중앙대학교
CHUNG ANG UNIVERSITY

정보와 지식: 탐구 과제 2 보고서

(2023-1)

강의 / 분반:	정보와 지식 02분반
교수:	박재화
제출일자:	2023. 06. 24
학번:	20184757
이름:	주영석

목차

1. 서론

주성분 분석이란 (PCA – Principal Component Analysis), 고차원 데이터를 차원 축소하고 (Dimensionality Reduction) 특징 추출하는 (Feature Extraction) 수학적 기법으로, 데이터에서 가장 중요한 패턴과 특징을 잘 유지하도록 부분 공간에 사영하는 기법이다. 이를 통해 차원 축소를 하더라도 기존 데이터에서 중요한 정보를 잃지 않고, 데이터의 물리적인 정보량은 효율적으로 축소할 수 있다. 이러한 주성분 분석 기법은 데이터 시각화의 기본적인 방법이며, 데이터 사이언스뿐만 아니라 다양한 컴퓨터 비전, 인공지능 분야에 활용된다.

얼굴 인식이란 (Face Detection), 이미지에서 얼굴을 찾는 것, 사람의 얼굴인지 판별하는 것, 누구의 얼굴인지 확인하는 것 등을 목표로 하는 컴퓨터 비전의 하나의 분야이다. 얼굴에는 각 기관의 위치뿐만 아니라 감정 등의 고차원적인 정보도 표현이 되므로, 감정 분류와 같은 문제도 존재한다. 얼굴 인식은 컴퓨터 비전의 연구 초기부터 연구되어 오던 주제이며, 최근에는 딥러닝의 발전에 힘입어 얼굴 인식 알고리즘의 성능이 크게 발전했다. 그러나, 딥러닝 이전 얼굴 인식에서 가장 중요한 알고리즘으로는 Eigenface가 있다. Eigenface는 주성분 분석을 이용해 얼굴에서 중요한 특징을 추출하고 저차원의 데이터로 표현하여 효율적으로 얼굴인식을 할 수 있는 알고리즘이다.

이 글에서는 Kaggle의 Olivetti face dataset을 (https://www.kaggle.com/datasets/imrandude/olivetti) 사용하여 eigenface를 이용한 face detection을 수행하고자 한다. 해당 데이터는 400개의 얼굴 이미지 데이터를 가지며, 한 명당 10장의 흑백으로 저장된 얼굴 이미지가 40명에 대해 존재한다. 하나의 이미지는 가로 64, 세로 64개의 픽셀을 가지므로, 총 4,096개의 픽셀 데이터를 포함한다. 이 데이터에 대해 eigenface 시스템을 구축하고, projection subspace의 dimension에 따른 face detection의 정확도를 비교하고자 한다. 또한, projection subspace의 dimension에 따른 eigenface의 차이를 살펴보고자 한다.

2. 본론 및 구현

2-1. Data Preprocessing

Train과 test 과정에 쓰일 데이터를 나누고자 총 400개의 데이터를 8:2로 나누어, 320개와 80개로 나누었다. 이때, 치우침이 없도록 한 명당 10장의 얼굴 이미지를 8개와 2개로 나누었다. 그 다음, 하나의 이미지가 64*64픽셀 크기를 가지고 있으므로, 4096 길이의 벡터로 만든 뒤, 데이터셋을 N*4096 크기의 행렬 형태로 저장했다.

위의 그림은 데이터셋에서 10명에 대해 10장의 얼굴 이미지를 시각화한 것이다. 이 중 하나의 열에 대해서 왼쪽부터 8개를 train 데이터셋으로, 나머지 2개를 test 데이터셋으로 삼았다.

Kaggle에서 제공하는 원본 파일은 npy 형태로 제공되며, 이를 읽고 파일을 데이터를 가공 및 csv 파일로 저장하기 위해 numpy 패키지를 사용했다. 또한, 데이터 시각화를 위해 matplotlib.pyplot 패키지를 사용했다.

2-2. PCA

우선, PCA를 하기 위해 주어진 데이터의 공분산 행렬을 (covariance matrix) 구한다. 공분산 행렬을 구하기 위해,

우선 데이터의 평균 벡터를 구한다. 즉, N개의 D 차원 데이터에 대해서 N에 대해 D 차원 평균 벡터를 구하는 것이다. 다음으로 모든 데이터 벡터에 구한 평균 벡터를 뺀다. 이렇게 구해진 행렬을 제곱하는데, 행렬과 전치행렬의 행렬 곱셈을 함으로써, (D*N)*(N*D)=(D*D) 행렬을 얻는다. 마지막으로 이 D*D 행렬에 상수 N을 나눈다. 코드는 아래와 같다.

```
# calculate covariance matrix of input matrix X with shape (D, N) that has D dimension and N samples
def calculate_covariance_matrix(X):
    # calculate mean of data
    mean = np.mean(X, axis=1)
    # calculate centered data matrix
    X_centered = X - mean[:, None]
    # compute covariance matrix with centered data matrix
    N = X.shape[1]
    covariance_matrix = np.dot(X_centered, X_centered.T) / N
    return covariance_matrix
```

다음으로, 구한 공분산 행렬의 eigenvector와 eigenvalue를 구한다. Eigenvector를 구하는 알고리즘은 power iteration, Lanczos algorithm, QR algorithm 등 여러 가지가 있으나 각각의 시간 복잡도, 안정성, 정확성, 효용성이 다르고 요구되는 수학적 배경지식이 많으므로, numpy 라이브러리의 np.linalg.eig를 이용해서 간단하게 구했다. 해당 함수를 이용하면 eigenvalue와 normalized eigenvector를 얻을 수 있다. 코드는 아래와 같다.

```
# calculate eigenvector and eigenvalue of given matrix
def calculate_eigenvalue_eigenvector(X):
    eig_val, eig_vec = np.linalg.eig(X)
    return eig_val, eig_vec
```

마지막으로, 얻은 eigenvalue를 내림차순으로 정렬한 뒤, 가장 큰 eigenvalue를 가지는 k개의 eigenvector를 선택한다. 이 eigenvalue로 이루어진 k*D 행렬이 곧 transformation (projection) matrix이며, projection subspace의 unit vector이며, 주어진 데이터셋의 k개의 principal component이다. 또한, eigenface system에서는 이를 eigenface로 부른다. 해당 projection matrix를 통해 주어진 데이터와 행렬 곱셈 연산을 하여 projected data를 계산한다. 이때, eigenface에서는 주어진 데이터를 그대로 projection 하는 것이 아니라, 데이터와 평균 데이터의 차를 projection 한다. 코드는 아래와 같다.

```python
# compute principal component analysis of given matrix X with
# shape (D, N) with D dimension and N samples to k principal components
def pca(X, k):
    # calculate covariance matrix of X
    print('calculating covariance matrix')
    covariance_matrix = calculate_covariance_matrix(X)
    # calculate eigenvalue and eigenvector of covariance matrix
    print('calculating eigenvalue and eigenvector')
    eigenvalue, eigenvector = calculate_eigenvalue_eigenvector(covariance_matrix)
    # sort eigenvalue and eigenvector in descending order
    print('sorting eigenvalue and eigenvector')
    eigenvalue_sorted_idx = np.argsort(eigenvalue)[::-1]
    # eigenvalue_sorted = eigenvalue[eigenvalue_sorted_idx]
    eigenvector_sorted = eigenvector[:, eigenvalue_sorted_idx]
    # calculate projection matrix
    projection_matrix = eigenvector_sorted[:, :k].T
    # calculate projected data matrix
    print('calculating projection')
    mean = np.mean(X, axis=1)
    projected_data = np.dot(projection_matrix, X-mean[:, None])
    # return projected data matrix and projection matrix
    return projected_data, projection_matrix
```

2-3. Training

Training 과정은 앞서 정의해 둔 PCA 함수를 호출함으로써 이루어진다. 축소할 차원 k를 정해서 PCA 함수에 인자로 전달한다. PCA 함수에서 계산된 projected data와 projection matrix를 저장한다.

우선, training data, k를 입력받고, training 데이터셋 파일을 불러온다.

```python
# get input parameter k from command line, default 100
k = 100
if len(sys.argv) > 1:
    k = int(sys.argv[1])
print(f'k = {k}')

# get train data from command line
train_data_path = 'train_data.csv'
if len(sys.argv) > 2:
    train_data_path = sys.argv[1]

# load training data
print('loading training data')
train_data = np.loadtxt(train_data_path, delimiter=',')
```

다음으로, PCA 함수를 호출하여 projected data와 projection matrix를 구한다.

```python
# compute eigenface of train data with projection dimension k
projected_data, projection_matrix = pca(train_data.T, k)
```

마지막으로, 구한 projected data와 projection matrix를 csv 파일로 저장하고, eigenvalue가 가장 높은 eigenface와 가장 낮은 eigenface를 시각화한다.

```python
# save projected data and projection matrix to csv file
np.savetxt(f'projected_data_{k}.csv', projected_data, delimiter=',')
np.savetxt(f'projection_matrix_{k}.csv', projection_matrix, delimiter=',')
print('saved projected_data.csv and projection_matrix.csv')
```

```
if k>=8:
    # plot first 4, last 4 eigenface
    fig, axes = plt.subplots(2, 4, figsize=(8, 5))
    for i in range(4):
        axes[0][i].imshow(projection_matrix[i,:].reshape(64, 64).astype(np.float64), cmap='gray')
        axes[1][i].imshow(projection_matrix[-i-1,:].reshape(64, 64).astype(np.float64), cmap='gray')
        axes[0, i].axis('off')
        axes[1, i].axis('off')
    plt.suptitle(f'First 4 and Last 4 Eigenfaces of k={k}')
    plt.show()
else:
    # plot all eigenface
    fig, axes = plt.subplots(2, 4, figsize=(8, 4))
    for i in range(k):
        axes[i//4][i%4].imshow(projection_matrix[i,:].reshape(64, 64).astype(np.float64), cmap='gray')
    for i in range(4):
        axes[0, i].axis('off')
        axes[1, i].axis('off')
    plt.suptitle(f'First {k} Eigenfaces')
    plt.show()
```

2-4. Testing

Testing 단계는 다음과 같다. 우선, 테스트 데이터셋을 projection 할 projection 함수, projected test data와 projected train data 간의 유클리드 거리를 계산할 함수, 계산된 거리를 토대로 test data가 누구인지, 해당 테스트 데이터의 레이블을 결정할 함수, 마지막으로 정확도를 계산할 함수이다.

첫째로, projection 함수는, 데이터 행렬과 projection matrix가 주어졌을 때, 단순하게 행렬 곱셈을 계산하는 함수이다. 이는 numpy 패키지의 dot 함수로 계산이 가능하다. 코드는 아래와 같다.

```
# calculate projection of data X with projection matrix
def projection(X, projection_matrix):
    return np.dot(projection_matrix, X)
```

다음으로, 주어진 두 벡터 간의 유클리드 거리를 측정할 함수로, 마찬가지로 numpy 함수의 linalg.norm 함수로 계산이 가능하다. 코드는 아래와 같다.

```
# calculate euclidean distance between two vectors
def euclidean_distance(x, y):
    return np.linalg.norm(x - y)
```

가장 중요한 test data의 레이블을 결정하는 함수이다. Projection 된 test data 하나당, 모든 projection 된 train data 벡터와의 거리를 계산하고, 가장 거리가 작은 train data의 레이블을 prediction label로 삼는다. 이 과정을 수행하기 위해 함수는 projected test data, projected train data를 인자로 받아야 하며, train data의 label 또한 받아야 한다. 코드는 아래와 같다.

```
# calculate prediction label of test data
def predict_label(test_data, train_data, train_label):
    # initialize prediction label
    test_prediction_label = np.zeros(test_data.shape[1], dtype=np.int64)
    for i in range(test_data.shape[1]):
        # calculate euclidean distances between test data and train data
        distance = np.array([euclidean_distance(test_data[:,i], train_data[:,j]) for j in range(train_data.shape[1])])
        # get prediction label by finding the index of minimum distance
        test_prediction_label[i] = train_label[np.argmin(distance)]
    return test_prediction_label
```

마지막으로, test data를 prediction 한 결과의 정확도를 계산할 함수이다. 정확도는 *(올바른 prediction의 수) / (전체 test data 수)*로 계산한다. 이를 위해 prediction label과 test data의 ground truth label을 인자로 받는다. 코드는 아래와 같다.

```
# calculate accuracy of prediction
def calculate_accuracy(prediction_label, ground_truth_label):
    return np.sum(prediction_label == ground_truth_label) / ground_truth_label.shape[0]
```

위의 함수들을 이용하여, 다음과 같이 코드를 구성한다. 우선, 프로그램 실행 시 test data, train label, test label을 입력받고, 해당 파일들을 불러온다.

```
# get test data from command line
test_data_path = 'test_data.csv'
if len(sys.argv) > 3:
    test_data_path = sys.argv[3]

# get train target from command line
train_target_path = 'train_target.csv'
if len(sys.argv) > 4:
    train_target_path = sys.argv[4]

# get test target from command line
test_target_path = 'test_target.csv'
if len(sys.argv) > 5:
    test_target_path = sys.argv[5]

# load test data
print('loading test data')
test_data = np.loadtxt(test_data_path, delimiter=',')

# load ground truth label of train and test data
print('loading train and test label')
train_label = np.loadtxt(train_target_path, delimiter=',', dtype=np.int64)
test_label = np.loadtxt(test_target_path, delimiter=',', dtype=np.int64)
```

Test data에서도 마찬가지로 projection을 하기 전 train 데이터셋의 평균 얼굴 데이터와의 차를 projection 해야만 한다. 따라서, 평균 얼굴을 구하고, test 데이터셋과의 차를 구한다. 앞서 정의한 projection 함수를 이용해 구한 test 데이터셋 행렬을 projection 시킨다. Projection matrix는 training 단계에서 저장한 것을 그대로 사용한다. 코드는 아래와 같다.

```
# calculate mean face
mean_face = np.mean(train_data, axis=0)

# subtract mean face from test data
test_data_centered = test_data.T - mean_face[:, None]

# project test data to k eigenface subspace
projected_test_data = projection(test_data_centered, projection_matrix)
```

다음으로, projection 된 test 데이터셋에 대해 label 예측을 계산한다. 또한, 예측된 test 데이터셋의 레이블에 대해 정확도를 구한다. 코드는 아래와 같다.

```
# predict label of test data
prediction_label = predict_label(projected_test_data, projected_data, train_label)

# calculate accuracy of prediction
accuracy = calculate_accuracy(prediction_label, test_label)
print(accuracy)
```

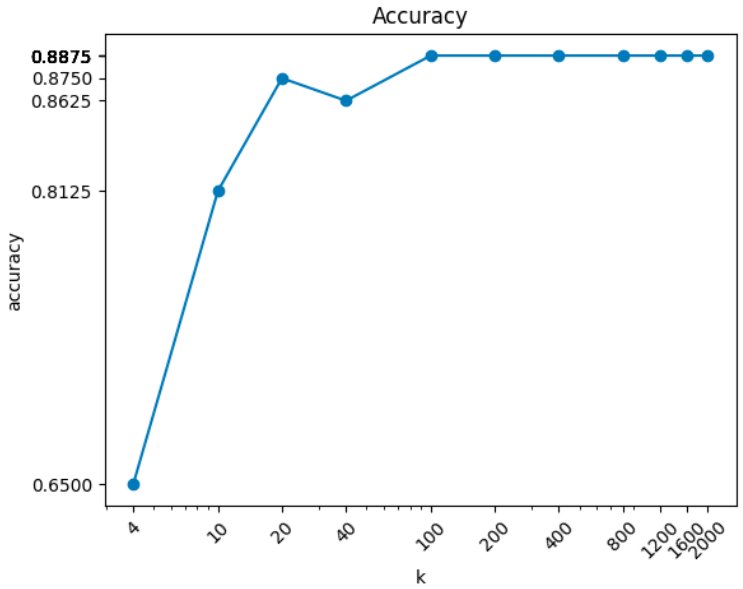
추가로, test data에 대해 예측된 레이블을 csv 파일로 저장한다. 코드는 아래와 같다.

```
# save prediction label to csv file
np.savetxt('prediction_label.csv', prediction_label, delimiter=',', fmt='%d')
```

3. 결과 및 분석

Projection dimension인 k 값을 4, 10, 20, 40, 100, 200, 400, 800, 1200, 1600, 2000으로 늘려가며 정확도와 eigenvalue가 가장 높은 eigenface 4개와 가장 낮은 eigenface 4개를 비교하였다.

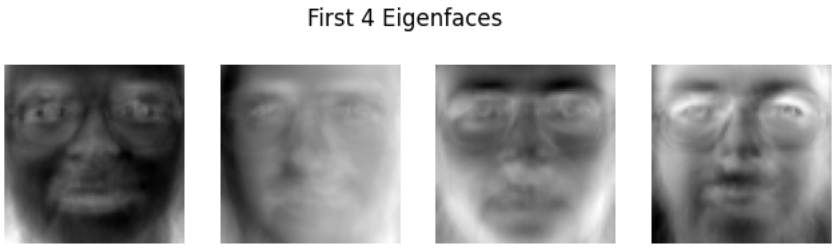
우선, k의 증가에 따른 accuracy의 그래프는 다음과 같다. 아래 그래프의 가로축은 log scale로 표현되었다.



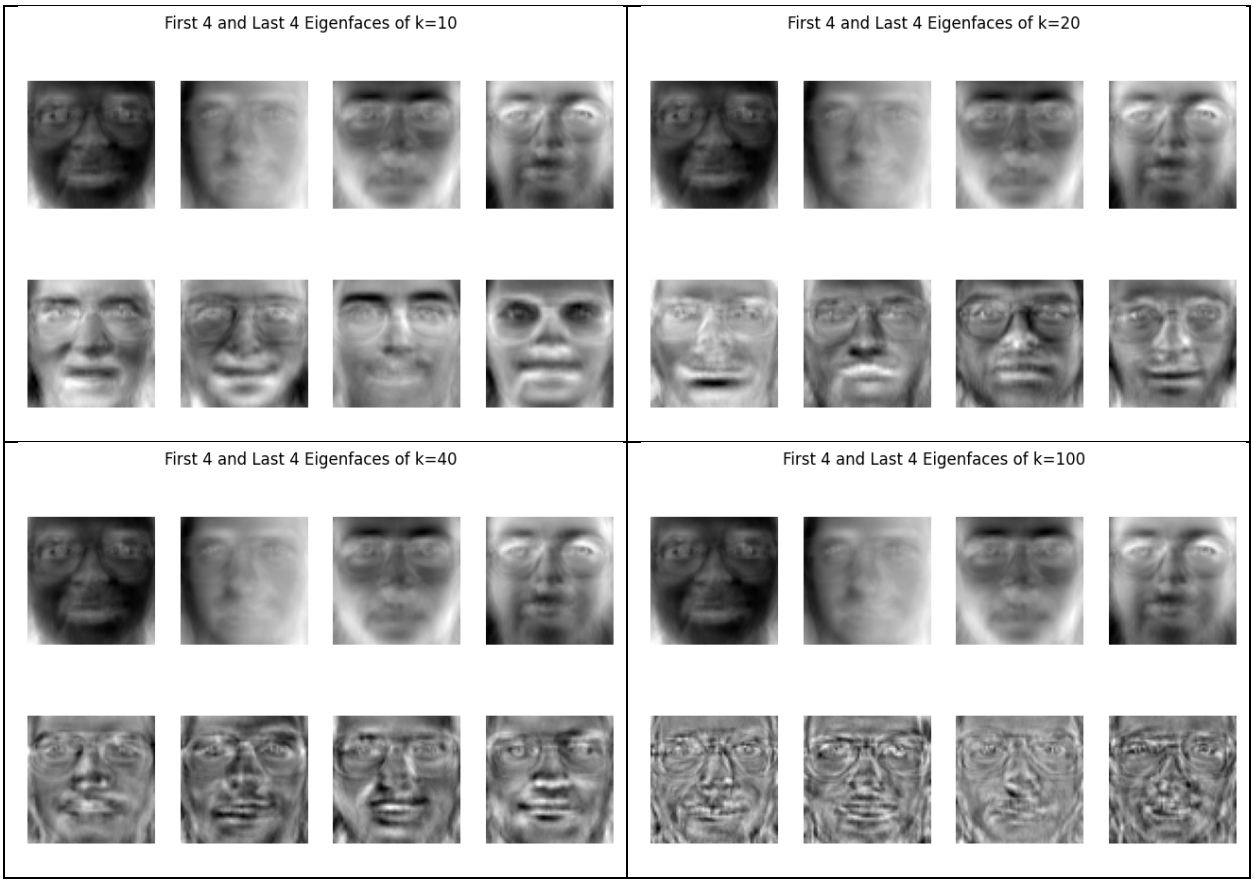
k=4일 때, 정확도가 0.65, k=10일 때 0.8125, 20일 때 0.8750으로 증가하는 양상을 보여준다. 40일 때 이전에 비해 소폭 감소한 0.8625의 정확도를 보여주지만, K=100 이후부터는 0.8875로 수렴하는 양상을 보여준다. k=100일 때 원본 이미지 데이터의 차원 64*64=4096에 비해 0.025배의 물리적인 정보량만을 담고 있지만,

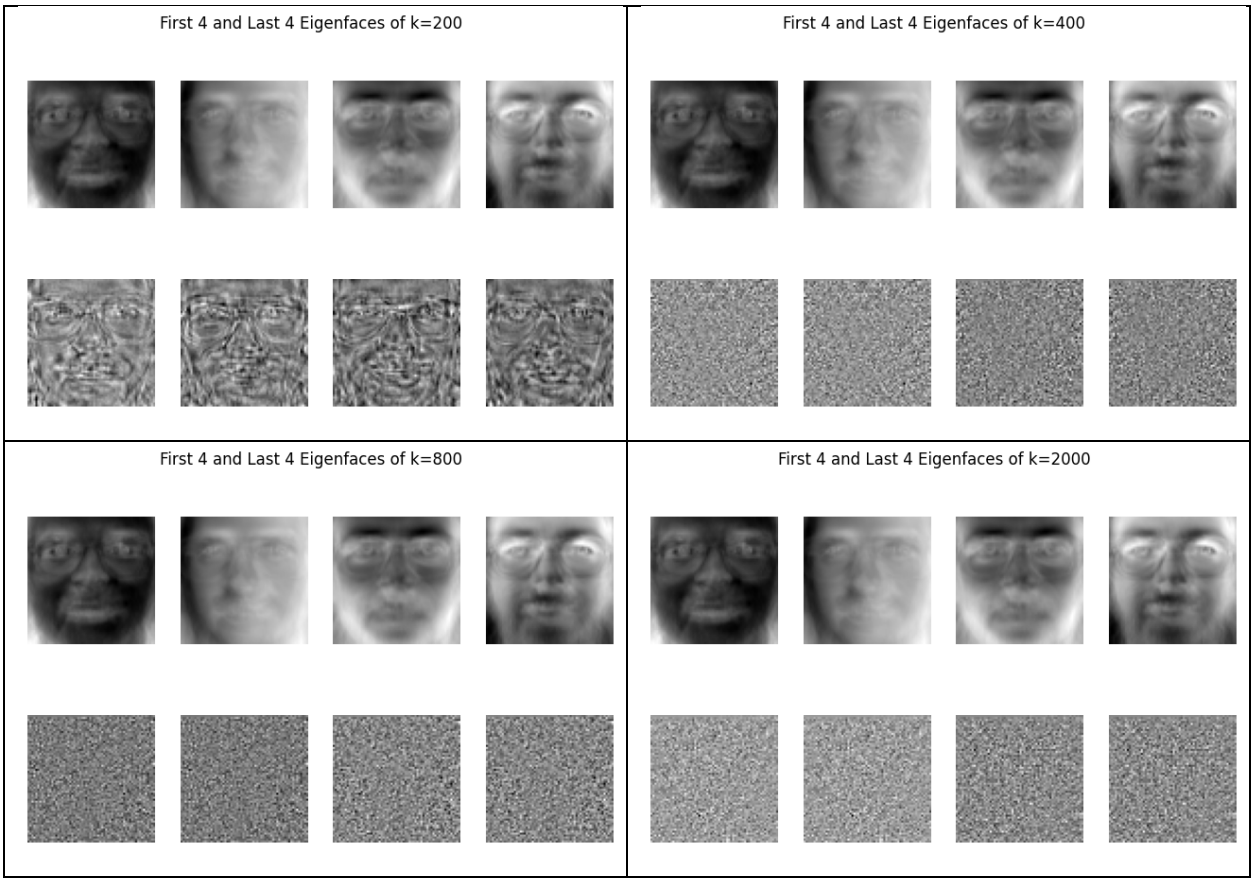
주성분 분석을 통해 얼굴 인식에 중요한 정보만을 가지기 때문에 얼굴 인식이 가능하다는 사실을 알 수 있다. 그러나, 데이터셋의 인물이 40명으로 상당히 적기 때문에 높은 정확도에 빠르게 수렴했을 수도 있다는 추측을 할 수 있다.

다음으로, k 값에 따른 eigenface를 살펴보고자 한다. 우선 k=4일 때 4개의 eigenface는 다음과 같다.



전반적으로 사람 얼굴의 형태를 띰을 확인할 수 있다. 이후의 k 값이 커지더라도, eigenvalue가 가장 큰 4개의 eigenface는 k=4일 때의 eigenface와 동일하다. 아래는 k 값의 변화에 따른 eigenvalue가 가장 큰 eigenface 4개와 eigenvalue가 가장 작은 eigenface 4개를 시각화한 것이다.





k 값이 작을 때의 eigenface는 전반적인 얼굴의 형태를 띠고 있으나, k 값이 커질수록, eigenvalue가 가장 작은 eigenface 4개의 형태는 얼굴의 형태를 잃어버리고, 사람이 인식했을 때 noise처럼 보인다. 이는 즉, 얼굴 데이터의 eigenvalue가 큰 주성분은 얼굴의 공통된 요소, 얼굴의 전반적인 형태를 나타내지만, eigenvalue가 작은 주성분은 세부적인 차이 정보, 그리고 noise 형태의 정보를 나타냄을 알 수 있다.

4. 한계점

첫째로, 속도가 느리다. PCA를 하는 과정에서 eigenvalue, eigenvector를 구하는 알고리즘이 있는데, 앞서 소개한 바와 같이 알고리즘의 종류에 따라 시간 복잡도, 안정성, 정확성 등이 차이가 난다. 현재 프로그램에 사용된 np.linalg.eig 함수는 약 O(N^3), 즉 세제곱의 시간복잡도를 보여준다. 이는 시간 측면에서 매우 비효율적임을 알 수 있다. 또한, 검사하고자 하는 사람의 수를 늘리려고 하면, 데이터를 추가하고 PCA 알고리즘을 다시 수행해야 하므로, 데이터를 추가함에 있어서도 효율적이지 않음을 알 수 있다. 그리고 해당 알고리즘의 경우 축소하고자 하는 차원의 수 k와 무관하게 동작하므로, 차원 축소를 매우 낮은 차원에 대해 하고 싶어도 PCA 알고리즘의 소요 시간은 동일하다.

두 번째로, 훈련 데이터의 양과 질에 의존적이다. 불충분한 데이터셋이나, 좋지 않은 데이터, 예를 들면 노이즈가 섞인 이미지 혹은 얼굴을 제대로 나타내지 않는 데이터에 대해 매우 민감하며, 이러한 상황에 성능은 크게 저하될 수 있다. 이 문제는 data-driven 모델에서는 모두 나타나는 공통적인 문제이기도 하다.

세 번째로, 성능이 좋지 않다. 3장의 결과에서 보여주듯, 훈련 데이터 320개, 사람 40명에 대해서도 정확도가 0.8875에서 수렴함을 보여준다. 이는, 최신 얼굴 인식 알고리즘에 비하면 매우 떨어지는 수치이며, 훈련 데이터의 규모에 비해서도 그다지 좋지 못한 정확도이다. 이는 PCA를 통해 얼굴 데이터의 주요 성분을 분석하는 방법 자체의 한계임을 알 수 있다.

5. 프로그램 실행 방법

프로그램 train_test.py는 Python3 언어로 작성되었다. 필요한 라이브러리는 numpy, matplotlib 이다. Numpy 라이브러리는 파일 읽기 및 쓰기, 벡터 및 행렬 연산, eigenvalue 및 eigenvector 연산을 위해 사용되었으며, matplotlib의 경우 데이터 시각화를 위해서 사용되었다. 해당 라이브러리를 설치하기 위해서는 python 패키지 관리 툴인 pip이 설치되어 있어야 한다. 터미널에서 다음의 명령어를 실행하면 모든 라이브러리를 설치할 수 있다.

$ pip3 install numpy matplotlib

다음으로, training.py를 실행하기 위해서는, 다음의 명령어를 실행하면 된다.

$ python3 train_test.py [projection dimension k] [train file path] [test file path] [train label path] [test label path]

첫 번째 인자인 [projection dimension]은 데이터를 projection할 차원의 수, 프로그램에서 k값이다. 기본값은 k=100으로 설정되어 있다. 두 번째 인자인 [train file path]는 training 과정에 사용될 파일의 경로이다. 기본경로는 2-1 장에서 전처리를 한 Olivetti face dataset의 training 데이터셋이다. 세 번째 인자인 [test file path]는 testing 과정에 사용될 파일의 경로이다. 기본경로는 2-1 장에서 전처리를 한 Olivetti face dataset의 testing 데이터셋이다. 네 번째와 다섯 번째 인자인 [train label path], [test label path]는 train과 test 데이터셋의 레이블 파일의 경로이다. 두 경로 모두 기본적으로 Olivetti face dataset의 train, test 레이블 파일의 경로로 설정이 되어있다. 모든 파일 입력은 csv 파일 형태만 지원한다.

projection dimension k값만 입력하고 기본으로 지정된 파일을 실행하고자 한다면, 다음의 명령어만 실행해도 된다.

$ python3 train_test.py [k]

혹은, 기본값인 k=100으로 프로그램을 실행하고자 한다면, 다음의 명령어만 실행해도 된다.

$ python3 train_test.py

프로그램을 실행하면, 우선 training 단계가 실행이 된다. 이 단계에서는 training 데이터에 대해 PCA를 수행하고, k에 따른 train 데이터셋의 projected matrix와 projection matrix가 계산된다. 동일한 경로에 k에 따라 서로 다른 projected_data_[k].csv, projection_matrix_[k].csv 파일들이 저장된다. 또한, k가 8보다 작거나 같을 경우 모든 eigenface가 시각화되어 출력되고, k가 8보다 클 경우 eigenvalue가 가장 높은 eigenface 4개와 가장 낮은 eigenface 4개가 출력된다.

다음으로, testing 단계가 실행이 된다. 이 단계에서는 testing 데이터를 앞서 계산한 projection matrix에 대해 projection을 계산하고, projected된 train 데이터셋과의 비교를 통해 test 데이터셋의 레이블을 예측한다. 마지막으로 Test 데이터셋의 실제 레이블에 따라 정확도를 계산하고 출력한다. 또한, 예측된 레이블을 동일한 경로에prediction_label_[k].csv 파일로 저장한다.

프로그램 실행 예시는 아래의 사진과 같다.

```
robinjoo1015@YoungSeok-MacBook-Air proj2 % python3 train_test.py
k = 100
==================== TRAINING PHASE ====================
loading training data
calculating covariance matrix
calculating eigenvalue and eigenvector
sorting eigenvalue and eigenvector
calculating projection
saved projected_data_100.csv and projection_matrix_100.csv
==================== TESTING PHASE ====================
loading test data
loading train and test label
calculating projection
predicting label
calculating accuracy
Accuracy: 0.8875
saved prediction_label_100.csv
robinjoo1015@YoungSeok-MacBook-Air proj2 %
```

해당 프로그램은 Python 3.11.4, macOS 13.4 환경에서 개발 및 실행되었다. Python3와 필요한 라이브러리만 설치 되어 있다면 모든 OS 환경에서 실행할 수 있다.