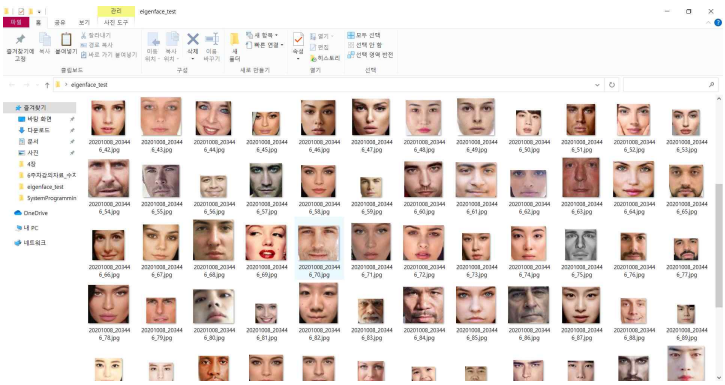
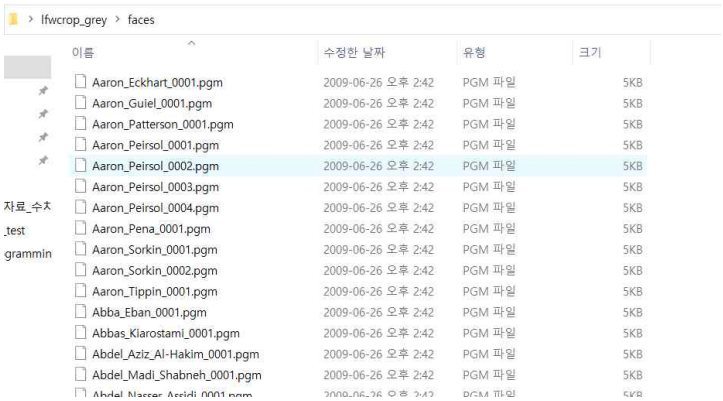
1. Collect face images

- Crop the same image size for face area
- Image size: 32x32
- At least 1000 gray images



[그림 1] '알캡처'를 이용한 face dataset

: 처음에는 나만의 자료를 위해 이미지를 무작정 모으기 시작했습니다. '알캡처'라는 툴을 사용하여 원하는 범위를 직접지정하여 이미지를 자르기 시작했습니다[그림1]. 검색어는 '얼굴모음, 얼굴사진모음, 얼굴정면사진모음, set of people face, front face, 모델남자정면' 등 여러 가지 검색어를 사용하였습니다. 또한, 얼굴의 범위를 눈썹위부터 입술아래까지 지정하여 crop하였습니다. 그렇게 100여장을 crop하고 이를 테스트용 이미지로 정하고 'eigenface dataset'으로 서칭을하여 pgm(grey image data set)파일을 아이겐페이스 데이터셋으로 사용하기로 하였습니다. 각각의 이미지는 [그림1]과 달리 눈썹부터 턱아래까지 얼굴 전체를 가지는 이미지셋입니다.



[그림 2] grey image dataset

2. Construct data matrix and covariance matrix using the face images

1. Calculate the mean vectors of your collected face images

1.에서 구한 얼굴이미지는 모두 각 픽셀의 값이 0부터 255까지 이루어진 vector element입니다. 원점을 포함하기 위해서는 collect한 face의 mean을 구한 후 각각의 vector에서 mean을 뺀 후 ±값을 갖는 32X32(1024)차원의 벡터들로 만듭니다.

그리고 구한 A행렬과 A^T를 구한 후 $R = A^T \cdot A$를 이용하여 covriance matrix를 생성합니다.

```
###################### Path ######################
path = "C:\\Users\\LG\\Desktop\\resize\\"
file_list = os.listdir(path)
##################################################

N = 5000 # 벡터공간을 생성할 이미지의 개수
train_data = np.zeros((N,1024)).astype(np.uint8) # 이미지를 행벡터로 나열하여Matrix구성

for i,file in enumerate(file_list):
if i == N:
break
img = cv2.imread(path+file)[:,:,0] # png파일에서 맨마지막채널만을 읽음
train_data[i] = np.resize(img,(1024))
```

우선, 과정을 간소화하기 위해 pgm파일을 32*32 png파일로 전처리를 하였습니다. 1.과정에서 crop한 아이겐페이스를 이룰 이미지의 개수를 설정하고(총 데이터의 양은 13000여장입니다.) 각각의 이미지를 (1024,)인 열벡터로 전환하여 train_data에 저장하였습니다.

- Find the mean vector (M) of your collected image vectors (F's)
  - Make the vector space by including the origin
- Subtract each face vector with the mean vector
  - Fk – M => ak
    - ▶ k : index of face image vector
- Make the data matrix, A with the column vector ak
  - A = [a1 a2 . . . . aN]

```
## A행렬구하기
# 1. 평균벡터구하기
meanFace = np.mean(train_data,axis=0)
# 2. normalize를 위해 행렬에서1.에서 구한 벡터 빼기
A = train_data - meanFace
A_T = np.transpose(A,(1,0))
cov = np.matmul(A_T,A) #covariance matrix
```

train_data배열은 m*m정방행렬이 아니므로 eigenvalue를 구하기 위해서는 SVD(Singular Value Decomposition)을 해야 합니다. 이 때, 각각의 이미지는 원래 -128~127의 픽셀값으로 이루어진 데이터이지만 시각화하기 위해 128을 더한 0~255의 픽셀값으로 이루어집니다. 하지만, 이러한 데이터들로 만든 공간은 원점을 포함하지 못해 meanface를 구해 각각의 이미지에 빼서 벡터공간을 이루게 해야합니다. SVD를 위해 covariance matrix를 A의

transpose(이하 A_T)와 A의 inner product를 하여 구합니다.

covariance matrix의 구성을 $A^T \cdot A$로 하였을 때, 자료의 개수가 전체차원의 개수보다 많아질 경우에 A를 열벡터로 구성한다면 covariance matrix의 행렬의 크기가 더 커질 것입니다. 하지만, 행벡터로 구성할 경우 행렬의 크기를 축소시킬 수 있습니다. 이 때, 어차피 행렬을 이루는 basis 즉, 기저벡터들의 개수는 모두 동일할 것이므로 연산을 줄이기 위해 행렬의 크기를 작게 하는 것이 더 효율적인 방법일 것입니다.

3. Apply SVD

❑ Apply SVD (PCA) to the covariance matrix

❑ Find some eigenvectors for the largest singular values.

- Number of eigenfaces is your choice.
- Dependent on your training face data

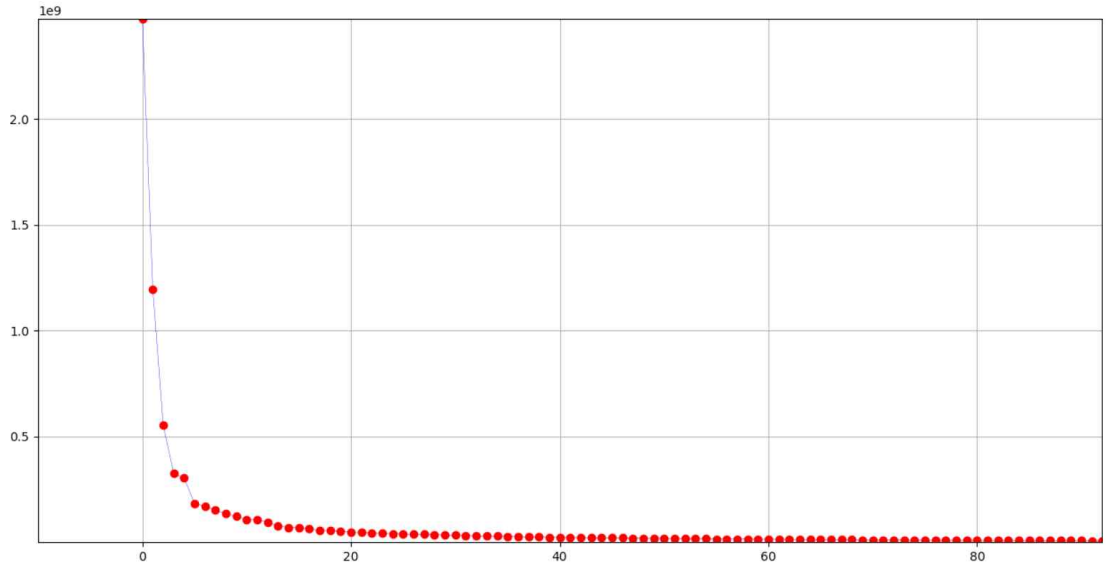
```
# 1.covariance matrix의eigenvalues, eigenvectors구하기
eig_val,eig_vec = np.linalg.eig(cov)
# 2. Principal component 구하기위해eigenvalues 내림차순으로 정렬
index = sorted(range(len(eig_val)), key=lambda k: eig_val[k])
index.reverse()
# 3. threshold를 설정하기
total_sum = eig_val.sum()
eig_faces = []
cur_sum = 0

for i in index:
cur_sum += eig_val[i]
threshold = cur_sum/total_sum
if threshold > 0.9:
break
eig_faces.append(eig_vec[:,i])
print("Principal Component eigvals의index: {0}".format(len(eig_faces)))
```

eigenface를 구하기 위해 covariance matrix의 eigenvector를 numpy.linalg.eig라이브러리 함수를 이용하여 구했습니다. 이 때, eigenvector(eigenface)를 그대로 image출력을 할 시 검정화면만이 출력됩니다. 여기서 구한 각각의 eigenvector들은 -128~127의 픽셀값으로 구성된 벡터들이므로 시각화하기 위해 아까 구한 meanvector를 더하여 출력해야만 우리가 알고있는 eigenface를 출력할 수 있습니다.

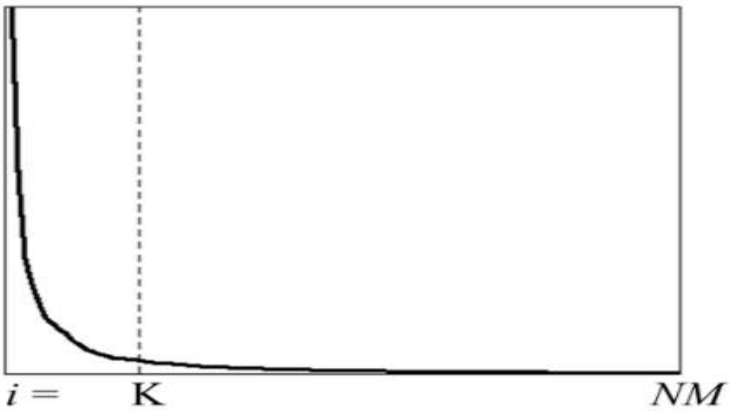
$$\frac{\sum_{i=1}^{K}\lambda_i}{\sum_{i=1}^{N}\lambda_i} > Threshold\,(e.g., 0.9 \text{ or } 0.95)$$

[수식1] 유효한 아이겐벡터의 비율

principal component를 구하기 위해 eigenvalue들을 시각화하여 그래프를 그릴 경우 다음과 같습니다.



[그림 3] eigenvalue값들의 그래프



[그림 4] 유효한 아이겐벨류를 찾기위한 그래프의 이해

이미지를 reconstruct하기위해 eigenvector들을 찾을 때, eigenvalue들을 내림차순하여 k번째까지의 eigenvector들을 찾습니다. 앞선 그래프를 보면 알 수 있듯이 아이겐벨류들의 값은 급격하게 줄어듦을 알 수 있습니다. Principal Component란 SVD과정을 통해 구한 아이겐 벨류들 중 0에 근접하지 않은 값들을 말하는데 실제로 계산을 해보면 0에 근접할 뿐 0이 되진 않습니다. 따라서, 프로그래머가 적당히 k번째까지를 끊어 유효한 index를 구해야 합니다. [수식1]에서 볼 수 있듯이 전체 eigenvalue들의 합과의 비율을 Threshold로 설정하여 index를 구했습니다. 이 때, 저는 13,233개의 image set 중 5000개의 image set을 이용하여 eigenvector들을 구했는데, 고유값의 합을 전체의 90%로 잡고 index를 구했을 시

```
print("Principal Component eigvals의index: {0}".format(len(eig_faces)))
```

```
Principal Component eigvals의 index: 82
```

다음과 같이 출력되어 82개의 eigenface들을 설정하였습니다.

[그림 8] eigenvalue가 높은 상위 30개의 eigenface모음

여기서 특이한 점은 3행 7열의 사진을 보면 웃고있는 듯한 모습을 알 수 있습니다. 따라서, 밑의 결과에서 웃고있는 모습을 reconstruct할 수 있을 것이라고 예상할 수 있습니다.

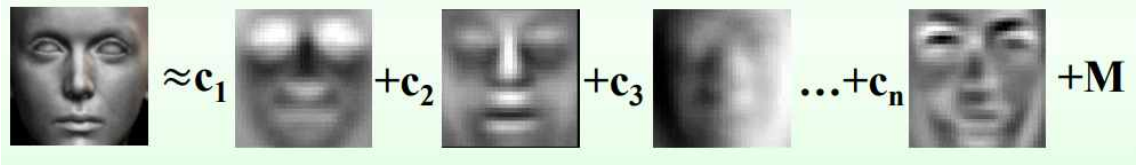
```
vec = np.array(eig_faces)
coef = np.zeros(len(eig_faces))

# test할 이미지를 통해c_k구하기
path = "C:\\Users\\LG\\Desktop\\eigenface_test\\"
file_list = os.listdir(path)
img = cv2.imread(path+file_list[0])[:,:,0]
img = cv2.resize(img,(32,32))
test = img.reshape(1024)
test = test-meanFace
for i in range(len(eig_faces)):
coef[i] = (test * vec[i]).sum() # [N,1] [1,N] inner project = [N,1] * [N,1]
```

❑ inner product of eigenface vector and test face image vector

- Eigenfaces => orthonormal

각각의 eigenface의 coefficient를 구하기 위해 양변에 각각의 eigenvector들을 내적을 합니다. 이 때, 앞선 numpy.linalg.eig라이브러리를 통해 구한 벡터들은 모두 orthonormal하게 구했기 때문에 한 eigenvector를 양변에 inner product할 경우 해당되는 벡터를 제외한 모든벡터들은 inner product value가 0으로 바뀝니다. 그리고 테스트할 이미지는 열벡터로 전환하였고, eigenvector들 또한 열벡터이기 때문에 numpy.dot을 할 필요없고 multiplication 연산한 값의 합으로도 inner product가 표현 될 것입니다.

```
# reconstruct한 이미지는( coefficient_k * eigenface_k )의 합+ meanface
result = np.zeros((1024))
for i in range(len(eig_faces)):
result += (coef[i] * vec[i])
result += meanFace

result = result.reshape((32,32)).astype(np.uint8)
ret = np.concatenate((img, result), axis=1)
```



[그림 9] eigenface를 기반으로한 image의 reconstruct

reconstruct할 이미지는 [그림 3]과같이 각각의 eigenface에 coefficient를 곱한 값의 합에 원래 eigenface를 구성할 이미지의 meanface를 더하면 됩니다. 주의할점은 여기서의 meanface는 테스트할 이미지의 평균이 아니라 앞서 구했던 eigenface를 구한 이미지의 평균이라는 것입니다.

```
ret = cv2.resize(ret,(640,320),interpolation=cv2.INTER_LANCZOS4)
cv2.imshow("frame",ret)
key = cv2.waitKey(0)
cv2.destroyAllWindows()
```

[그림 10] reconstruction한 사진과 비교 (좌측 : 원래이미지, 우측 : reconstruct한 이미지, 32*32 이미지를 320*320으로 확대함 이 때의 보간법은 INTER_LANCZOS4를 사용)

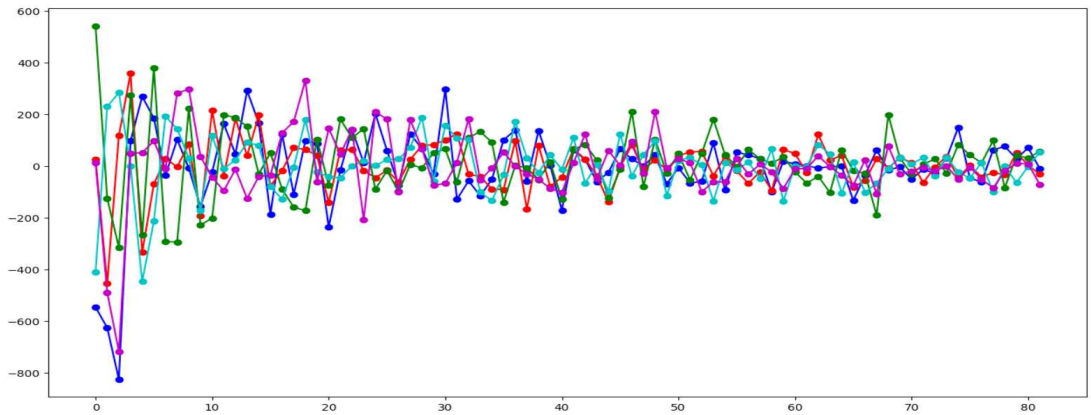
❑ Collect 10 different cropped face images

➢5 test images for one face

- Represent each face images using the eigenfaces
- Compare the coefficients {c1, c2, …, cn} for face recognition

1. Abdullah_Gul

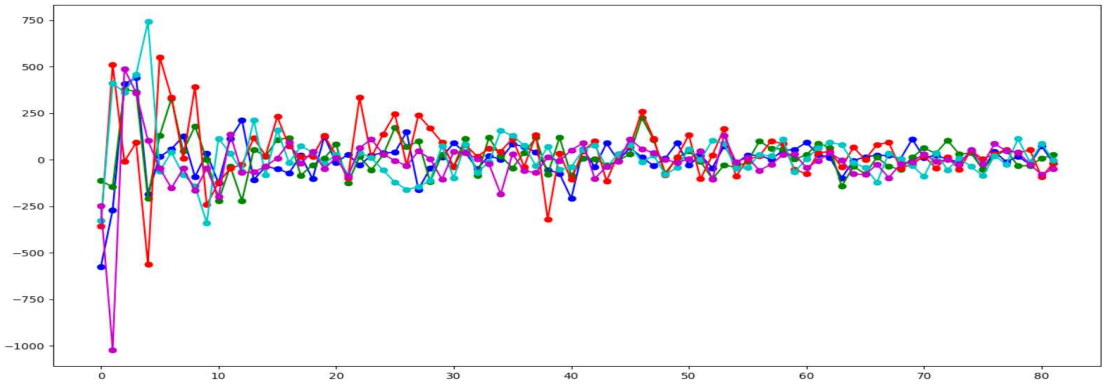
첫 번째 사람은 수염을 가지고 있는 인물이었습니다.
이 사람의 coefficient의 값을 그래프로 살피면 다음과 같습니다. 이는 밑에서 살펴볼 그래프와 굉장히 다른 점을 확인할 수 있습니다. 수염이 있어서 그런지 편차가 가장 컸고 들쑥날쑥한 그래프입니다. 이를 통해 eigenface를 구성한 데이터 속에는 수염을 가진 인물이 드물어 수염을 표현하는 부분이 적어 다른 얼굴들로 대체를 하여 값이 일정하지 못함을 알 수 있습니다.



이는 밑(face recognition)에서 다시한번 다뤄보겠습니다.

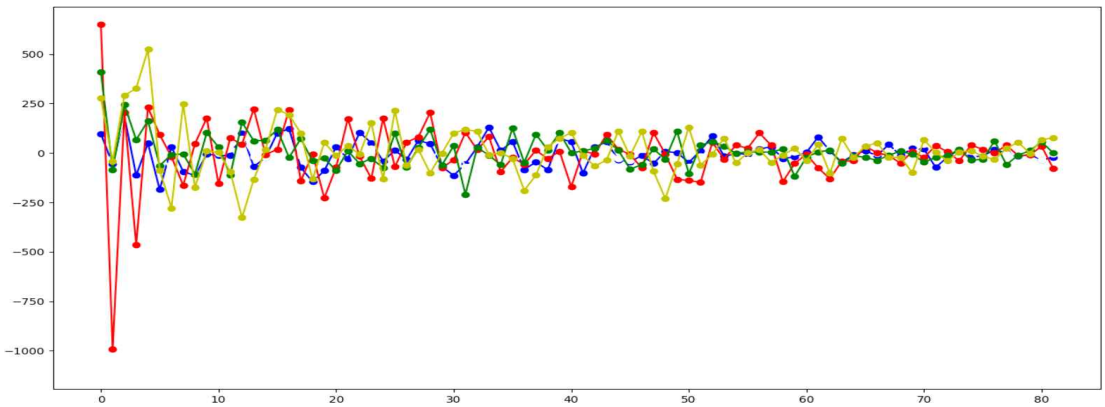
2. Adrien_Brody

이 사람에서 주목할 점은 두 번째, 세 번째 사진에서 선글라스를 쓰고있어 표현이 덜된다는 점입니다. 다만, 어느정도 표현은 되어 있는 것을 알 수 있습니다.



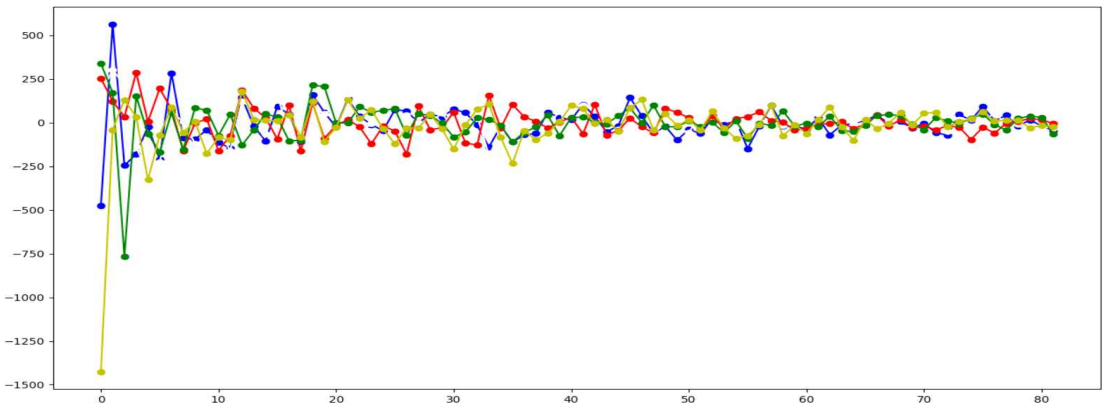
3. Al.Gore

2.사람과 다르게 얇은테를 쓰고있어 아예 표현이 되지 않는다는 것을 알 수 있습니다. 앞선사람은 선글라스여서 이 부분이 어둡게 표현되어 선글라스임이 약간 유추될 수 있었지만 얇은테는 아예 표현이 되지 않는다는 것을 알 수 있습니다.



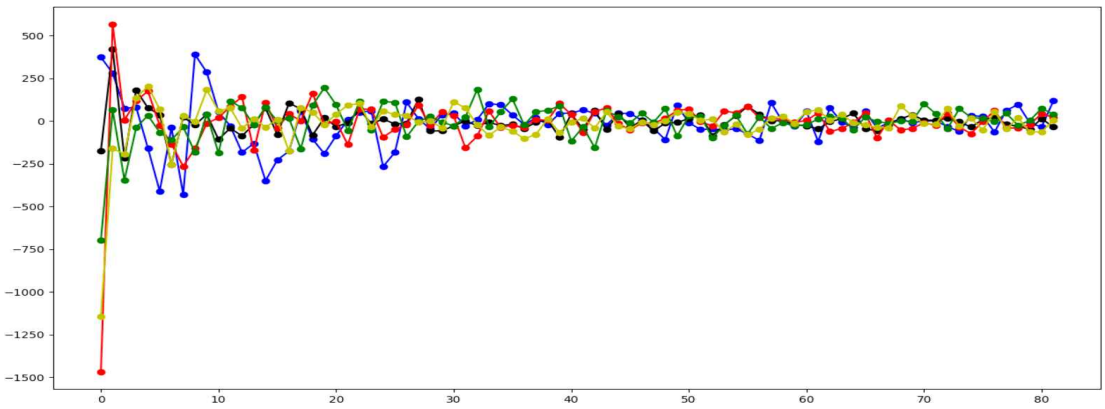
4. Alejandro_Toledo

이 사람도 마찬가지로 모든사진이 어느정도 동일하게 표현되어있지만 마지막그림에서 얇은테를 쓰고있음에도 아예표현이되지 않는다는 것을 알 수 있습니다.

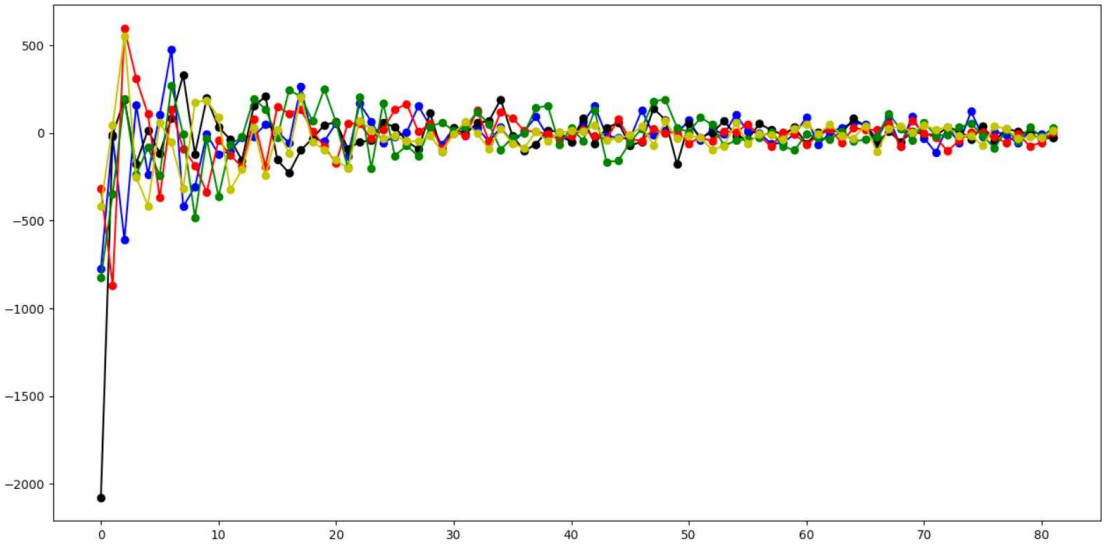


5. Ali_Naimi

이 사람은 다양한 표정을 짓고 있는 모습인데, 4번의 모습을 육안으로 확인했을 때는 찡그린표정이지만 reconstruct된 사진에서는 웃고있는 모습과도 같았습니다. 이는 앞선 eigenface집합에서 웃고있는 eigenface가 존재하여 그러한 모습이 반영되었다고 할 수 있습니다.

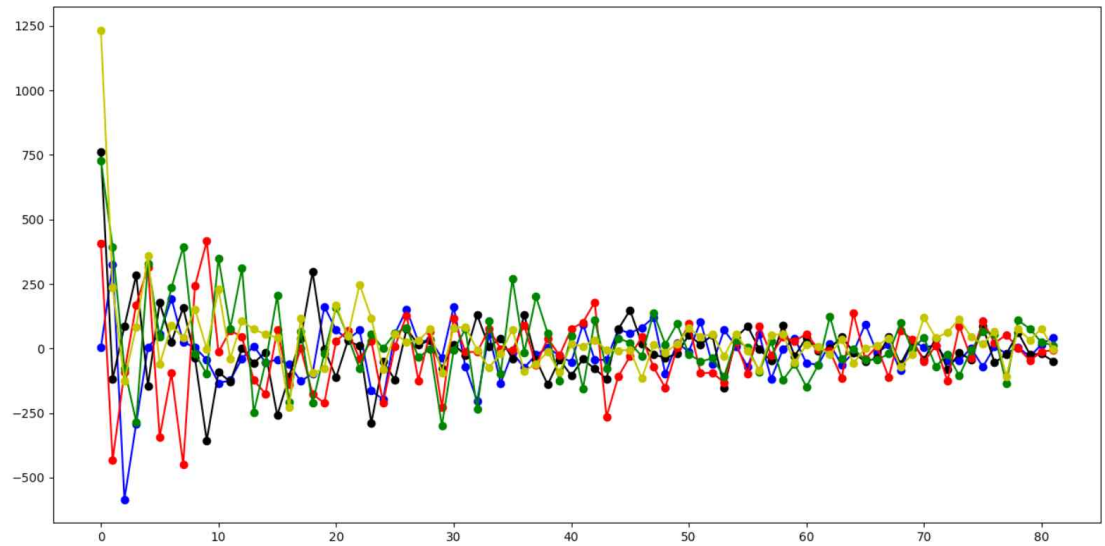


6. Ana_Palacio



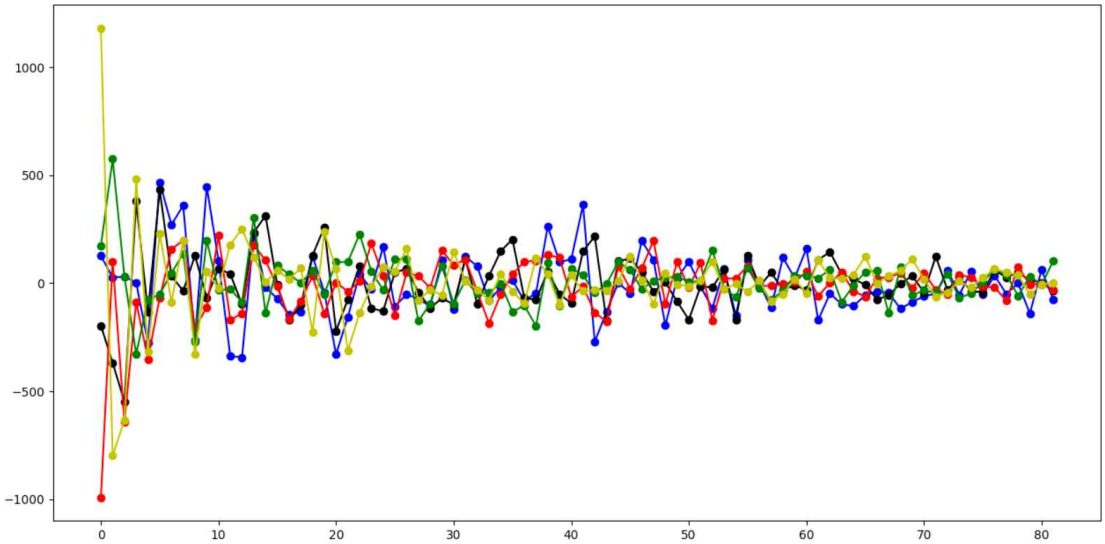
7. Andre_Agassi

이 사람의 가장 큰 특징은 측면의 얼굴이 거의 제대로 출력되지 않았다는 것입니다. 또한, 맨마지막의 찡그린 표정을 보면 표정의 극적인 부분이 표현되지 않음을 알 수 있습니다.



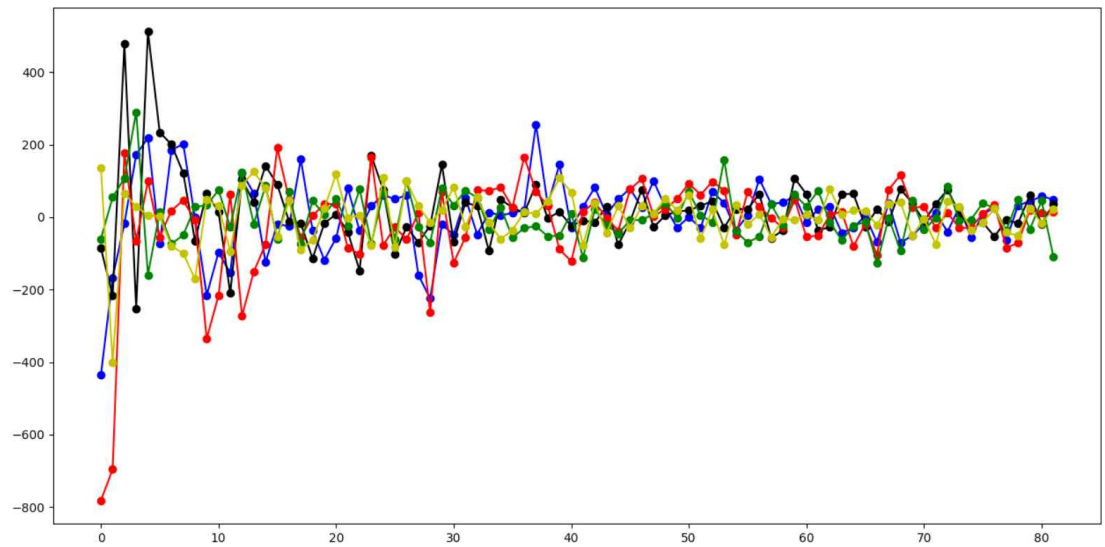
8. Andy_Roddick

두 번째와 세 번째 사진에서 알 수 있는 점은 얼굴의 각도를 약간만 틀어도 이미지가 잘



표현되지 않는다는 점입니다. 그리고 아래의 coefficient값들의 분포를 보아도 index가 낮은 즉, eigenvalue가 높은 eigenface에서의 값의 편차가 굉장히 크고 다른 사람들의 그래프와 달리 모든 영역의 값들이 편차가 큼을 확인할 수 있습니다. 이를 통해, 이 사람의 얼굴을 표현할 eigenface가 제 데이터셋의 eigenface로는 구현할 수 없어 제각각인 것이라는 생각을 했습니다.

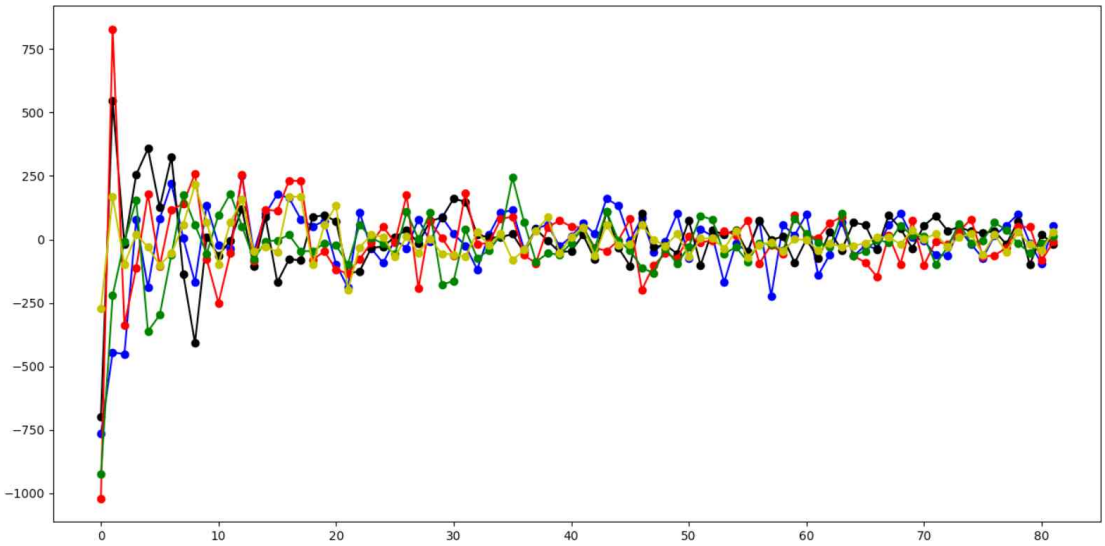
9. Angelina_Jolie



위와 마찬가지로 표현이 안되는 얼굴이 육안으로도 확인되는데 이럴 경우에 coefficient들의 값들이 굉장히 많이 튐을 알 수 있습니다.

10. Ann_Veneman

첫 번째 사진에서 입을 다물고 있는 것은 어느정도 표현이 되는 걸 알 수 있지만 눈이 표현되지 않는다는 점에서 얼굴을 약간 틀고 입을 다물어도 reconstruct가 제대로 이루어지지 않음을 알 수 있습니다. 또한, 네 번째 사진을 보면 얼굴이 기울어져있고 웃고있을 때, 코와 눈이 뭉개지는 걸 확인할 수 있습니다. 또한, 이를 coefficient의 그래프로 살펴보았을 때 편차가 큼을 통해 데이터의 수가 부족함을 알 수 있습니다.



○ Conclusion

1. EigenVector의 수를 조절하여 가장 잘 reconstruct된 이미지를 찾기

```
for i in index:
cur_sum += eig_val[i]
threshold = cur_sum/total_sum
if threshold > 0.99:
break
eig_faces.append(eig_vec[:,i])
```

다음과 같은 코드를 통해 eigenvector들의 수를 비율을 통해 조절할 수 있었습니다. 이는 실제 1024차원의 공간을 N%의 벡터들 즉, N차원 부분공간으로 표현하여 원래의 data의 차원을 낮추는 것이라고 할 수 있습니다. 이를 data compression과정이라고 할 수 있습니다. 전체 eigenvalue의 90퍼센트를 설정하였을 때는 eigenvector가 82개였지만 99퍼센트로 설정했을 때는 다음과같이 456개의 벡터들을 설정함을 알 수 있습니다.

```
Principal Component eigvals의 index: 456
```

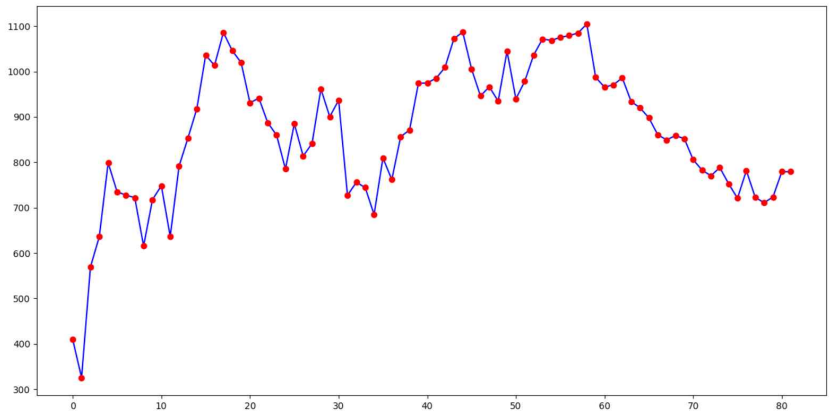
[그림 36] # of Eigenface = 456일 때(Threshold = 0.9), 2.사람을 reconstruct한 이미지

[그림 37] # of Eigenface = 82일 때(Threshold = 0.99), 2.사람을 reconstruct한 이미지

차원축소의 비율을 약 1/100 : 1/2로 했을 때의 차이가 다음과같이 큼을 알 수 있습니다. 차원축소, data compression을 많이하게 된다면 데이터의 양은 줄어들고 속도가 빨라지겠지만 정확도가 현저히 떨어짐을 알 수 있고 반대로 적게할 경우 데이터의 양은 많고 속도가 느려지지만 정확도 측면에서는 확실이 높음을 알 수 있습니다. 이를 통해 얼굴인지 아닌지 판단하는 정도는 많은 data compression을 통해 빠르고 효율적으로 처리하는 방법을 사용하고 사람을 구별하는 데에 있어서는 적은 data compression을 통해 정확도를 높이는 것이 더 중요할 것입니다.

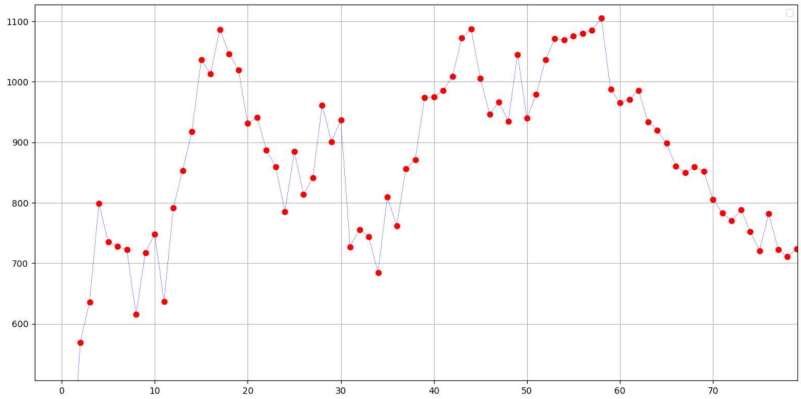
2. reconstruct된 이미지들의 coefficient분석하기

앞선 [그림 26]의 4번째 사진을 분석해보자면



[그림 38] 0~82까지의 eigenface로 구성된 coefficient값들의 그래프

coefficient들의 값이 같은 index에서 거의 똑같다는 것을 알 수 있습니다. 따라서, 안경을 표시하는 부분은 다른 eigenvector들에서 표현한다고 할 수 있습니다. 처음에는 coefficient의 값이 조절되면서 안경이 표현된다고 생각했지만 안경을 표현하는 eigenvector들은 기존의 eigenvector들이 쓰이는 것이 아닌 것입니다.

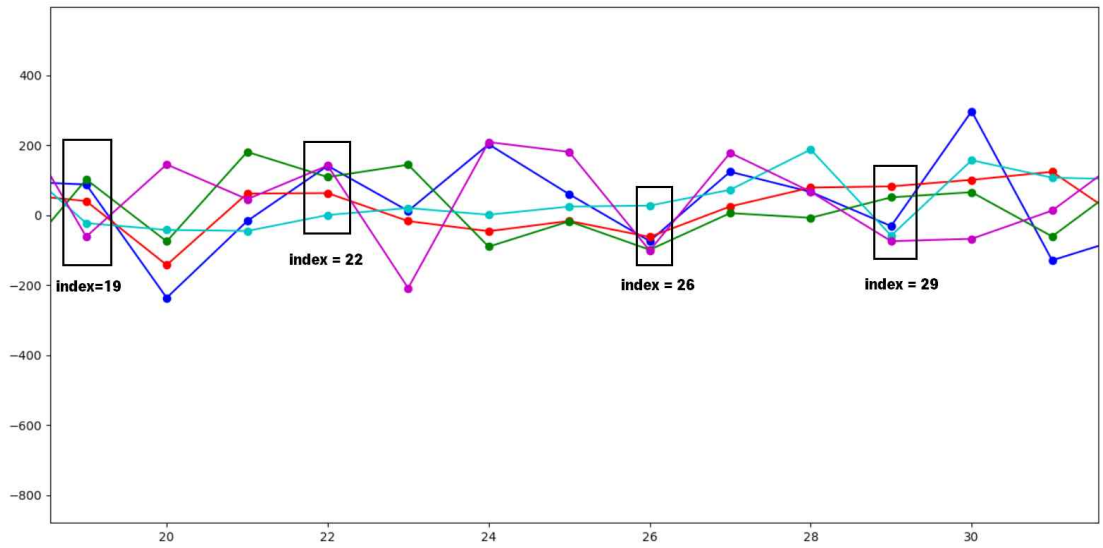


[그림 39] 0~456까지의 eigenface로 구성된 coefficient값들의 그래프

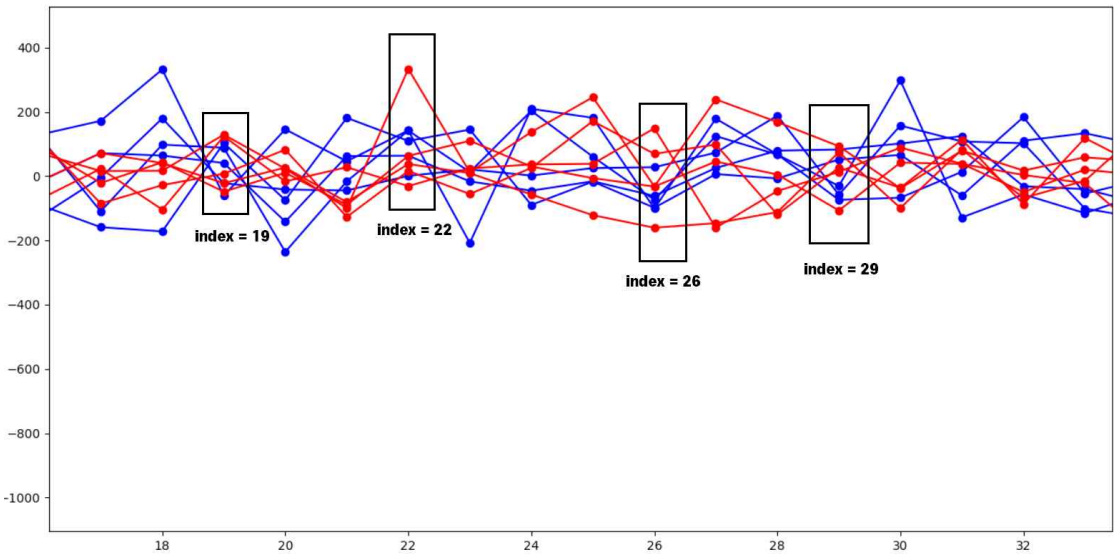
기존에 image를 collect할 때 안경을 쓰고있는 사진들을 기준으로 eigenface를 생성했을 때, 90퍼센트의 threshold로 eigenface를 구성하였다면 그 중에 안경을 표현하는 부분이 있어 더 적은 자료들로 안경을 인식할 수 있을 것입니다. 99퍼센트의 eigenface를 사용한다면 더 정확한 이미지를 사용할 수 있지만 앞선 실험에서 eigenvalue들의 합을 9퍼센트를 높임에 있어서 eigenface의 수는 약 400퍼센트 증가하였습니다. 이는 비효율적임을 알 수 있습니다. 따라서, 얼굴을 인식하는 기능을 만들 때, 처음에 안경을 쓴 얼굴의 집합으로 만든 80여개의 eigenface로 구성된 것으로 확인한 후 썼다면 그대로 인식을하고 안경을 쓰지 않은 얼굴이라면 안경을 쓴 얼굴의 집합으로 만든 eigenface로 확인한다면 456개의 eigenface가 아닌 80+80 = 160여개의 eigenface로도 확인할 수 있을 것입니다. 더 나아가 안경 뿐아니라 현 시국에 마스크를 많이 착용하여 얼굴인식이 잘 되지 않고있어 문제가 되어있는데 처음에 마스크를 썼는지를 확인하고 그다음 사용자의 얼굴의 나머지부분을 인식하여 확인한다면 마스크를 쓴 상태에서도 얼굴인식프로그램을 사용할 수 있을 것입니다.

3. face recognition

사람의 얼굴에 따른 coefficient를 분석하여 다른사람의 얼굴이라는 것을 인식하기 위해 우선 그래프를 그려보았습니다. 앞선 분석과정에서 eigenvalue의 값의 차이가 최대 200이 넘지 않는 부분을 따로 찾아보았습니다. 우선 index가 낮은 즉, eigenvalue가 높은부분들에서의 계수값들은 대체로 200이 넘는 것을 확인할 수 있습니다. 하지만 특이한 점은 index20번대에서 200을 넘지 않는 것들이 많이 발견되고 30번대에서는 거의 발견되지 않는다는 점이었습니다. 따라서, 20번대의 eigenface가 얼굴을 구별하는 데에 있어서 유의미한 데이터라 판단했습니다. 그리고 그 이후의 eigenvalue들은 값의 변화가 미미하므로 우선 고려대상에서 제외하였습니다.



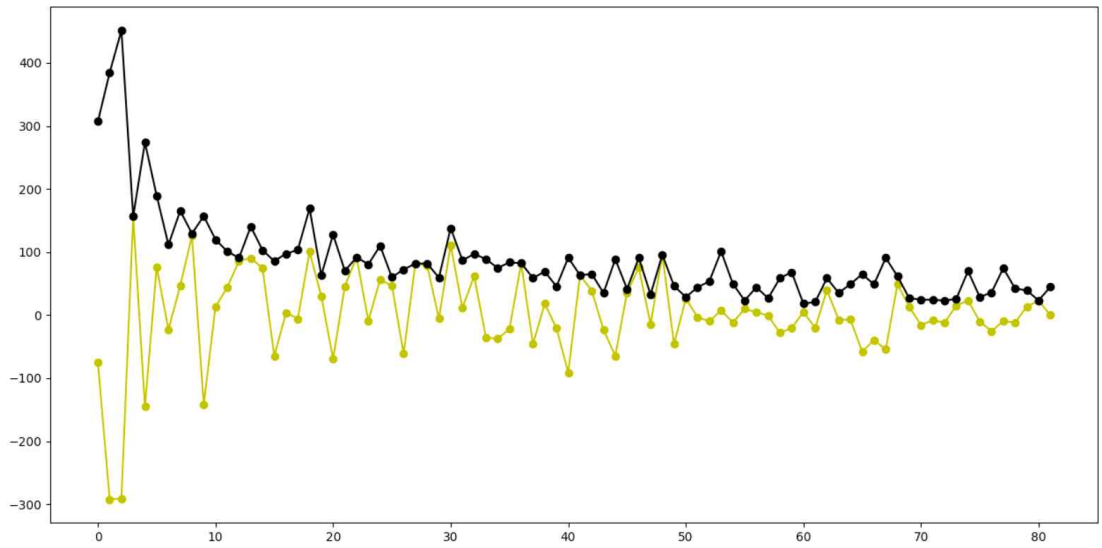
각각의 계수들이 비슷하다는 것은 다른 얼굴을 입력하더라도 같은 사람의 얼굴이라면 비슷한 계수가 측정된다 생각하여 다른 사람의 것들과 비교해보았습니다.
이번엔 Abdullah_Gul과 Adrien_Brody의 그래프를 각각 비교분석 해보았습니다.



index 19번과 29번은 마찬가지로 eigenvalue의 값의 차이가 200을 넘어가지 않는 것을 알 수 있습니다. 차이가 많이 나지 않는다는 것은 좋은 결과였지만 다른 사람의 얼굴임에도 불구하고 비슷한 값이 나온다는 것은 두 얼굴의 특징을 구별할 수 없으므로 특정한 eigenface의 coefficient로는 구별지을 수 없다는 결론을 내렸습니다.

두 번째 방법으로는 coefficient의 mean vector를 찾은 후 각 데이터들의 mean vector와의 편차의 절대값을 평균낸 벡터를 찾아 이들의 합이 달라지는 추이를 보겠습니다.
1. coefficient들의 mean vector를 찾는다.
2. 각 데이터들의 mean vector들과의 편차의 절대값을 평균낸 벡터를 찾는다.

3. 같은얼굴의 새로운 사람의 얼굴이 입력되었을 때의 편차를 보고 오차를 설정하여 비교한다.



[그림 42] mean value집합(노랑)과 편차의 평균의 집합(검정)

Abdullah_Gul의 편차 평균의 합은 다음과 같이 출력되었습니다.

```
7137.584312235847
```

다른인물의 편차평균의 합을 출력하였더니 각각 다음과 같이 출력되었습니다.

Al.Gore : `5747.1073188900955`

평균의합과 새로운 이미지의 coefficient의 값의 편차의 합 :

```
5747.1073188900955
7755.063083151421
```

하지만 평균의 합과 같은이의 새로운 이미지의 편차가 굉장히 큼을 알 수 있습니다. 따라서 다음과 같은 방법 또한 의미 없는 데이터임을 알 수 있었습니다.