ML hw7

0756607 范姜良瑜

Principle component analysis (PCA)

Assume we have data of $N \times dim$. N is the number of samples and dim is the number of features for each sample. `mnist_X.csv` provides data of shape 5000×784.

1. **Center each feature data to zero-mean**

```
x = X - (np.mean(X, axis=0))
```

2. **Compute covariance matrix Σ to get eigenvalues and eigenvectors**

$\Sigma = \frac{1}{n} \sum_{i=1}^{n} \left(X_i^T \ X_i \right)$

```
covariance_mat = np.cov(x.T)
eig_vals, eig_vecs = LA.eig(covariance_mat)
```

3. **Select principle components**
The eigenvectors with higher eigenvalues are more informative than those with low eigenvalues. As a result, we rank the eigenvectors with eigenvalues and select the top k eigenvectors.

```
eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:,i]) for i in range(len(eig_vals))]
eig_pairs = sorted(eig_pairs,key=lambda k: k[0], reverse=True)
```

We use **explained variance** to see how informative of each eigenvector (principle component).

```
eigv_sum = sum(eig_vals)
for i, j in enumerate(eig_pairs):
    print('eigenvalue {0:}: {1:.2%}'.format(i+1, (j[0]/eigv_sum).real))
```

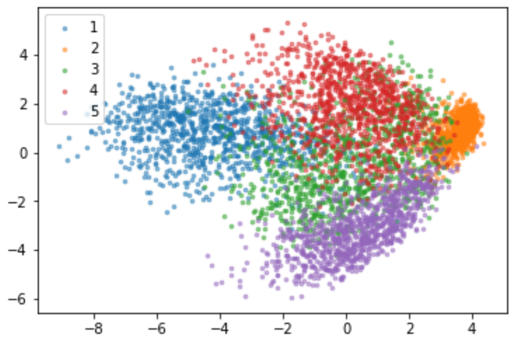
Sample output:

```
eigenvalue 1: 14.78%
eigenvalue 2: 8.17%
eigenvalue 3: 6.31%
eigenvalue 4: 5.99%
eigenvalue 5: 5.11%
eigenvalue 6: 3.88%
eigenvalue 7: 3.38%
eigenvalue 8: 2.66%
eigenvalue 9: 2.55%
eigenvalue 10: 2.19%
...
```

Here, we project the data to 2D space, i.e. $k = 2$.

```
N, dim = x.shape
```

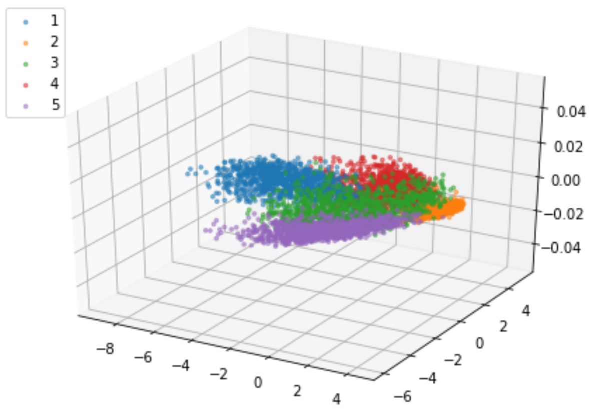
```python
w = np.hstack((eig_pairs[0][1].reshape(dim,1), eig_pairs[1][1].reshape(dim,1)))
x_PCA = x.dot(w)
# plot
for l,c in zip(labels,('C0', 'C1', 'C2', 'C3', 'C4')):
    target_data = x_PCA[y == l]
    plt.scatter(target_data[:,0].real, target_data[:,1].real, c=c, s=7, alpha=0.5)
plt.legend(labels.astype(int), loc='upper left')
plt.show()
```



project 784-dimensional data to 2D space with PCA

Also, we try to map the data to 3D space by taking top 3 eigenvectors.

```python
num_dims = 3
w = np.hstack((eig_pairs[0][1].reshape(dim,1), eig_pairs[1][1].reshape(dim,1)))
for i in range(2,num_dims):
    w_3 = np.hstack((w, eig_pairs[i][1].reshape(dim,1)))
x_PCA_3 = x.dot(w_3)
```



project 784-dimensional data to 3D space with PCA

Linear discriminant analysis (LDA)

Assume we have data of $N \times dim$. N is the number of samples and dim is the number of features for each sample. `mnist_X.csv` provides data of shape 5000×784.

1. **Compute dim-dimensional mean vectors m_i**

$$m_i = \frac{1}{n_i} \sum_{x \in D_i}^{n} x_k$$

```python
mean_vectors = []
for l in labels:
    # column mean, i.e. mean of each feature, in the same class
    mean_vectors.append(np.mean(x[y == l], axis=0))
```

2. **Compute scatter matrices**

First, we compute the within-class scatter matrix S_w. S_i is the scatter matrix for each class.

$$S_w = \sum_{i=1}^{c} S_i$$

$$S_i = \sum_{x \in D_i}^{n} (x - m_i)\,(x - \ m_i)^T$$

```python
dim = x.shape[1]
s_w = np.zeros((dim, dim))
for l, mv in zip(labels, mean_vectors):
    scatter_mat = np.zeros((dim, dim))
    for d in x[y == l]:
        d = d.reshape(dim, 1)
        mv = mv.reshape(dim, 1)
        scatter_mat += (d - mv).dot((d - mv).T)
    s_w += scatter_mat
```

Then, we compute the between-class scatter matrix S_b. N_i is the number of data with label i. m_i is the i^{th} mean in the mean vector. m is the overall means of each feature across classes.

$$S_b = \sum_{i=1}^{c} N_i (m_i - m)(m_i - m)^T$$

```python
overall_mean = np.mean(x, axis=0)
s_b = np.zeros((dim, dim))
for l, mv in zip(labels, mean_vectors):
    # n is the number of data with label l
    n = x[y == l].shape[0]
    overall_mean = overall_mean.reshape(dim, 1)
    mv = mv.reshape(dim, 1)
    s_b += n * (mv - overall_mean).dot((mv - overall_mean).T)
```

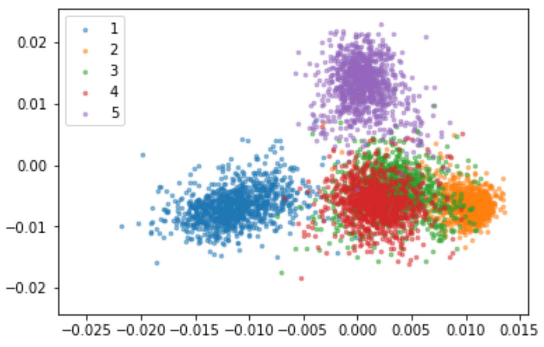
3. **Eigendecomposition of $S_W^{-1} S_B$**

```python
# check if s_w is a singularmatrix
if LA.det(s_w) == 0:
    eig_vals, eig_vecs = LA.eig(LA.pinv(s_w).dot(s_b)) # use pseudo inverse
else:
    eig_vals, eig_vecs = LA.eig(LA.inv(s_w).dot(s_b))
```

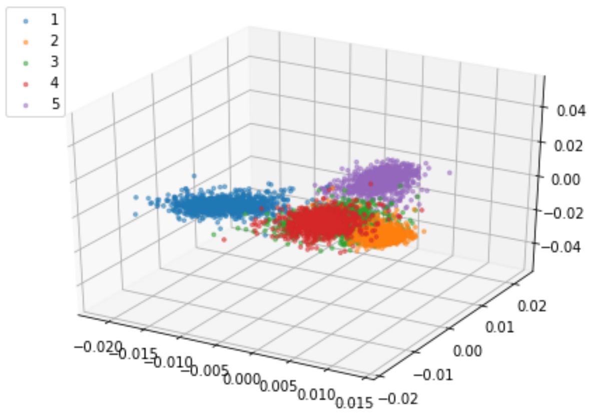
4. **Select linear discriminants**

Here, we select the top 2 linear discriminants to project the data onto a 2D space.

```python
eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:,i]) for i in range(len(eig_vals))]
eig_pairs = sorted(eig_pairs,key=lambda k: k[0], reverse=True)
w = np.hstack((eig_pairs[0][1].reshape(dim,1), eig_pairs[1][1].reshape(dim,1)))
x_LDA = x.dot(w)
# plot
for l,c in zip(labels,('C0', 'C1', 'C2', 'C3', 'C4')):
    target_data = x_LDA[y == l]
    plt.scatter(target_data[:,0].real, target_data[:,1].real, c=c, s=7, alpha=0.5)
plt.legend(labels.astype(int), loc='upper left')
plt.show()
```



project 784-dimensional data to 2D space with LDA



project 784-dimensional data to 3D space with LDA

Symmetric SNE and t-SNE

SNE

Convert the euclidean distances between data points to conditional probabilities that represent similarities.

$$p_{j|i} = \frac{\exp\left(-||x_i - x_j||^2 / 2\sigma_i^2\right)}{\sum_{k\neq i}\exp\left(-||x_i - x_k||^2 / 2\sigma_i^2\right)}$$

The probability of point x_j being a neighbor of point x_i is proportional to the distance between these two points.
note: $p_{i|i} = 0$

y is an $N \times 2$ matrix that is our 2D representation of x.

$$q_{j|i} = \frac{\exp\left(-||y_i - y_j||^2\right)}{\sum_{k \neq i} \exp(-||y_i - y_k||^2)}$$

Goal: pick points in y such that q is similar to p.
Minimize the following cost by using gradient descent:

$$C = \sum_i KL(P_i||Q_i) = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}$$

Pick radius of Gaussian σ_i
The **perplexity** of any of the rows of the conditional probabilities matrix P is defined as:

$$Perp(P_i) = 2^{H(P_i)}$$

$$H(P_i) = -\sum_j p_{j|i} \log_2 p_{j|i}$$

Higher perplexity will increase the number of neighbors each point has.
Perform **binary search** over each σ_i until $Perp(P_i) =$ the desired perplexity.

Symmetric SNE
Minimize a KL divergence over the joint probability distributions with entries p_{ij} and q_{ij}.

$$p_{ij} = \frac{p_{i|j} + p_{j|i}}{2N}$$

$$q_{ij} = \frac{\exp\left(-||y_i - y_j||^2\right)}{\sum_{k \neq l} \exp(-||y_k - y_l||^2)}$$

```python
def q_joint(Y):
    """
        Given low-dimensional representations Y, compute matrix of joint
        probabilities with entries q_ij.
    """
    # compute distances from every y_i to y_j
    dists = neg_squared_euclidean_dists(Y)
    exp_dists = np.exp(dists)
    # let q_ii = 0
    np.fill_diagonal(exp_dists, 0.)
    # divide by the sum of the exponential matrix
    Q = exp_dists / np.sum(exp_dists)
    return Q, None
```

Use the following gradient to update the i'th row of our low-dimensional representation y.

$$\frac{\partial C}{\partial y_i} = 2 \sum_j (p_{ij} - q_{ij})(y_i - y_j)$$

note: the coefficient on the handout is 2, but some use 4 according to some online articles.

```python
def grad_symmetric_sne(P, Q, Y, _):
    pq_diff = P - Q                                             # N x N
    pq_expand = np.expand_dims(pq_diff, 2)                      # N x N x 1
    y_diff = np.expand_dims(Y, 1) - np.expand_dims(Y, 0)        # N x N x 2
    grad = 2. * (pq_expand * y_diff).sum(1)                     # N x 2
    return grad
```

t-SNE
In high dimension, Gaussian distribution is used to turn euclidean distances into probabilities. In low dimension, Student's t-distribution is used to alleviate crowding problem.

$$p_{ij} = \frac{\exp\left(-||x_i - x_j||^2 / 2\sigma_i^2\right)}{\sum_{k \neq l} \exp\left(-||x_l - x_k||^2 / 2\sigma_l^2\right)}$$

$$q_{ij} = \frac{\left(1+||y_i - y_j||^2\right)^{-1}}{\sum_{k \neq l} (1+||y_k - y_l||^2)^{-1}}$$

$$\frac{\partial C}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j) \left(1 + ||y_i - y_j||^2\right)^{-1}$$

Differences between symmetric SNE and t-SNE
In t-SNE, Student's t-distribution is to turn euclidean distances into probabilities in low dimension. As a result, the computation for gradients are also different.

symmetric SNE	t-SNE
$q_{ij} = \frac{\exp\left(-\lvert\lvert y_i - y_j\rvert\rvert^2\right)}{\sum_{k \neq l} \exp(-\lvert\lvert y_k - y_l\rvert\rvert^2)}$	$q_{ij} = \frac{\left(1+\lvert\lvert y_i - y_j\rvert\rvert^2\right)^{-1}}{\sum_{k \neq l} (1+\lvert\lvert y_k - y_l\rvert\rvert^2)^{-1}}$
$\frac{\partial C}{\partial y_i} = 2 \sum_j (p_{ij} - q_{ij})(y_i - y_j)$	$\frac{\partial C}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j) \left(1 + \lvert\lvert y_i - y_j\rvert\rvert^2\right)^{-1}$

Redefine the provided `tsne(.)` as `sne(.)`. Add new parameters `q_func` and `grad_func`. The default setting is symmetric SNE. The implementation of `q_joint(.)` and `grad_symmetric_sne(.)` is shown above.

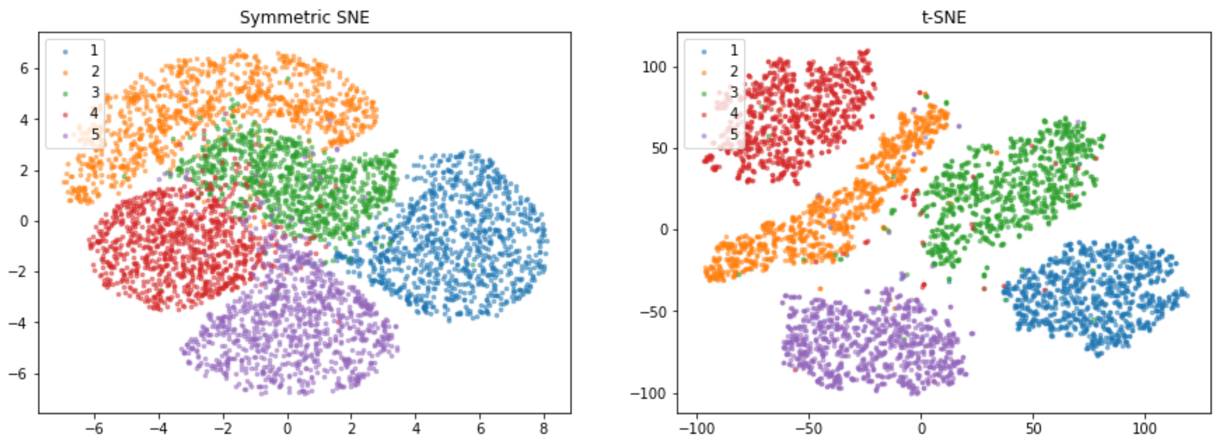
```python
def sne(X=np.array([]), no_dims=2, initial_dims=50, perplexity=30.0, q_func=q_joint, grad_func=grad_symmetric_sne):
    """
        Runs symmetric SNE or t-SNE on the dataset in the NxD array X to reduce its
        dimensionality to no_dims dimensions. The syntaxis of the function is
        `Y = tsne.tsne(X, no_dims, perplexity), where X is an NxD NumPy array.
    """
    ...
    for iter in range(max_iter):
        Q, dists = q_func(Y)
        Q = np.maximum(Q, 1e-12)
        # Compute gradient
        dY = grad_func(P, Q, Y, dists)
    ...
```

The following shows the iteration error of symmetric SNE.

```
Iteration 10: error is 26.346534
Iteration 20: error is 20.141809
Iteration 30: error is 17.839637
Iteration 40: error is 17.920184
Iteration 50: error is 17.982681
...
Iteration 980: error is 2.288007
Iteration 990: error is 2.288007
Iteration 1000: error is 2.288007
```

The following shows the iteration error of t-SNE.

```
Iteration 10: error is 25.862092
Iteration 20: error is 22.084489
Iteration 30: error is 19.595014
Iteration 40: error is 18.567858
Iteration 50: error is 18.268595
...
Iteration 980: error is 1.302603
Iteration 990: error is 1.301775
Iteration 1000: error is 1.300965
```



project 784-dimensional data to 2D space with symmetric SNE and t-SNE

Eigenfaces

First, we print the first 50 images from `att_faces` to see how they look like.

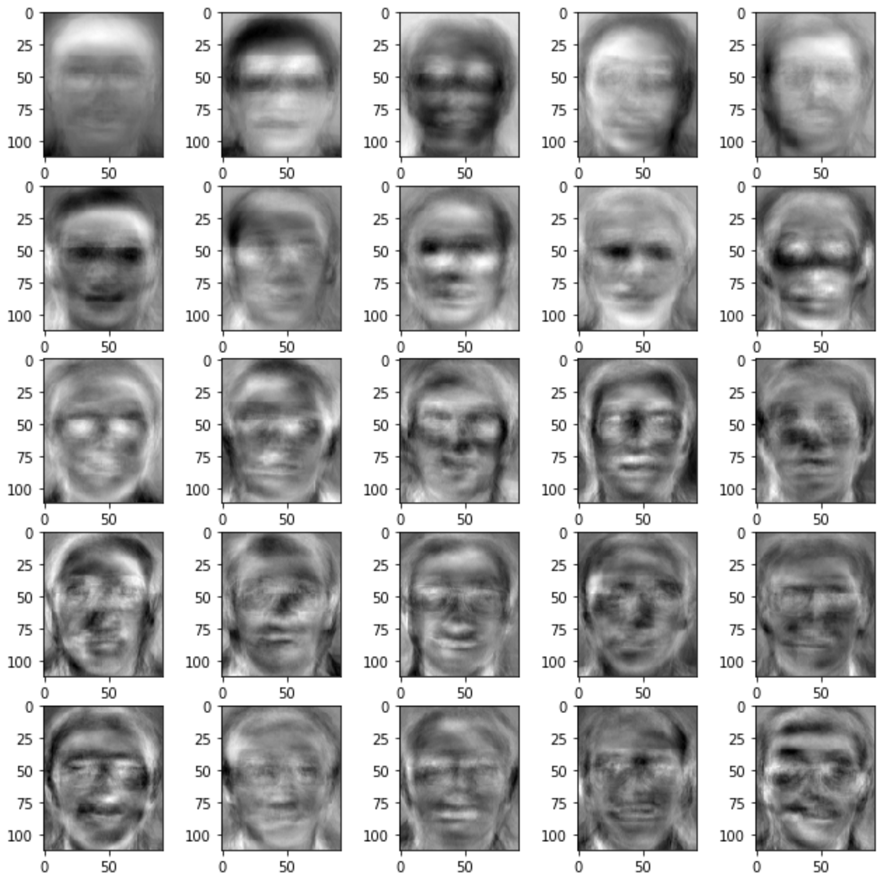
the first 50 images from `att_faces`

Use PCA to find the top 25 principal components. Reshape the eigenvectors, i.e. principal components, to get the eigenfaces.

```python
n_samples, h, w = images.shape
n_components = 25
centered_data, pc, mean = PCA(X, n_components)
eigenfaces = (pc.T).reshape((n_components, h, w))
plot_faces(eigenfaces.real, 5, 5)
```

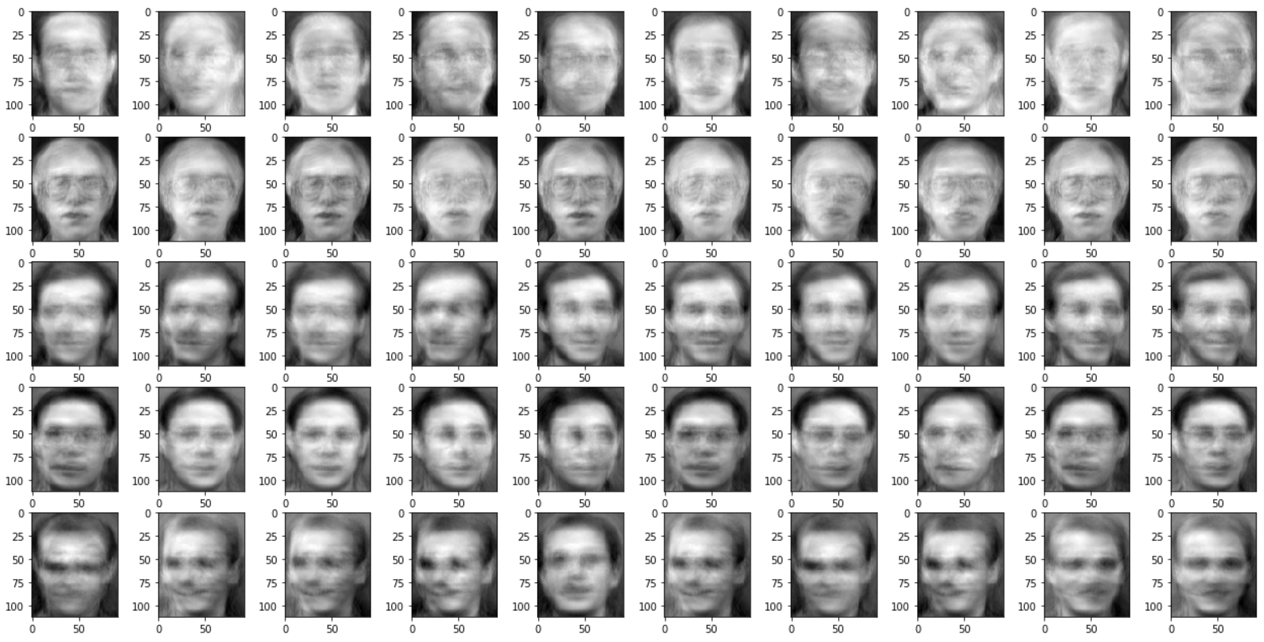
```python
def PCA(X, n_components):
    # standardize data to ~ N(0,1)
    N, dim = X.shape
    # x = (x - x.mean()) / x.std()
    mean = (np.mean(X, axis=0))
    x = X - mean

    # compute covariance matrix
    covariance_mat = np.cov(x.T)
    eig_vals, eig_vecs = LA.eig(covariance_mat)

    eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:,i]) for i in range(len(eig_vals))]
    eig_pairs = sorted(eig_pairs,key=lambda k: k[0], reverse=True)

    w = np.hstack((eig_pairs[0][1].reshape(dim,1), eig_pairs[1][1].reshape(dim,1)))
    for i in range(2,n_components):
        w = np.hstack((w, eig_pairs[i][1].reshape(dim,1)))

    return x, w, mean
```

```python
def plot_faces(images, n_row, n_col):
    plt.figure(figsize=(2.2 * n_col, 2.2 * n_row))
    for i in range(n_row * n_col):
        plt.subplot(n_row, n_col, i+1)
        plt.imshow(images[i], cmap=plt.cm.gray)
    plt.show()
```


eigenfaces

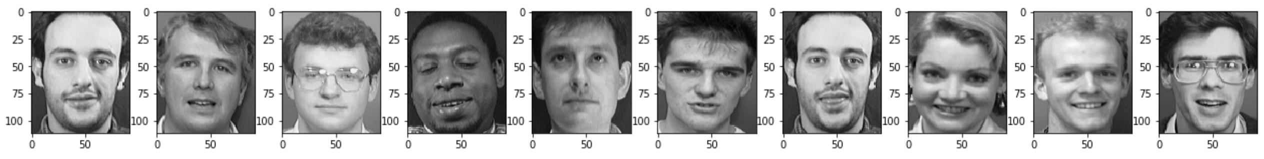
Reconstruct the faces using the eigenfaces. Each face is a weighted combination of the eigenfaces. We dot the centered data and eigenfaces to get the weights. Weight the eigenfaces will give us the centered face and add it with the mean face to get the final reconstructed faces.

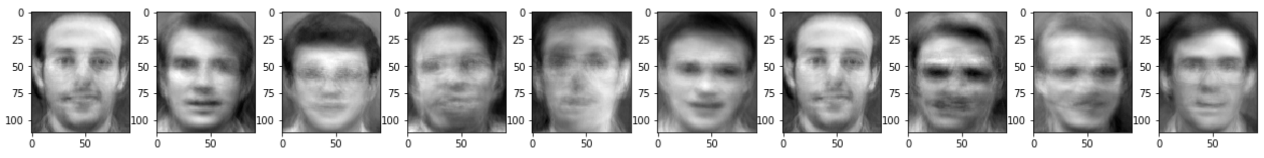
```python
def reconstruct_face(centered_data, pc, mean, h, w, img_idx):
    weights = np.dot(centered_data, pc.T)
    centered_vector = np.dot(weights[img_idx, :], pc)
    reconstructed_image = ( mean + centered_vector).reshape(h, w)

    return reconstructed_image
```


reconstructed faces the first 50 images

Then, we randomly pick 10 images and reconstruct them.

```
img_idx = np.random.randint(images.shape[0], size=10)
img_samples = np.array([plt.imread(img) for img in (np.array(image_names)[img_idx])], dtype=np.float64)
n, n_col = 1, 10
plot_faces(img_samples, n, n_col)      # show the original images
recovered_images = [reconstruct_face(centered_data, pc.T.real, mean, h, w, i) for i in img_idx]
plot_faces(recovered_images, n, n_col) # show the reconstructed results
```


randomly sampled 10 original images


reconstructed the above randomly sampled images

Reference

PCA

- 機器/統計學習:主成分分析(Principal Component Analysis, PCA)
- Implementing a Principal Component Analysis (PCA) – in Python, step by step

Eigenfaces

- Eigenfaces: Recovering Humans from Ghosts

LDA

- Linear Discriminant Analysis – Bit by Bit

SNE

- Symmetric SNE and t-SNE