.ones(2)
         m = np.full((2,), 2)
         S = np.eye(2)*2

         expected_affine_cov = np.array(
             [[ 2.,  6.],
             [ 6., 26.]])

         assert_allclose(affine_covariance(S, A, b),
                         expected_affine_cov, rtol=1e-4)

         ### Some hidden tests below
         ### ...
```

Once the two functions above are implemented, we can verify the correctness our implementation. Assuming that we have some A and b.

```
In [23]: random = np.random.RandomState(42)
         A = random.randn(4,4)
         b = random.randn(4)
```

Next we can generate some random dataset X

```
In [24]: 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 [25]: X1 = ((A @ (X.T)).T + b)  # applying affine transformation once
         X2 = ((A @ (X1.T)).T + b) # twice
```

One very useful way to compare whether arrays are equal/similar is use the helper functions in `numpy.testing`.

Check the Numpy documentation for details.

If you are interested in learning more about floating point arithmetic, here is a good paper.

```
In [26]: np.testing.assert_allclose(mean(X1), affine_mean(mean(X), A, b))
         np.testing.assert_allclose(cov(X1),  affine_covariance(cov(X), A, b))
```

```
In [27]: np.testing.assert_allclose(mean(X2), affine_mean(mean(X1), A, b))
         np.testing.assert_allclose(cov(X2),  affine_covariance(cov(X1), A, b))
```

```
In [ ]:
```

```
In [ ]:
```