EigenFace Project

FVE6013 Machine Learning and Pattern Recognition, Inha University
Team: Team #2
Presenter: Jumabek Alikhan

Overview

Task Definition

Nearest Neighbor Face Recognition

EigenFace with Nearest Neighbor Face Recognition

[Source Code Link: https://github.com/Jumabek/eigenface](https://github.com/Jumabek/eigenface)

Task

Face Recognition with EigenFace approach:

Use PCA to reduce image dimensionality

Nearest neighbor classifier with L2 distance

Dataset:

Train images: 25 unique subjects. Single image per subject

Test images: 20 pairs of subjects . 2 image per subject

Nearest Neighbor Face Recognition WITHOUT PCA

Steps:

1. Read images - N,D dimensions
2. Define Distance function - computes distance between two images
3. Obtain genuine and imposter matching scores
4. Find the best threshold and calculate performance metrics FAR(FMR) and FRR(FNMR)

Step 1: Read Images

```
def read_test_images():
    filenames = [f for f in glob.glob(join('images/Test','*.jpg'))]
    filenames = sorted(filenames)
    print("There are {} test images".format(len(filenames)))

    images = [cv2.imread(f,cv2.IMREAD_GRAYSCALE) for f in filenames]
    images = np.reshape(images, (len(images),-1))
    labels = [int(ntpath.basename(f)[:5]) for f in filenames]

    return np.array(images),np.array(labels)
```

```
def read_train_images():
    filenames = [f for f in glob.glob(join('images/Train','*.jpg'))]
    filenames = sorted(filenames)
    print("There are {} train images".format(len(filenames)))

    images = [cv2.imread(f,cv2.IMREAD_GRAYSCALE) for f in filenames]
    images = np.reshape(images, (len(images),-1))
    labels = [int(ntpath.basename(f)[:5]) for f in filenames]

    return np.array(images),np.array(labels)
```

Step 2: Define Distance Function

L2 distance

Note ***a*** and ***b*** are vectors

```
def compute_l2_distance(a,b):
    # Input:
    # a - D dimensional image as a row
    # b - D dimensional image as a row

    # Returns `distance` scaler value
    distance = np.sqrt(np.sum((a-b)*(a-b)))
    return distance
```

Step 3: Obtain Matching Scores

Note: In loading phase test images are ordered in specific way

so that every next pair of images belong the same class. Example:

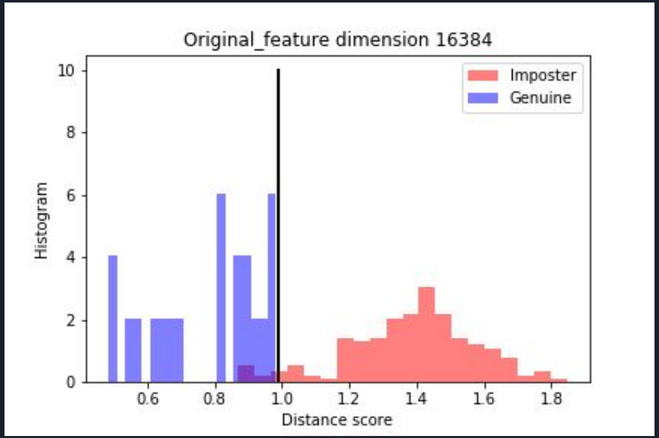
['images/Test/00770_960530_fa.jpg', 'images/Test/00770_960530_fa_a.jpg', 'images/Test/00771_941205_fa.jpg', 'images/Test/00771_941205_fb.jpg']

```
In [9]: # genuine_matching_scores
        def get_genuine_scores(X_test):
            num_test = X_test.shape[0]
            # Use pairs of same person to compute the score distribution of
            #genuine matching scores.
            genuine_matching_scores = [compute_l2_distance(X_test[i],X_test[i+1]) for i in range(0,num_test,2)]
            return genuine_matching_scores

        def get_imposter_scores(X_test):
            num_test = X_test.shape[0]
            #Use the first image of each person to matching with the first images
            #of others to compute the imposter matching scores
            imposter_matching_scores = []
            for i in range(0,num_test,2):
                for j in range(i+2,num_test,2):
                    imposter_matching_scores.append(compute_l2_distance(X_test[i],X_test[j]))
            return imposter_matching_scores
```

Step 4: Finding threshold and FAR/FRR

- For threshold of 0.99
- Falsing Matching Rate(FMR/FAR)%: 3.68
- False Non-Matching Rate(FNMR/FRR)%: 0




```python
def get_threshold_and_performance_metrics(genuine_scores,imposter_scores):
    min_error = len(genuine_scores) + len(imposter_scores)
    min_threshold = 0
    #print(len(genuine_scores), len(imposter_scores))
    all_scores = genuine_scores + imposter_scores
    #print(len(all_scores))
    for threshold in all_scores:
        # Type1 error: rejecting genuine face - FRR
        type1 = len([s for s in genuine_scores if s >= threshold ])
        # Type2 error: accepting imposter - FAR
        type2 = len([s for s in imposter_scores if s < threshold ])
        num_errors = type1 + type2
        if num_errors < min_error:
            min_error = num_errors
            min_threshold = threshold

    FRR = type1/len(genuine_scores)
    FAR = type2/len(imposter_scores)
    # print(min_error, min_threshold)
    return min_threshold,FAR,FRR
```

That was Nearest Neighbor Face Recognition on Raw Pixels of 128x128 images

Problem is Computational Expense

Next, EigenFace AKA Nearest Neighbor Face Recognition on reduced (PCA) features

Nearest Neighbor Face Recognition WITH PCA

Steps:

1. Read images - N,D dimensions
2. **Obtain Projected images from PCA (first need to implement PCA)**
3. Define Distance function - computes distance between two images
4. Obtain genuine and imposter matching scores
5. Find the best threshold and calculate performance metrics FAR(FMR) and FRR(FNMR)

PCA implementation

Steps:

1. Calculate Covariance Matrix
2. Find EigenValues/EigenVectors and Sort them in descending order
3. Project image into new best `K` dimensions (aka Principal Components)

Tools:

1. Numpy.linalg.ieg - to compute eigenvalues / eigenvectors

Step 1: Computing Covariance Matrix

```
# find covariance matrix C  [p x p]
from sklearn.preprocessing import normalize
num_train = X_train.shape[0]
mean = np.mean(X_train,axis=0) # p
B = X_train - mean # n x p
B = normalize(B) # I think normalizing will speed up eigenvector/eigenvalue matrix factoriz
C = 1./(num_train-1)*np.dot(B.T,B) # [p x p]
```

```
compute_n_save_eigens(C) # finds, sorts and saves eigen vecor and eigen values
```

```
Computed,sorted and saved eigens
```

```
# loading saved eigenvectors
eigenvectors = np.load('eigenvectors.npy')
eigenvalues = np.load('eigenvalues.npy')
print('Loaded eigen vectors of shape ', eigenvectors.shape)
```

Step1: Covariance matrix original and after 100 PC projection

Covariance Matrix visualization

For original image convariance Matrix

```
#
# reszie train image (128,128) -> (10,10) so that we can compare Cov matrix with PCA 100 CovMat
X_train_reshaped = X_train.reshape((-1,128,128)) # returning to original shape
sz = (10,10) # we will resize (128,128) --> (10,10)
new_X_train = []
for i in range(X_train_reshaped.shape[0]):
    new_X_train.append(cv2.resize(X_train_reshaped[i],sz))
new_X_train = np.array(new_X_train)
new_X_train = new_X_train.reshape(new_X_train.shape[0],-1)
```

```
C = find_covariance_matrix(new_X_train)
C = C*C
C = normalize_to_image_range(C,increase_whiteness=100)

cv2.imwrite('Cov.png',C)
```

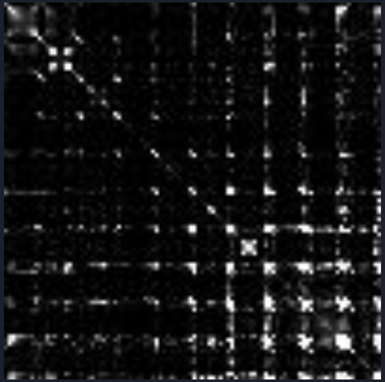
```
True
```

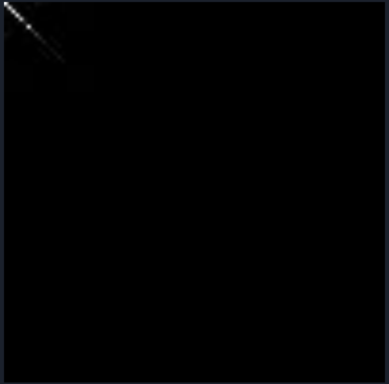
For new image with 100 PCs

```
n_components=100
X_train_projected = pca(eigenvectors,X_train,n_components=n_components)
C = find_covariance_matrix(X_train_projected)
C = C*C
C = normalize_to_image_range(C,increase_whiteness=n_components)

cv2.imwrite('Cov-for-{}PCs.png'.format(n_components),C)
```

```
True
```





Step2: Computing EigenValues/EigenVectors from Covariance Matrices

```
# saving eigenvectors in sorted fashion only once
def compute_n_save_eigens(C):
    orig_eigenvalues, orig_eigenvectors = np.linalg.eig(C)
    eigenvectors = np.real(orig_eigenvectors)
    eigenvalues = np.real(orig_eigenvalues)

    # we need descending order, remember higher eigenvalue means more important dimension
    indices = np.flip(np.argsort(eigenvalues))
    eigenvectors = eigenvectors[:,indices] # note, sorting column because x[:,i] columns cor
    eigenvalues = eigenvalues[indices]

    #saving matrices, because computing was a bit time consuming
    np.save('eigenvectors.npy',eigenvectors)
    np.save('eigenvalues.npy',eigenvalues)
    print("Computed,sorted and saved eigens")
```

Step 2: visualizing eigenfaces

Visualizing EigenFaces

```
def normalize_to_image_range(x,increase_whiteness=1):
    x *= 255./np.max(x)*increase_whiteness
    x = np.minimum(255,x)
    return x
```

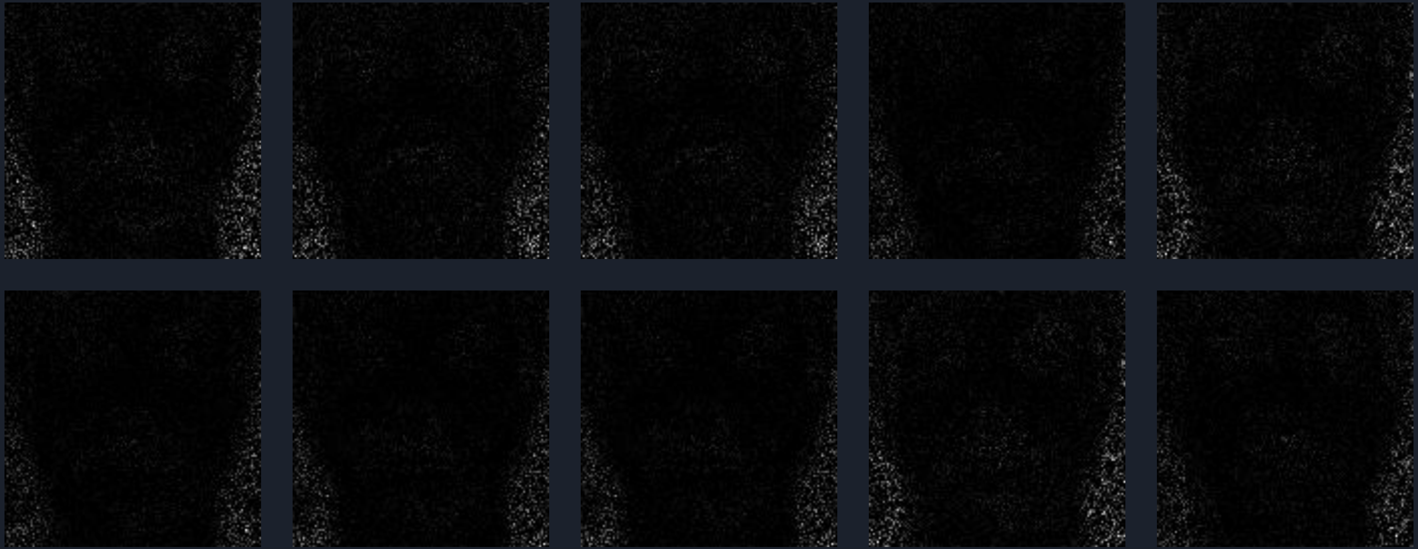
PCA with n_components = 100

```
num_eigenfaces_to_visualize = 100
sz = (128,128)
# eigenvectors [D x D] where D = 128*128
for i in range(num_eigenfaces_to_visualize):
    eigenface = eigenvectors[:,i] # [D x 1]
    filename=  'images/visualizations/eigenface_{0:05}.png'.format(i)
    eigenface = eigenface.reshape(sz) #[128 x 128]
    eigenface = normalize_to_image_range(eigenface)
    cv2.imwrite(filename,eigenface)
```

Step 2: visualize 10 best eigenfaces

Step 2: Visualize 20-30th best eigenfaces

Step 2: Visualize 90-100th eigenfaces



Step3: PCA projection

```
def pca(eigenvectors,img,n_components=10):
    # Input:
    # img           -   [N x p]
    # eigenvectors/principal components -   [p x p] - each columns is eigenvector sorted acc

    # Returns:
    # projected_img - [N x L] where L<<p

    best_components = eigenvectors[:n_components] # [p x L]
    projected_img = np.dot(img,best_components)
    return projected_img
```

Results

Metric interpretations

- False Matching Rate 1%: **1** out of 100 imposter face **unlocks your phone**
- False Non-Matching Rate 90%: Out of 100 attempts **90 times you cannot unlock** your phone

Number of Components	Threshold distance	False Matching Rate(FMR/FAR) %	False Non-Matching Rate (FNMR/FRR) %	imgs/sec
10	.33	0.00	50	
23	0.65	3.16	10.0	
24	0.65	3.16	10.0	
100	.68	3.68	5.0	
500	.70	3.68	0	
1000	.72	3.68	0	
2000	.74	3.68	0	
5000	0.77	2.63	0.00	
Without pca 128*128	.99	3.68	**0**	

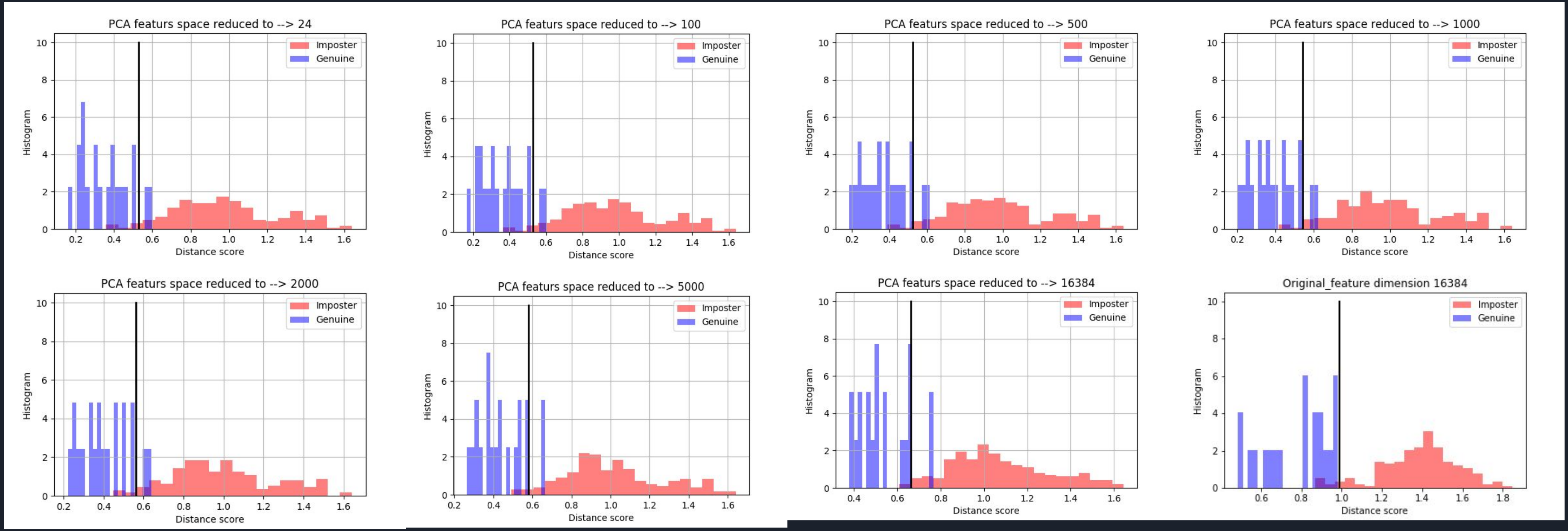
Results

Metric interepreations

- False Matching Rate 1%: **1** out of 100 imposter face **unlocks your phone**
- False Non-Matching Rate 90%: Out of 100 attempts **90 times you cannot unlock** your phone

Number of Components	Threshold distance	False Matching Rate(FMR/FAR) %	False Non-Matching Rate (FNMR/FRR) %	Imgs/sec
10	0.31	0.00	35.0	160,000
22	0.39	0.53	30.0	160,000
24	0.53	2.63	10.0	160,000
100	0.53	2.63	10.0	160,000
500	0.52	2.11	10.0	130,000
1000	0.54	2.11	10.0	120,000
2000	0.56	2.11	10.0	97,000
5000	0.58	1.58	10.0	65,000
Without pca	0.74	1.58	20.00	29,000

Visualizing Thresholds

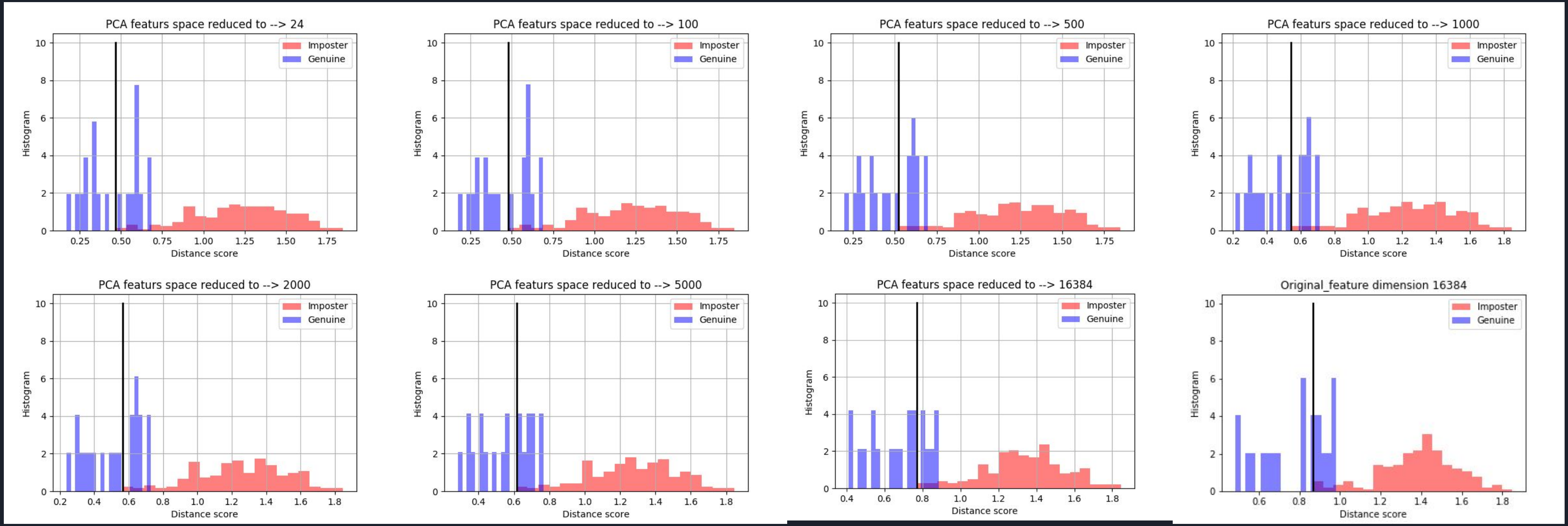


Practical Considerations

- Type 2(FAR/FMR) error is more severe than Type1 (FRR/FNMR) (Biometrics-ICE7058, Huan Van Nguyen, Ph.D.)
- So lets set Type 2 error to small as possible by penalizing 10x more

Number of Components	Threshold distance	False Matching Rate(FMR/FAR) %	False Non-Matching Rate (FNMR/FRR) %
10	0.31	.0	**35.0**
22	0.36	.0	50
24	0.36	.0	50
100	0.37	.0	50
500	0.39	.0	45
1000	0.42	.0	40
2000	0.45	.0	40.0
5000	0.49	.0	**35.0**
128*128=16384	0.61	**.0**	**35.0**
W/O PCA 128*128	0.66	.0	35.00

Visualizing Thresholds: Type 2 error 10x penalty



Conclusion

EigenFace is Effective - reduces computation by reducing dimension

Visualizing Eigenfaces could help us determine number of PCs to keep

PCA sometimes can improve performance by removing noisy pixels

Lessons

Covariance Matrix tells a lot about data feature space

Best feature representation results in Covariance Matrix that only consists elements in the diagonal

Each PC is a vector of ***D*** dimensions where ***D*** is the dimension of the features in original data