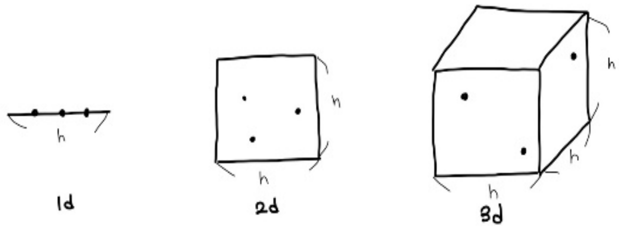
PCA 와 LDA 를 이용한 이미지 차원 축소

바이오인공지능융합학과 2021192861 김가연

1. Introduction

1.1 Curse of Dimensionality



Data 의 개수가 고정되어 있을 때, 차원이 늘어남에 따라 data 들 사이 간격은 늘어나게 되고 그 사이사이는 0 으로 채워지게 된다. 0 으로 채워진다는 것은 정보가 없다는 것으로 model 의 성능저하를 초래한다.

이것을 curse of dimensionality, 차원의 저주라고 한다. Curse of dimensionality 를 피하는 방법에는 여러가지가 있는데 그 중 차원을 축소하여 해결하는 방법이 있다. 본 실험에서는 현재 데이터의 특징을 조합하여 차원을 축소하는 방법인 'feature extraction' 중에서 PCA(Principal Components Analysis) 와 LDA(Linear Discriminant Analysis)를 이용해 실험을 하였다.

1.2 PCA(Principal Components Analysis)

$$\mathbf{y} = \mathbf{W}^T(\mathbf{x} - \mathbf{b})$$

PCA dimension reduction

PCA 는 차원 축소된 y 의 분산을 최대로 유지하는 것이 목적이다. 데이터(x)에 평균(b)를 빼서 평균을 원점으로 옮긴 후 weight(W)를 곱하여 차원 축소를 한다.

$$\max E[y_i^2] = E[(w_i^T(x-b))^2]$$
$$= E[w_i^T(x-b)(x-b)^T w_i]$$
$$= w_i^T E[(x-b)(x-b)^T] w_i$$
$$= w_i^T S w_i$$

즉 $\max w_i^T S w_i$ s.t. $\|w_i\|^2 = 1$

=> (lagrange multiplier)

$J: w_i^T S w_i - \lambda(\|w_i\|^2 - 1)$

=> $D_w J = 2Sw_i - 2\lambda w_i = 0$

$Sw_i = \lambda w_i$

w_i : scatter matrix의 eigenvector

λ : 해당 eigenvector에 해당하는 eigenvalue.

즉 PCA 를 통해 차원축소를 하려면 데이터의 scatter matrix 의 eigenvector 들로 W 를 구성하여 사용한다.

1.3 LDA (Linear Discriminant Analysis)

$$S_B = \sum_{i=1}^{C} N_i(\mu_i - \mu)(\mu_i - \mu)^T \qquad S_W = \sum_{i=1}^{C} S_i$$

$$\text{where } \mu = \frac{1}{N}\sum_{\forall x} x = \frac{1}{N}\sum_{x \in \omega_i} N_i \mu_i \qquad \text{where } S_i = \sum_{x \in \omega_i}(x - \mu_i)(x - \mu_i)^T \text{ and } \mu_i = \frac{1}{N_i}\sum_{x \in \omega_i} x$$

LDA 는 class 간 거리는 멀게(S_B), class 안의 분산은 작게(S_W), 유지시키는 것이 목적이다. S_B의 N_i 는 class 데이터 수에 따른 가중치인데 본 실험에서는 class 별 데이터의 개수가 모두 동일하므로 1 로 간주한다.

$$\max \frac{w^T S_B w}{w^T S_W w} \Rightarrow \max \; w^T S_B w \quad s.t \quad w^T S_W w = 1$$

$\Rightarrow$ (lagrange multiplier)

$$J: w^T S_B w - \lambda(w^T S_W w - 1)$$

$$\Rightarrow D_w J = S_B w - \lambda S_W w = 0$$

$$S_B w = \lambda S_W w$$

$$S_W^{-1} S_B w = \lambda w$$

w : $S_W^{-1}S_B$의 eigenvector

λ : 해당 eigenvector에 해당하는 eigenvalue

즉 LDA 를 통해 차원축소를 하려면 $S_W^{-1}S_B$의 eigenvector 들로 W 를 구성하여 사용한다.

본 실험에서는 각 이미지들을 차원축소 하여 eigenvector 들로 표현하였을 때 얼마나 잘 표현되는지에 대해 실험을 해 본다. PCA 는 차원축소 후 'eigen face'를 출력해 보고 reconstruct 후 'reconstructed image'도 비교해본다. LDA 는 차원축소 후 'fisher face'를 출력해 본다.

후에 두 방법 모두 다양한 dimension 으로 차원축소를 하여 정확도를 비교해보고, 10-Fold Cross Validation 방법을 통해 평균적으로 얼만큼의 정확도를 나타내는지 실험해 본다.

2. Method

2.1 Data

```
subject=[]
ori_subject=[]
for i in range(1,41):
  path = './att_faces/s'+str(i)
  file_list = os.listdir(path)
  line=[]
  line_1=[]
  for j in range(10):
    with Image.open(path+"/"+file_list[j]) as im:
      a = np.asarray(im)
      line.append(a)

      a_1=a.reshape(112*92,)
      line_1.append(a_1)
  ori_subject.append(line)
  subject.append(line_1)

subject=np.asarray(subject)
ori_subject = np.asarray(ori_subject)
```

```
subject.shape
```

```
(40, 10, 10304)
```

총 40 명의 사진이 10 장씩 400 장의 사진을 array 로 변환하여 (40, 10, 112*92[1]) shape 로 만들어 데이터(subject)로 사용한다.

[1] 이미지당 (112,92) pixel로 이루어져 있는데 이를 112*92 로 펼쳐서 사용

2.2 PCA function

2.2.1 def fit(data)

```
class PrincipalComponentAnalysis():


    def __init__(self):
        # eigen vector matrix
        self.eigen_mat = None
        # eigen value matrix (후에 크기별로 정렬할때 필요)
        self.eigen_val = None
        # n_components 만큼
        self.reduced_eigen_mat = None

    def fit(self, x):
        # data를 zero centered
        x = x - np.mean(x, axis=0, keepdims=True)

        n, d = x.shape
        cov = np.matmul(x.T, x)/n

        eigvals, eigvecs = np.linalg.eig(cov)
        eig_pairs = [(eigvals[i], eigvecs[:, i]) for i in range(d)]
        sorted_eig = sorted(eig_pairs, key=lambda tup: tup[0], reverse=True)

        self.eigen_mat = np.stack(list(map(lambda tup: tup[1], sorted_eig)), axis=1)
        self.eigen_val = np.array(list(map(lambda tup: tup[0], sorted_eig)))
```

Data 를 변수로 받아서 평균을 원점으로 옮기고 np.matmul 함수를 통해 covariance matrix 를 구한다. Covariance 의 eigvals(eigen values)와 eigvecs(eigen vectors)를 구하고 eigvals 가 큰 것부터 해당되는 eigvecs 를 정렬하여 eigen_mat 를 구성한다.

2.2.2 def transform(data, n_components)

```
def transform(self, x, n_components):
    x = x - np.mean(x, axis=0, keepdims=True)
    self.reduced_eigen_mat = self.eigen_mat[:, :n_components]
    results = np.matmul(x, self.reduced_eigen_mat)
    return results
```

Data 와 목표로 하는 차원 수(n_componets)를 변수로 받아 데이터의 평균을 원점으로 옮기고 eigen_mat 에서 n_components 만큼에 해당하는 열을 잘라 reduced_eigen_mat 로 칭하고 데이터와 reduced_eigen_mat 를 matmul 을 통해 차원 축소를 시킨다.

2.2.3 def reconstruct(data, x_transformed)

```
def reconstruct(self, data, x_transformed) :
    y = np.matmul(x_transformed, self.reduced_eigen_mat.T)+np.mean(data, axis=0, keepdims=True)
    y = y.reshape(y.shape[0],112,92)

    return y
```

차원 축소된 데이터(x_transformed)와 원래 데이터를 변수로 받아 x_transformed 와 reduced_eigen_mat 를 곱하고 원래 데이터의 평균을 더해준다. 후에 다시 112, 92 shape 으로 만들어준다.

2.3 LDA function

2.3.1 def fit(data)

```
class LinearDiscriminantAnalysis():

    def __init__(self):

        # eigen vector matrix
        self.eigen_mat = None
        # eigen value matrix (후에 크기별로 정렬할때 필요)
        self.eigen_val = None
        # n_components 만큼
        self.reduced_eigen_mat = None

        self.Sw=0
        self.Sw_inverse=None
        self.Sb=None
        self.cov=None
```

```
    def fit(self, x):
        # data를 zero centered

        # 클래스별 평균
        m=[]
        for i in range(len(x)):
            mean = np.mean(x[i], axis=0, keepdims=True)
            m.append(mean)
        m=np.asarray(m)

        # 클래스 전체 평균
        m_all = np.mean(m,axis=0)

        mean_set = m - m_all
        mean_set = mean_set.squeeze()

        #Sb
        self.Sb = np.matmul(mean_set.T, mean_set)


        # Sw
        si = (x-m)
        for i in range(len(si)):
            self.Sw += np.cov(si[i].T)/10


        self.Sw_inverse = np.linalg.pinv(self.Sw)

        x=x.reshape(x.shape[0]*x.shape[1],x.shape[2])
        n, d = x.shape
        self.cov = np.matmul(self.Sw_inverse, self.Sb)
        self.cov = (self.cov+self.cov.T)/2

        eigvals, eigvecs = np.linalg.eigh(self.cov)
        eig_pairs = [(eigvals[i], eigvecs[:, i]) for i in range(d)]
        sorted_eig = sorted(eig_pairs, key=lambda tup: tup[0], reverse=True)

        self.eigen_mat = np.stack(list(map(lambda tup: tup[1], sorted_eig)), axis=1)
        self.eigen_val = np.array(list(map(lambda tup: tup[0], sorted_eig)))
```

데이터를 변수로 받아 S_B와 S_W를 구한다. S_B는 각 class 별 평균에 전체 class 의 평균을 빼서 사용하고 S_W는 데이터에 class 별 평균을 뺀 S_i들의 합으로 이루어진다.

S_W가 singular 일 수도 있으니 pseudo inverse 를 통해 $S_W_inverse$를 구하고 $S_W_inverse$와 S_B를 matmul 하여 cov matrix 로 사용한다. 이때 cov matrix 의 eigen value 와 eigen vector 를 구하는데 복소수가 나오는 경우가 발생해 eigh 를 통해 eigvals 와 eigvecs 를 구하였다. np.linalg.eigh 함수를 이용하기 위해서 cov matrix 를 (cov + cov.T)/2 를 통해 임의로 symmetric 하게 만들어 주었다. 마지막으로 eigvals 가 큰 것부터 해당되는 eigvecs 를 정렬하여 eigen_mat 를 구성한다.

2.3.2 def transform(data, n_components)

```
def transform(self, x, n_components):

    self.reduced_eigen_mat = self.eigen_mat[:, :n_components]

    results = np.matmul(x, self.reduced_eigen_mat)

    return results
```

Data 와 목표로 하는 차원 수(n_componets)를 변수로 받아 데이터의 평균을 원점으로 옮기고 eigen_mat 에서 n_components 만큼에 해당하는 열을 잘라 reduced_eigen_mat 로 칭하고 데이터와 reduced_eigen_mat 를 matmul 을 통해 차원 축소를 시킨다.

2.4 공통된 function

2.4.1 def split_gallery_query(test_transformed)

```
def split_gallery_query(test_transformed):
    test_transformed = test_transformed.reshape(4,10,test_transformed.shape[2])
    print('test transformed shape : ', test_transformed.shape)
    gallery_trans=[]
    query_trans=[]
    for i in range(4):
        #gallery_trans.append(i+1)
        for j in range(10):
            if j < 7:
                gallery_trans.append(test_transformed[i][j])
            else :
                query_trans.append(test_transformed[i][j])

    gallery_trans=np.asarray(gallery_trans)
    query_trans=np.asarray(query_trans)
    #print('gallery : ₩n',gallery_trans,'₩n gallery shape : ₩n', gallery_trans.shape)
    #print('query : ₩n',query_trans,'₩n query shape : ₩n', query_trans.shape)
    return gallery_trans, query_trans
```

차원 축소된 데이터(test_transformed)를 변수로 받아 7:3 의 비율로 gallery, query 로 나누어 return 해준다.

2.4.2 def scatter(gallery, trans)

```
def scatter(gallery_trans, query_trans):
    plt.figure(figsize=(8, 8), dpi=120)

    fig = plt.figure()
    ax = fig.gca(projection='3d')

    x=gallery_trans[:,0].astype(np.float32)
    y=gallery_trans[:,1].astype(np.float32)
    z=gallery_trans[:,2].astype(np.float32)

    x1=query_trans[:,0].astype(np.float32)
    y1=query_trans[:,1].astype(np.float32)
    z1=query_trans[:,2].astype(np.float32)

    ax.scatter(x,y,z)
    ax.scatter(x1,y1,z1, c='red')

    plt.show()
```

Gallery 와 query 의 3 차원까지만 3-d scatter 로 뿌려 확인한다.

2.4.3 def knn_accuracy(gallery_trans, query_trans, knn_num)

```
def knn_accuracy(gallery_trans, query_trans, knn_num):
    gallery_trans = gallery_trans.reshape(1,28,gallery_trans.shape[1])
    query_trans = query_trans.reshape(12,1,query_trans.shape[1])

    # query gallery 거리 찾기
    distance = np.sum(np.square(query_trans-gallery_trans),axis=-1)
    #print(distance, distance.shape)

    # sorting하여 가까운 사진 index로 정렬
    sort_distance = np.argsort(distance, axis=1).reshape(4,3,28)
    #print(sort_distance, sort_distance.shape)

    # knn_num개 knn
    knn = sort_distance[:,:,:knn_num]//7
    #print(knn, knn.shape)
    knn = knn.reshape(12,knn_num)


    # count[0] = 1번 사람이라고 예측한 수
    count = []

    for row in knn:
        # knn행별로 사람 플러스 해줄 임시 배열
        temp = [0 for i in range(4)]
        for j in range(knn_num):
            temp[row[j]]+=1
        count.append(temp)
    count=np.asarray(count).reshape(4,3,4)
    #print(count, count.shape)

    # 정확도
    accuracy = (np.sum(count, axis=1)/(3*knn_num))*100

    accuracy = pd.DataFrame(accuracy, columns=['1','2','3','4'], index=['1','2','3','4'])


    return accuracy
```

gallery_trans 와 query_trans, knn 을 수행할 개수를 변수로 받아 정확도를 측정하는 함수이다.

Query 들과 gallery 데이터들 모두 거리를 측정하여 argsort 를 이용해 query 들이 gallery 몇 번째 사진과 가장 가까운지를 sort_distance 에 저장한다. 이때 gallery 에는 한 사람의 사진이 7 장씩 존재하므로 0~6 번은 1 번 사람, 7~13 번은 2 번 사람, … , 21~27 번은 4 번 사람에 해당되는 사진이다. 각 query 사진마다 몇 번 사진이 몇 번 나왔는지를 count 에 저장한다. 예를 들어 1 번 사람 query 사진이 1 번 사람 gallery 사진이 5 번, 3 번 사람 gallery 사진이 2 번 나왔다면 [5,0,2,0]이 count 에 저장된다.

후에 count 를 모두 더해 3*knn_num 으로 나누고 100 을 곱하여 정확도 matrix 를 구한다. 실제 정확도는 정확도 matrix 의 trace 를 모두 더해 4 로 나눈 값이 된다.

본 실험에서 한 사람 당 gallery 의 개수를 7 개로 정하였으므로 knn 은 최대 7 을 사용해야 의미 있는 결과를 도출해 낼 수 있다. knn_num 을 7 개로 설정을 하면 내 gallery 의 사진 7 개 모두가

들어왔는지 확인할 수 있기 때문이다. 그러므로 본 실험에서는 knn_num 을 7 으로 설정해 실험해 본다.

3. Result

3.1 다양한 dimension 으로 차원 축소했을 때 비교

Subject 에서 random 하게 4 명의 사진[2](총 40 장)은 test 로 나머지 36 명의 사진(총 360 장)은 train 으로 사용한다. 각 데이터의 shape 는 (n*10,10304)이다.

3.1.1 PCA

- eigen face

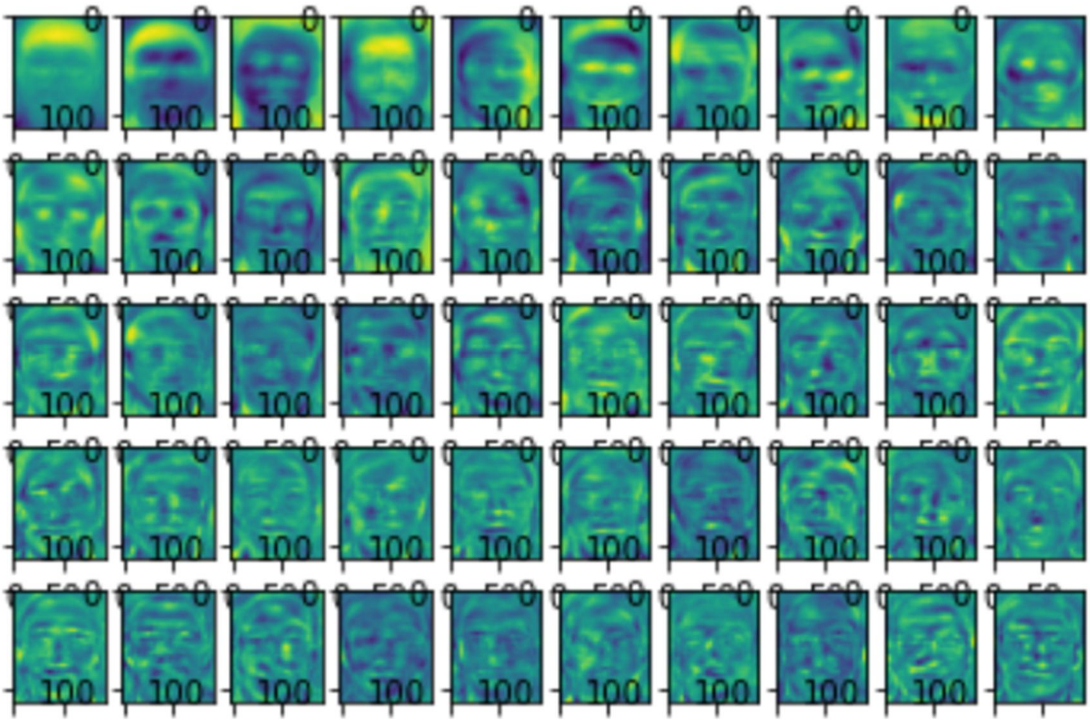
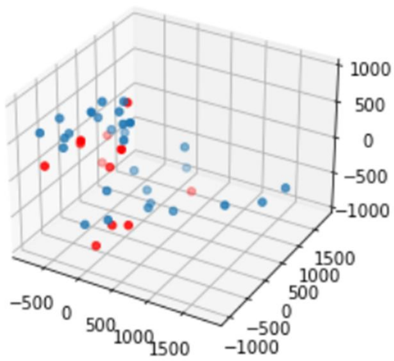


Figure 1 eigen face

[2] 차원에 따른 비교는 각 방법마다(PCA, LDA) 비교를 하여 test 데이터를 랜덤으로 추출하였다.

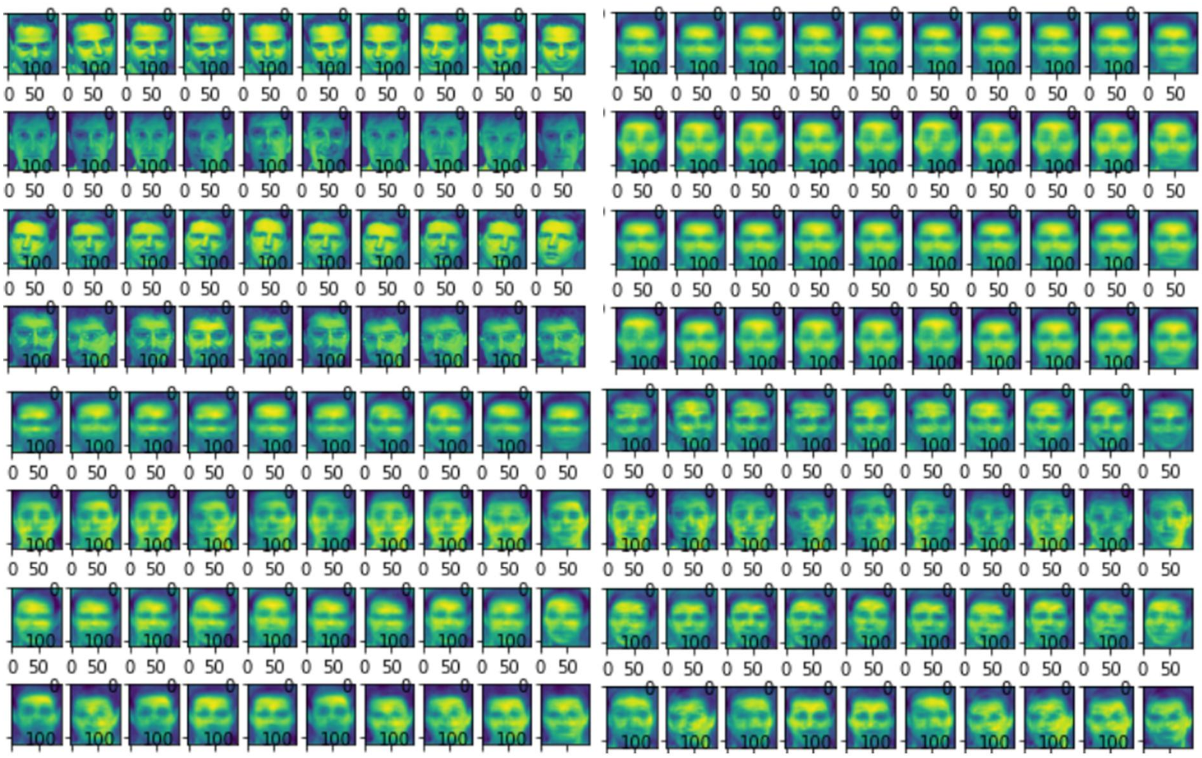
- scatter



scatter 함수를 통해 이미지들의 위치를 찍어 본다.

빨간 점이 query, 파란 점이 query이다.

- reconstruct 후 image



왼쪽 위부터 시계방향으로 각각 original image, n_components=3, n_components=50, n_components=10으로 차원축소 후 다시 reconstruct한 이미지이다. 축소한 차원이 클수록 원래 이미지와 좀 더 비슷하게 reconstruct 되는 것을 볼 수 있다.

- 차원 수에 따른 정확도

n_components = 3, 10, 50 에 따른 정확도 비교

```
print('n_components=3 일때 정확도 : ', accuracy_n_3)
```

```
n_components=3 일때 정확도 :  69.04761904761904
```

```
print('n_components=10 일때 정확도 : ', accuracy_n_10)
```

```
n_components=10 일때 정확도 :  70.23809523809524
```

```
print('n_components=50 일때 정확도 : ', accuracy_n_50)
```

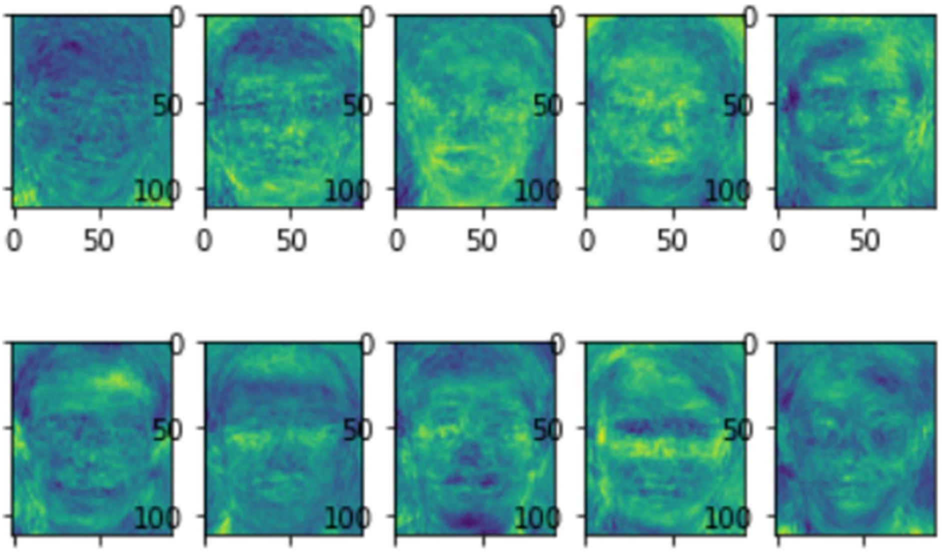
```
n_components=50 일때 정확도 :  77.38095238095238
```

위의 reconstruct image 결과와 비슷하게 n_components의 수가 높아질수록 정확도가 높아짐을 확일 할 수 있다.

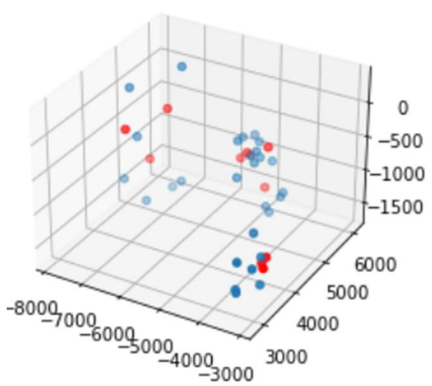
3.1.2 LDA

LDA는 PCA와 다르게 축소할 수 있는 최대 차원은 S_B의 최대 rank인 C-1까지이다. 본 실험에서는 36명을 train data로 사용하였으니 축소 가능한 최대 n_components는 35이다.

- fisher face



- scatter



scatter 함수를 통해 이미지들의 위치를 찍어 본다.

빨간 점이 query, 파란 점이 query이다.

- 차원 수에 따른 정확도 비교

n_components = 3, 10, 35 에 따른 정확도 비교

```
print('n_components = 3 일때 정확도 : ', accuracy_n_3)
```

```
n_components = 3 일때 정확도 :  82.14285714285714
```

```
print('n_components = 10 일때 정확도 : ', accuracy_n_10)
```

```
n_components = 10 일때 정확도 :  84.52380952380952
```

```
print('n_components = 35 일때 정확도 : ', accuracy_n_35)
```

```
n_components = 35 일때 정확도 :  86.9047619047619
```

PCA와 마찬가지로 차원의 수가 커질수록 정확도가 높아진다.

차원에 따른 비교는 각 방법별로 비교해 보았다. 먼저 test data로 사용한 4명의 사람을 random으로 뽑아 비교해 보았다. 원래의 차원은 10304 였지만 PCA, LDA 모두 20배가 넘는 차원으로 줄여도 높은 정확도를 나타냈다.

PCA에서는 3차원에서는 69.05%, 10차원에서는 70.34%, 50차원에서는 77.38%로 70%이상의 정확도를 볼 수 있었고, LDA에서는 3차원에서는 82.14%, 10차원에서는 84.53%, 최대 차원인 35차원에서는 86.90%로 82% 이상의 결과를 도출해 낸다. 특히 LDA에서는 최대 차원인 35차원으로 축소하였을 때 86.90%로 아주 높은 정확도를 나타냈다.

3.2 10-Fold Cross Validation 방법을 이용한 정확도 비교

원래의 데이터는 (n,10304) shape, 즉 10304 차원이다. 그중 1% 정도에 해당되는 10차원으로줄였으때 얼만큼의 정확도를 나타내는지 확인하고 싶어 n_components를 10으로 설정하고 10-Fold Cross Validation 방법을 이용한 실험 결과이다.

PCA, LDA 모두 40명의 이미지를 차례로 4명씩 잘라 test data로, 나머지 36명의 이미지는 train data로 설정하여 총 10번의 실험을 한 결과이다.

3.2.1 PCA

	1	2	3	4
1	88.571429	5.714286	1.428571	4.285714
2	1.428571	95.238095	1.428571	1.904762
3	8.571429	6.190476	75.238095	10.000000
4	2.380952	2.380952	11.904762	83.333333

PCA에서는 한사람당 query 사진 3장은 평균적으로 1번 사람은 88.57%, 2번 사람 사진은 95.24%, 3번 사람은 75.24%, 4번 사람은 83.33%의 정확도를 보여주었다. 평균적으로 총 **85.60%의 정확도**를 도출해냈다.

3.2.2 LDA

	1	2	3	4
1	95.714286	1.904762	0.952381	1.428571
2	0.476190	99.523810	0.000000	0.000000
3	2.857143	7.142857	83.333333	6.666667
4	1.904762	2.380952	7.142857	88.571429

LDA에서는 한사람당 query 사진 3장은 평균적으로 1번 사람은 95.71%, 2번 사람 사진은 99.52%, 3번 사람은 83.33%, 4번 사람은 88.57%의 정확도를 보여주었다. 평균적으로 총 **91.79%의 정확도**를 도출해냈다.

본 실험에서는 LDA의 성능이 더 좋게 나오는 것을 확인할 수 있다. 전체적으로 두 방법의 개념은 유사하지만 PCA는 class label 없이 데이터 셋 전체에 적용하는 'unsupervised' 알고리즘이고, LDA는 class label 정보를 이용하여 각 class를 최대한 분리해 내는 'supervised' 알고리즘이다. 이를 보면 LDA가 PCA보다 성능이 좋다는 것을 알 수 있다. 하지만 image 인식에서 class의 sample 개수가 적으면 PCA의 성능이 더 좋을 경우도 존재한다.[3] 따라서 데이터의 수가 더 적어졌을 때 본 실험을 수행하면 PCA의 성능이 더 높게 나올 수도 있을 것이다.

[3] Martinez et al., 2001, https://ieeexplore.ieee.org/document/908974?arnumber=908974