ML HW7 report

資科工碩一

0756138

黃伯凱

1. PCA

```
def pca(X,k)
    #mean of each feature
    n_samples, n_features = X.shape
    mean = np.mean(X, axis=0)

    # Calculate norm
    norm_X=X-mean

    # Calculate covariance matrix
    cov_matrix=np.dot(np.transpose(norm_X),norm_X)

    # Calculate the eigenvectors and eigenvalues
    # Sort it according to eigenvalues ascending
    eig_val, eig_vec = np.linalg.eig(cov_matrix)
    idx = eig_val.argsort()[::-1]
    eig_val = eig_val[idx]
    eig_vec = eig_vec[:,idx]

    # select the top k eig_vec
    final_eigvec=-np.array([ele for ele in eig_vec.T[ : k]])

    # Project to lower dimension
    data=np.dot(norm_X,np.transpose(final_eigvec))

    return np.real(data)
```

最後希望降到 k dimension

算出 training data 的平均值

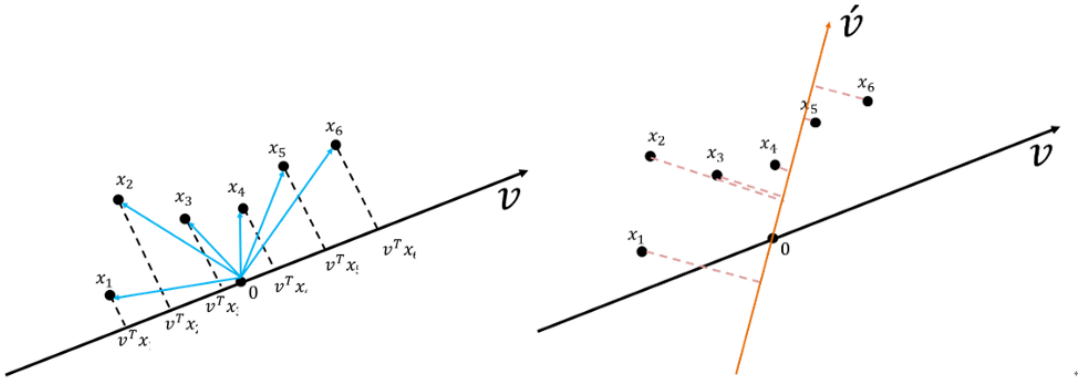
找出 covariance matrix 的 eigenvalue, eigenvector 並由大到小排序

選前 k 大的 eigenvector

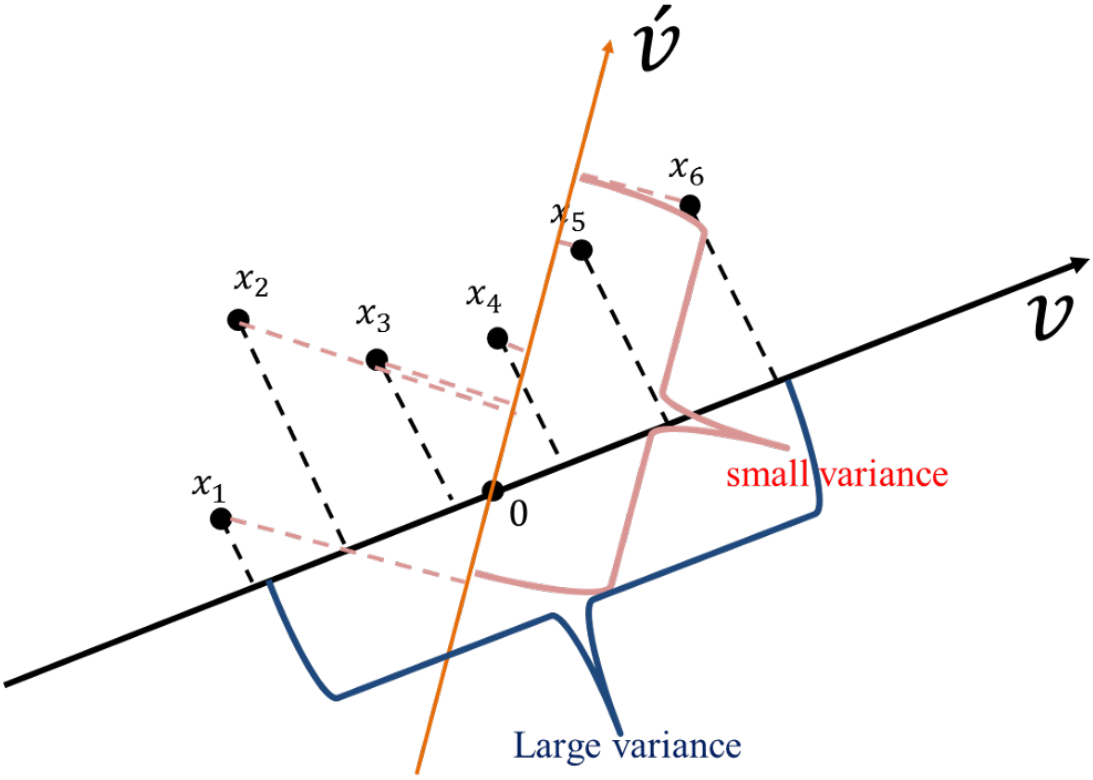
投影到 eigenvector 即為降維的動作

```
myPCA = pca(X_train, 2)

title = "MyPCA"
plot_class(myPCA, T_train, title)
```

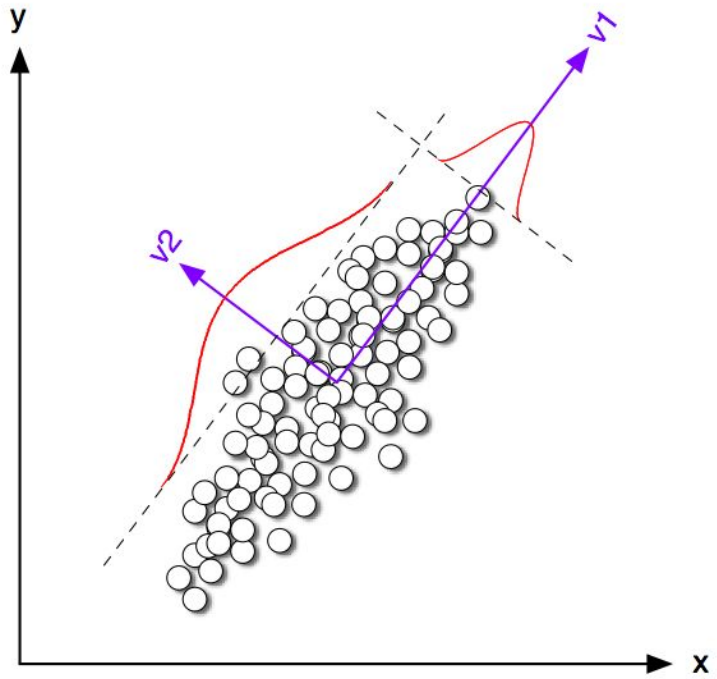
PCA 是利用降維來達避免因為高維而無法確實分類的一種方法。他的方法是把 data 投影到某個 vector 上，以下有兩個不同的 vector 投影結果：



如果把 data 通通投影到左圖的 vector v，可以清楚的分出每一個 data 之間的差異。但是如果把 data 都投影到右圖的 vector v'，有些 data 反而會靠得很近，甚至難以辨認。所以希望找到不改變相對距離的向量，而這種向量其實就是 data 的 eigenvector。

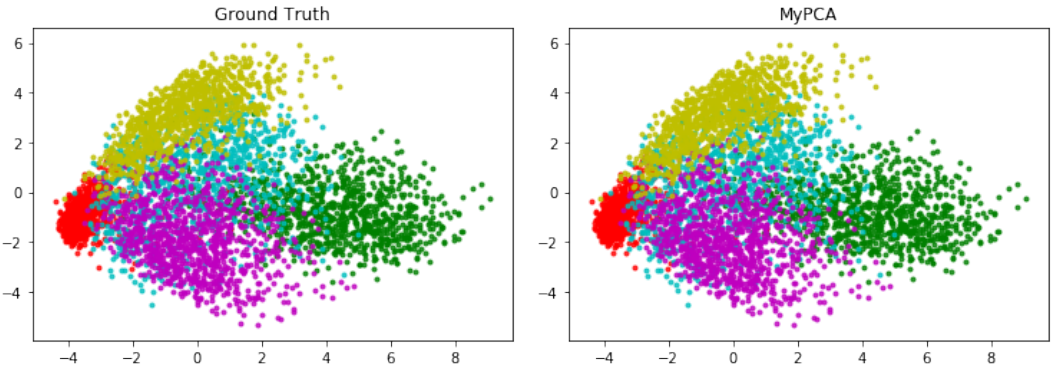


以二維的 data 為例，v1 和 v2 都是這個 data 的 eigenvector，而 v1、v2 分別是 eigenvalue 較大、較小的 eigenvector。下圖則可以很清楚的看出來，如果把 data 投影到 v1，可以把 data 分的較散，所以區分 data 的時候比較好分辨。但如果把 data 投影到 v2，則 data 都會擠在一起，最後難以分辨。



因此 PCA 要先算出 data 的 covariance，利用 covariance 算出 eigenvalue, eigenvector 之後，再把 data 投影到 eigenvector 上。

以下是 PCA 的結果：



這邊的 ground truth 是用 sklearn 的 function 畫出來的，用來比對自己寫的 PCA 有沒有做錯。我自己寫的 PCA，eigenvector 最後有乘上-1 才會跟 sklearn 算的一模一樣。不過其實這一步是沒有關係的，因為 eigenvector 乘上一個負號指示方向相反而已，對結果並不會有影響。

2. LDA

```
# Seperate data of different classes into array 'classes'
classes = []
for c in range(1, 6):
    tmp = []
    for idx, data in enumerate(X_train):
        if T_train[idx] == c:
            tmp.append(data)
    classes.append(tmp)
classes = np.array(classes)
```

這邊 initial 了一個 classes 的 list，相同 label 的 training data 都先分到同一個 tmp list 裡，這些 tmp 即為 classes 的每一個 row。

```
# Calculate class mean
mean = []
for class_datas in classes:
    mean.append(np.mean(class_datas, axis=0))
mean = np.array(mean)
```

算出每一個 class 當中每一個 pixel 的 mean

```
# Sw : within-class scatter
# Calculate Sw
Sw = np.zeros((784, 784))
for c, class_datas in enumerate(classes):
    for data in class_datas:
        tmp = data - mean[c]
        tmp = np.expand_dims(tmp, axis=0)
        Sw = Sw + np.matmul( tmp.T , tmp )
```

Sw 是 within-class scatter，每一張圖片要先減掉相對應 class 的 pixel mean，減完之後要做自相關，再加到 Sw 裡。

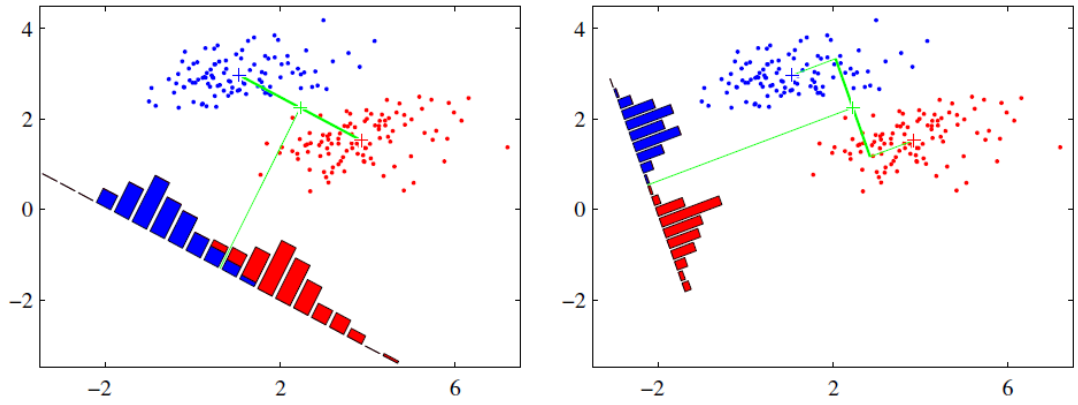
```
# Sb : between-class scatter
# Calculate Sb
total_mean = np.mean(X_train, axis=0)
Sb = np.zeros((784, 784))
for c in range(len(classes)):
    class_len = len(classes[c])
    tmp = mean[c] - total_mean
    tmp = np.expand_dims(tmp, axis=0)
    Sb = Sb + class_len*np.matmul( tmp.T , tmp )
```

Sb 是 between-class scatter，在算 Sb 之前要先算出所有圖片每個 pixel 的 mean。利用每個 class 的 mean 減去全部圖片的 mean 之後做自相關，再加到 Sb。

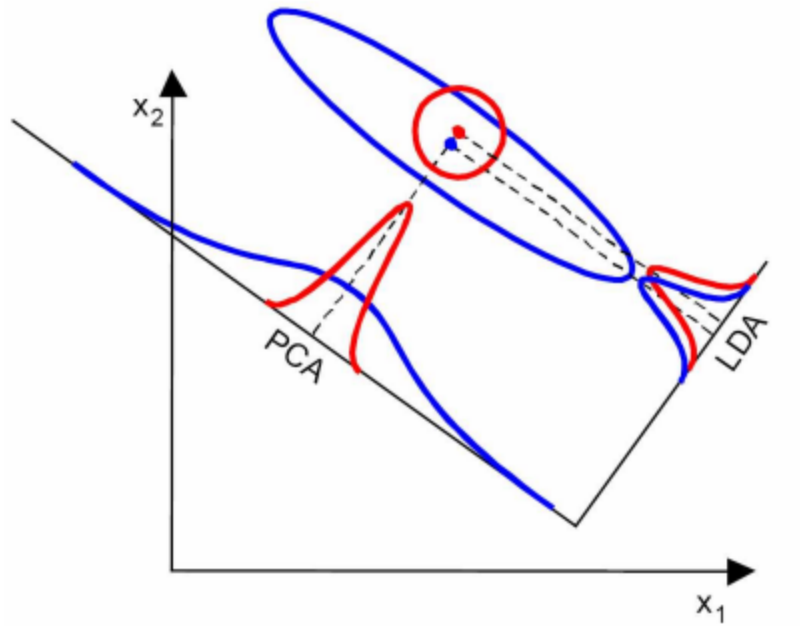
```
# Calculate eigenvalue, eigenvector of (Sw)**-1 * Sb
Sw_inv = np.linalg.pinv(Sw)
Target = np.matmul(Sw_inv, Sb)
eig_val, eig_vec = np.linalg.eig(Target)
idx = eig_val.argsort()[::-1]
eig_val = eig_val[idx]
eig_vec = eig_vec[:,idx]

# select the top k eig_vec
final_eigvec = np.real(np.array([ele for ele in eig_vec.T[: 2]]))
myLDA = np.matmul(X_train, final_eigvec.T)

title = "MyLDA"
plot_class(myLDA, T_train, title)
```

算出(Sw**-1)*Sb 的 covariance 之後，找到 covariance 的 eigenvalue, eigenvector，再挑出 k 個 eigenvector。最後將 data 投影到這 k 個 eigenvector 上。

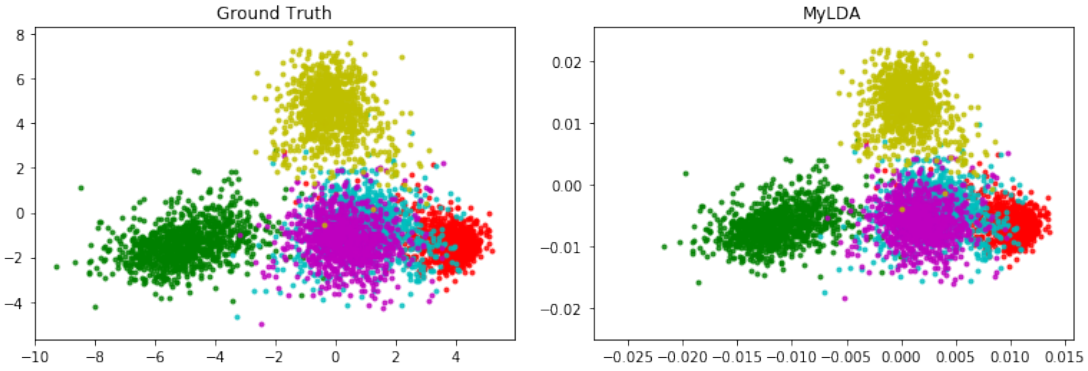
LDA 和 PCA 一樣的地方是在：一樣是將 data 投影到 eigenvector 上。但他不一樣的地方是在：他希望**不同 class 的 data** 之間的**差異大**、**相同 class 的 data** 之間的**差異小**。如下圖所示，左圖不同 class 之間的差異小，因此在 testing 時容易分錯。希望可以做到和右圖一樣的結果，不同 class 才能被清楚分開。



但是當然還是會有一些例外，如下圖的 data。如果用 PCA 會得到投影到左邊向量的結果，LDA 則會得到右邊。LDA 在這種情況下幾乎會把 2 的 class 都分為同一個 class，所以其實哪一種 model 比較好其實還是要看 data 的分佈大概是長怎樣。



以下是 LDA 的結果：



左圖的 ground truth 一樣是用 sklearn 算出來的，用來比對自己的答案是否正確。這兩張的分佈大致上一樣，只有 scale 和 sklearn 不一樣，不清楚是不是我 eigenvector 的處理方式和 sklearn 不一樣，初步猜測或許是 sklearn 在 scale 太小的時候會自動把間距拉大以免誤判。

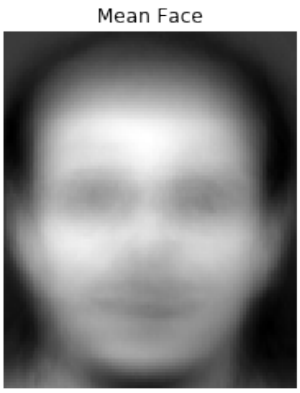
3. Eigenfaces

```
dataArray = np.empty((0, 112*92))

# Read all images into array 'dataArray'
for root, dirs, files in os.walk('att_faces'):
    for directory in dirs:
        for root_, dirs_, files_ in os.walk('att_faces/'+directory):
            for file in files_:
                img = cv2.imread('att_faces/'+directory+'/'+file, 0)
                img_row = np.array(img, dtype='float64').flatten()
                img_row = np.expand_dims(img_row, axis=0)
                dataArray = np.vstack((dataArray, img_row))
```

利用 os.walk 和 cv2.imread 來把所有資料夾裡的所有圖片都讀進 dataArray 裡，並將其存成 grayscale。我的方法是把每一張照片用一列存在 dataArray 裡。

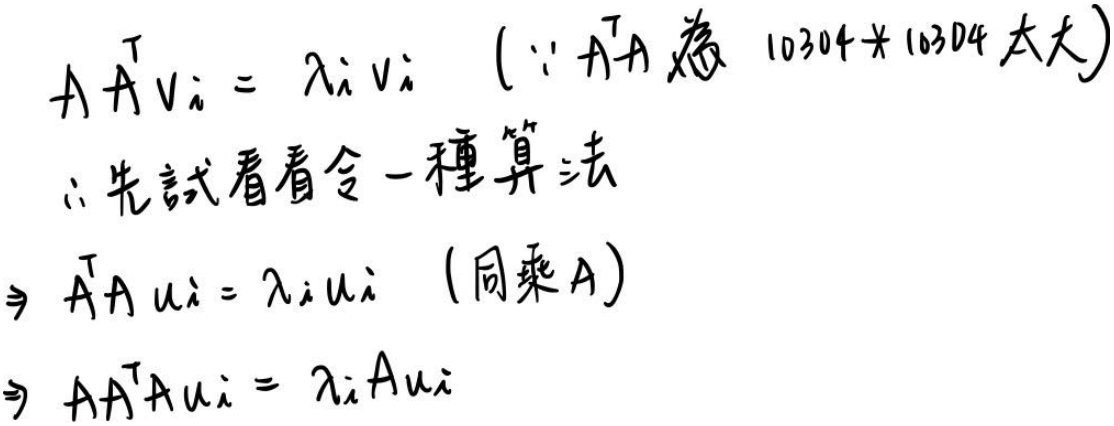
```
# Calculate Mean Face
dataAvg = np.mean(dataArray, axis=0)
tmp = dataAvg.reshape(112, 92)
plt.imshow(tmp, cmap='gray')
plt.axis('off')
plt.title("Mean Face")
plt.show()
```

把所有圖片每個 pixel 的 mean 算出來，把它 reshape 成圖片的 size。最後再畫出來，就是 mean face。



```
# Number of eigenvectors to pick
eig_k = 25

# Calculate the eigenvalue, eigenvector of At*A
# However, the shape of At*A is 10304*10304
# Since it is too large, we calculate A*At instead
dataDis = dataArray - dataAvg
A_AT = np.cov(dataDis)
eig_val, eig_vec = np.linalg.eig(A_AT)
idx = eig_val.argsort()[::-1]
eig_val = eig_val[idx]
eig_vec = eig_vec[:, idx]
eig_vec = eig_vec.T[:eig_k]
```

其實 eigenface 就是利用 PCA 達成的，所以接下來就是要算 dataDis 自相關的 eigenvalue, eigenvector。但是由於現在 dataDis 是(400, 10304)，dataDis 用 A 簡寫。如果要算 AT*A 的 covariance，就是要算(10304, 10304)極大矩陣的 eigenvalue, eigenvector。所以有以下這樣的快速算法：



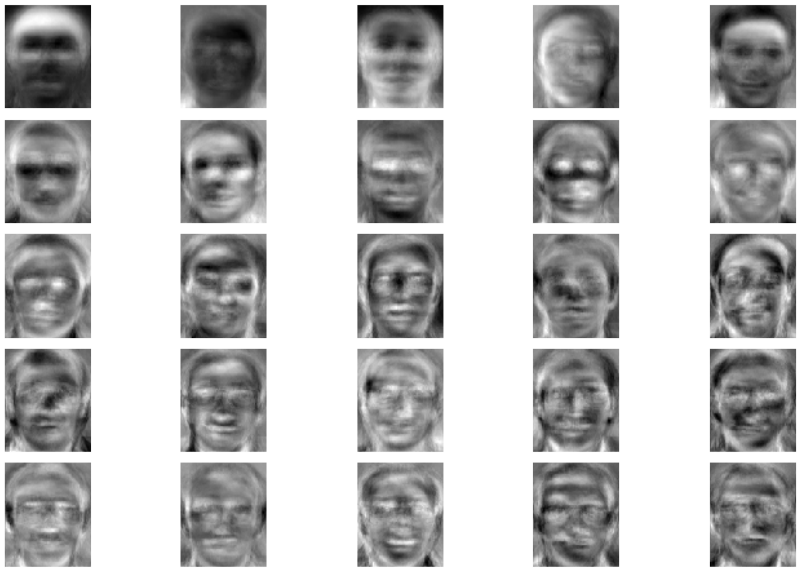
此時發現 Au 其實就是 v，而我們的目的剛好就是 v，所以我們只要先算出 At*A 的 eigenvector 後，將其*A 並且別忘記 normalize，就可以得到原本的 v。

```
eig_faces = np.matmul(eig_vec, dataDis)

# Normalize eigenfaces again since we multiply dataDis again
for i, eig_face in enumerate(eig_faces):
    eig_faces[i] = eig_faces[i] / np.linalg.norm(eig_face)

# Reshape eigenfaces to the image size
eig_faces = eig_faces.reshape(eig_k, 112, 92)
```

這邊就是上述乘回一個 A 的程式碼，但因為我是把圖片存成一列，異於我上面推導的式子，所以是 u*A 而非 A*u。

```
# Plot
plt.figure(figsize=(12,8))
for idx, eig_face in enumerate(eig_faces):
    plt.subplot(5,5,idx+1)
    plt.imshow(eig_face, cmap='gray')
    plt.axis('off')
plt.subplots_adjust(wspace=0.1, hspace=0.1)
plt.show()
```

這邊就是把 eigenfaces 都 show 出來看看



```
# Random pick 10 faces
from random import sample
rand_idx = sample(range(1,len(dataArray)), 10)
print(rand_idx)
```

隨機挑出 10 張 face

```
eig_faces = eig_faces.reshape(eig_k, 10304)

# Project dataDis to vector of eigenfaces
# Inorder to build Weight Matrix
# Each random-pick has k weights for k eigenfaces
W = []
for idx in rand_idx:
    tmp = []
    for eig_face in eig_faces:
        tmp.append(np.dot(dataDis[idx], eig_face))
    W.append(tmp)
W = np.array(W)
```

把單張 face 的 dataDis 投影到 eig_face 上（其實就是在做 PCA 的投影），這樣可以得到每一張圖對每一個 eigenface 的 weight。

```
# Reconstructing the random-pick faces
# For each image, (DataAvg + w[i]*eigenfaces[i]) will be the final image
res = []
for W_ in W:
    tmp = np.copy(dataAvg)
    for idx, w in enumerate(W_):
        tmp += w*eig_faces[idx]
    res.append(tmp)
res = np.array(res)
```

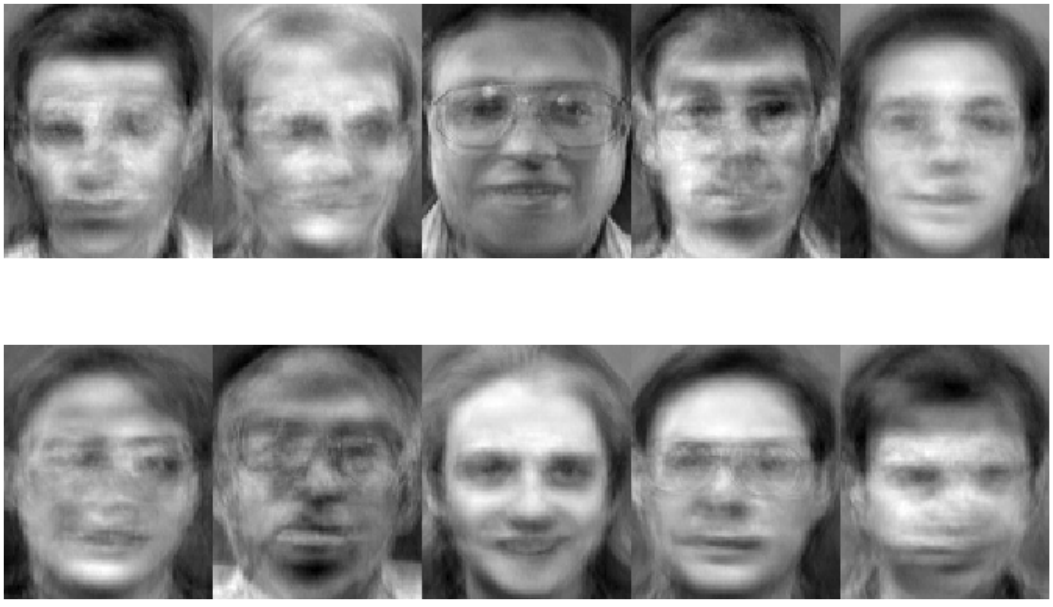
把這 10 張臉的 w 乘上所有 eigen_face，最後再加上 dataAvg，即可得到 reconstruct 的結果

```
# Plot
res = res.reshape(10, 112, 92)

plt.figure(figsize=(12,8))
for i, idx in enumerate(rand_idx):
    plt.subplot(2,5, i+1)
    plt.imshow(dataArray[idx].reshape(112, 92), cmap='gray')
    plt.axis('off')
plt.subplots_adjust(wspace=0, hspace=0)
plt.show()

print('\n---------\n')

plt.figure(figsize=(12,8))
for idx, res_ in enumerate(res):
    plt.subplot(2,5, idx+1)
    plt.imshow(res_, cmap='gray')
    plt.axis('off')
plt.subplots_adjust(wspace=0, hspace=0)
plt.show()
```

這是挑到的 10 張臉

這是我 reconstruct (k=25)的結果

這是我 reconstruct (k=50)的結果



這是我 reconstruct (k=70)的結果

這是我 reconstruct (k=100)的結果