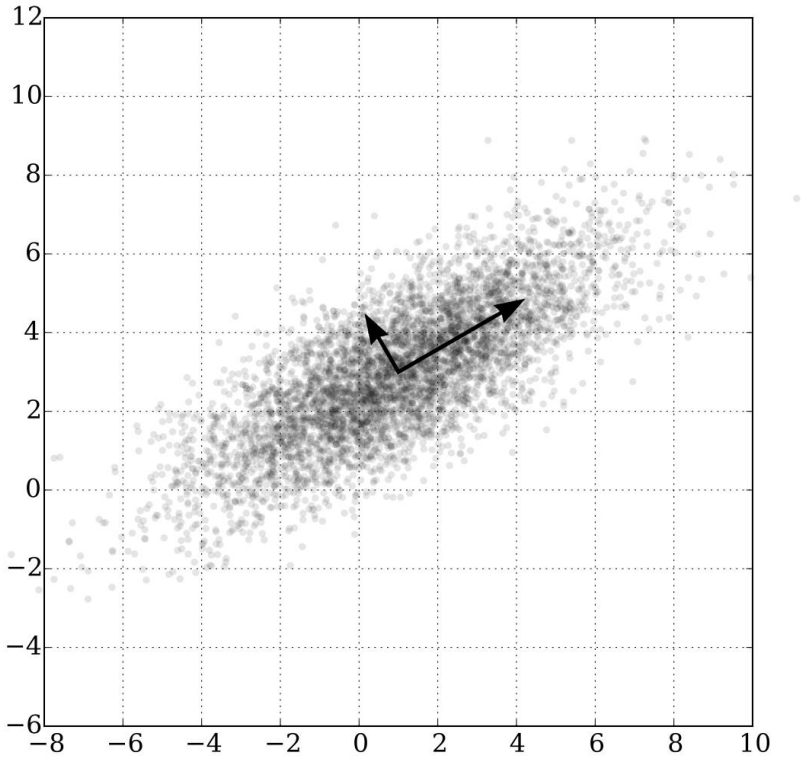
大数据算法结课报告

对Eigenfaces和Fisherfaces的一些探讨和思考

报告人：王家伟，缪立君

目录

Eigenfaces



https://en.wikipedia.org/wiki/Principal_component_analysis

Algorithmic Description

Let $X = \{x_1, x_2, \ldots, x_n\}$ be a random vector with observations $x_i \in R^d$

1. Compute the mean μ

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i$$

2. Compute the the Covariance Matrix S

$$S = \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu) (x_i - \mu)^T$$

3. Compute the eigenvalues λ_i and eigenvectors v_i of S

$$Sv_i = \lambda_i v_i, i = 1, 2, \ldots, n$$

4. Order the eigenvectors descending by their eigenvalue. The k principal components are the eigenvectors corresponding to the k largest eigenvalues. The k principal components of the observed vector x are then given by:

$$y = W^T (x - \mu)$$

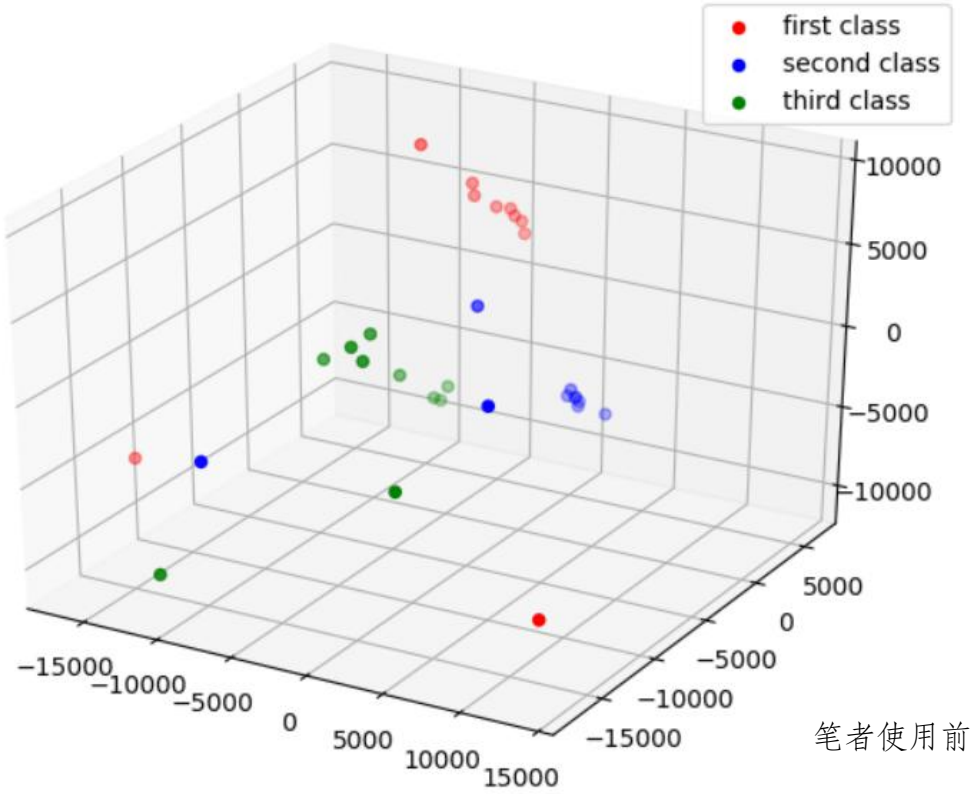
where $W = (v_1, v_2, \ldots, v_k)$. The reconstruction from the PCA basis is given by:

$$x = Wy + \mu$$

Eigenfaces method for face recognition by:

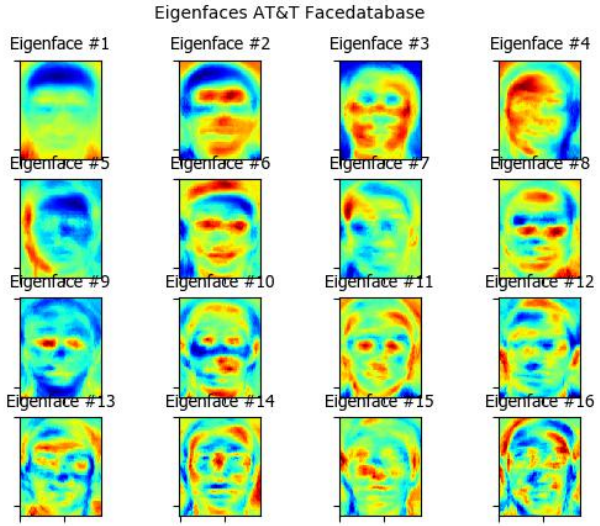
1. Projecting all training samples into the PCA subspace
2. Projecting the query image into the PCA subspace
3. Finding the nearest neighbor between the projected training images and the projected query image.

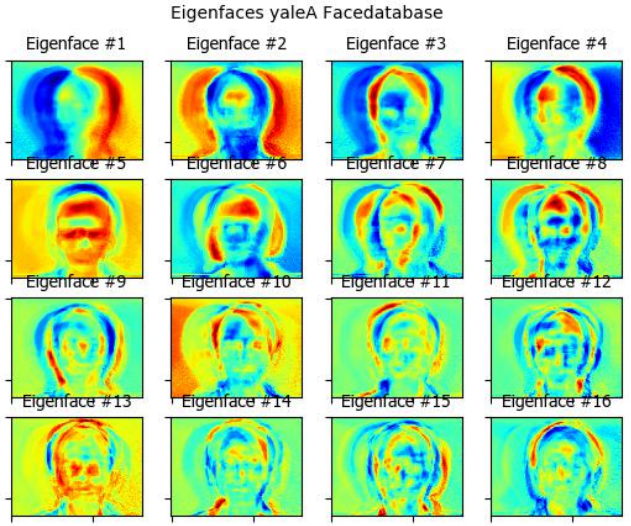
$$X^T X v_i = \lambda_i v_i \rightarrow X X^T (X v_i) = \lambda_i (X v_i)?$$

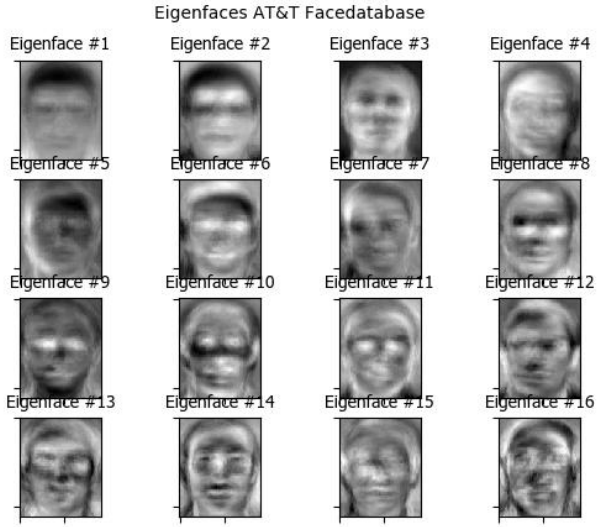


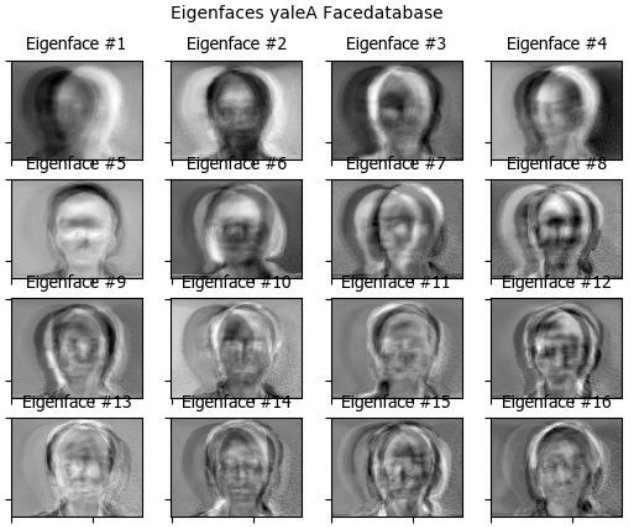
笔者使用前三类人脸画出

Experiments: Eigenfaces

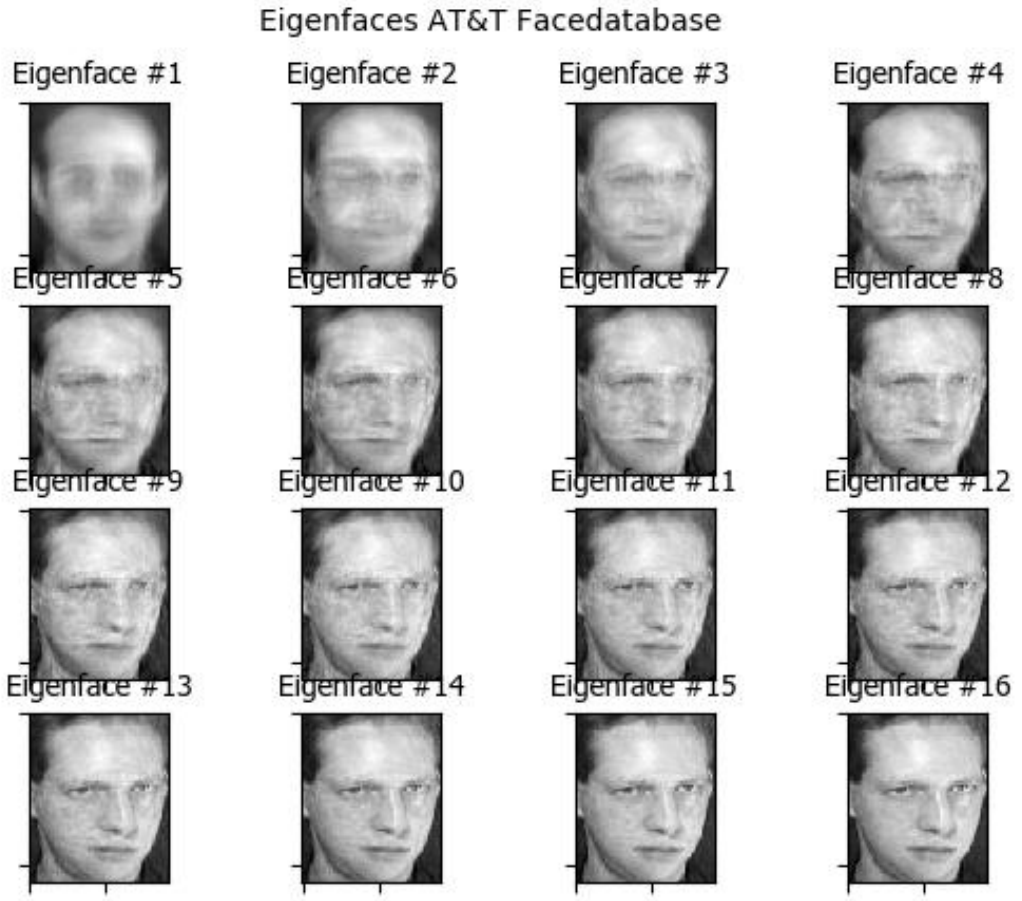




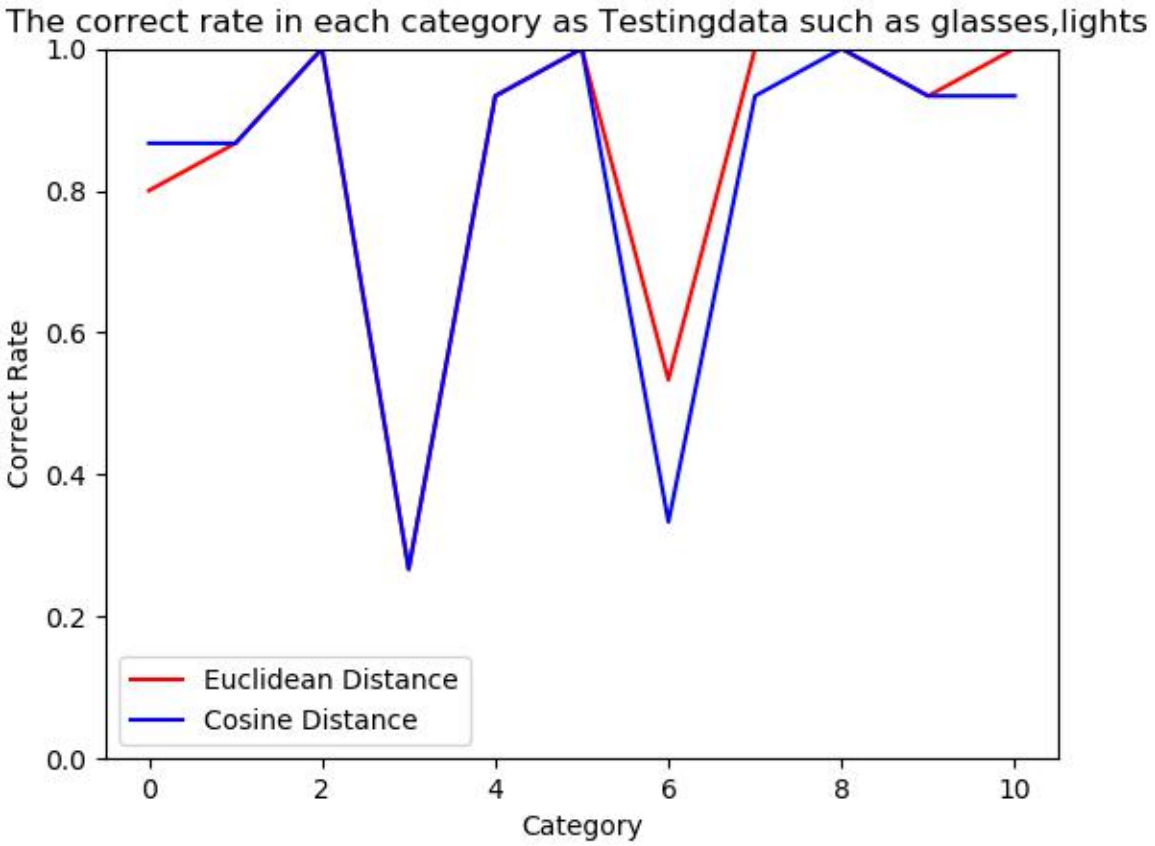


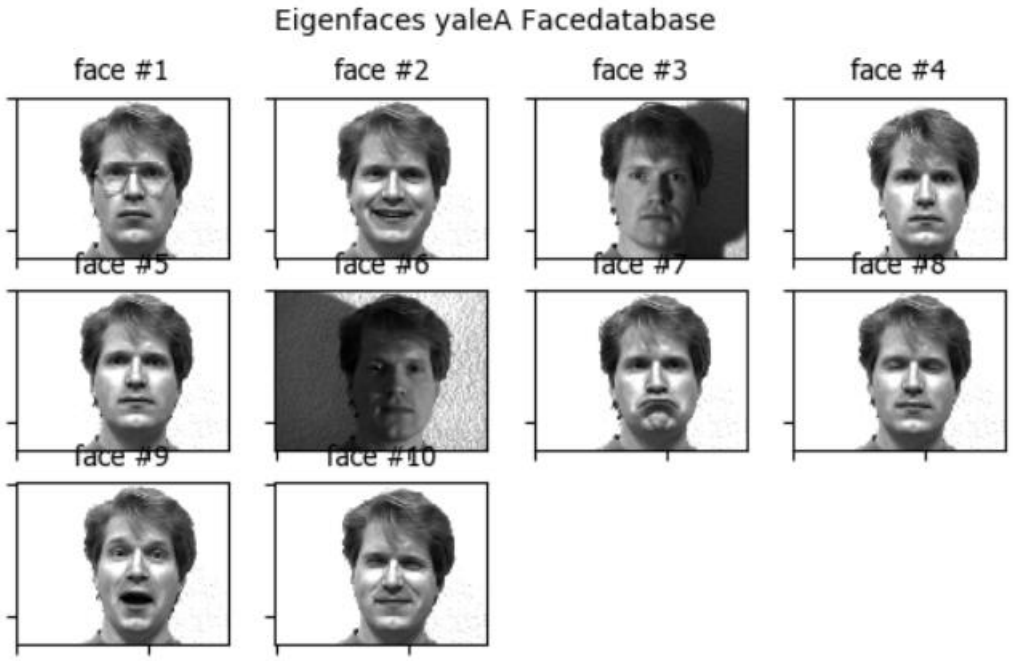


Experiments: Reconstruction

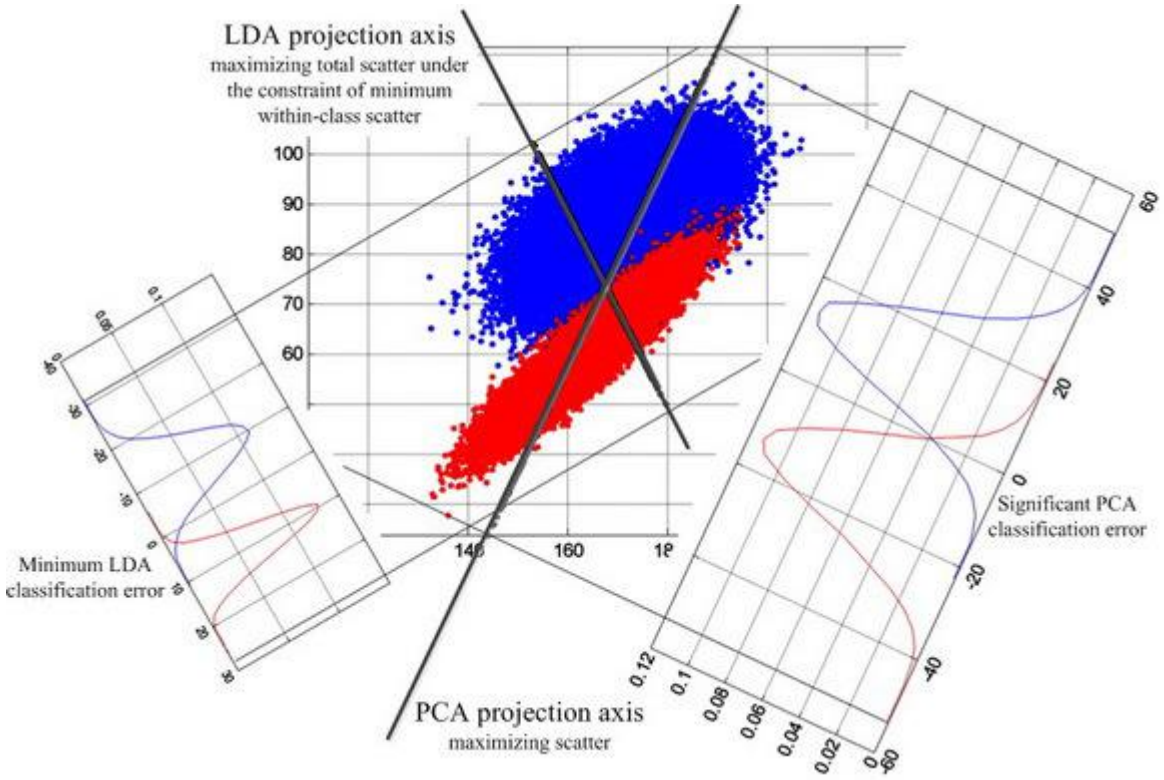


Experiments: Accuracy





Fisherface



https://hsto.org/files/c7a/3ee/740/c7a3ee7409aa41489452b7418eef5805.jpg

Fisherface

Different lighting direction, different facial expression affect the judgement of images.

Algorithmic Description

$$S_B = \sum_{i=1}^{c} N_i (\mu_i - \mu)(\mu_i - \mu)^T$$

$$S_W = \sum_{i=1}^{c} \sum_{\mathbf{x}_k \in X_i} (\mathbf{x}_k - \mu_i)(\mathbf{x}_k - \mu_i)^T$$

$$W_{opt} = \arg\max_W \frac{\left| W^T S_B W \right|}{\left| W^T S_W W \right|}$$

$$= \left[\mathbf{w}_1 \quad \mathbf{w}_2 \quad \dots \quad \mathbf{w}_m \right]$$

Idealized!

Algorithmic Description

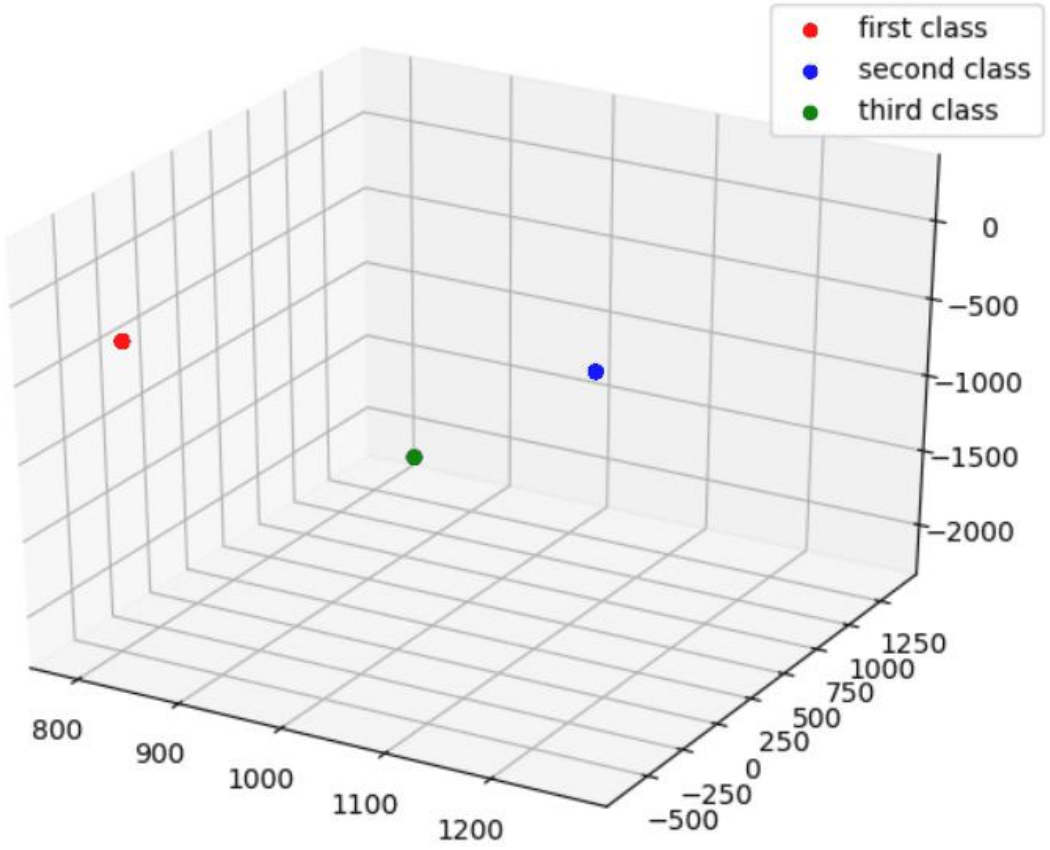
$$W_{opt}^T = W_{fld}^T W_{pca}^T$$

$$W_{pca} = \arg\max_W \left| W^T S_T W \right|$$

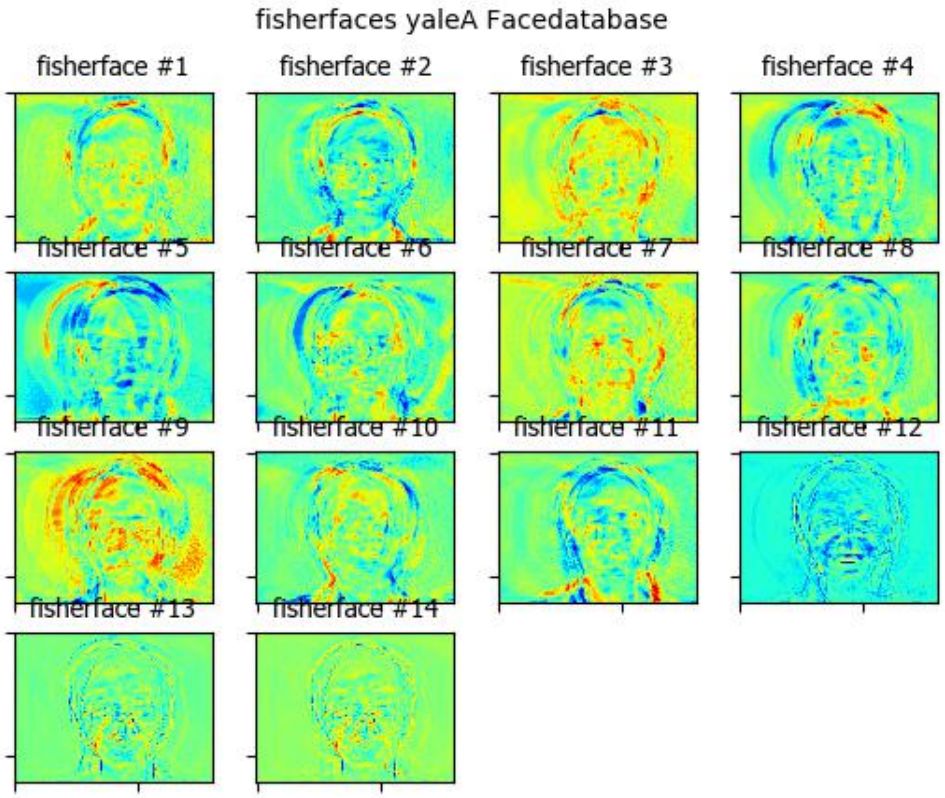
$$W_{fld} = \arg\max_W \frac{\left| W^T W_{pca}^T S_B W_{pca} W \right|}{\left| W^T W_{pca}^T S_W W_{pca} W \right|}$$

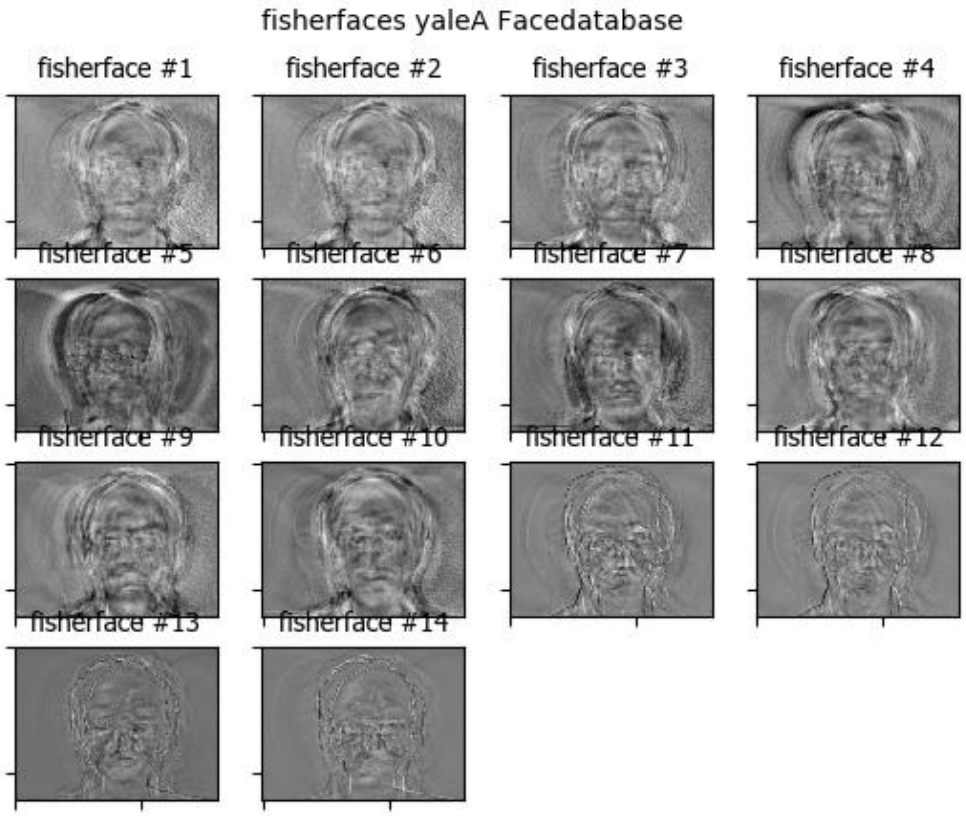
Eigenfaces method for face recognition by:

1. Projecting all training samples into the Fisherface subspace.
2. Projecting the query image into the Fisherface subspace.
3. Finding the nearest neighbor between the projected training images and the projected query image.

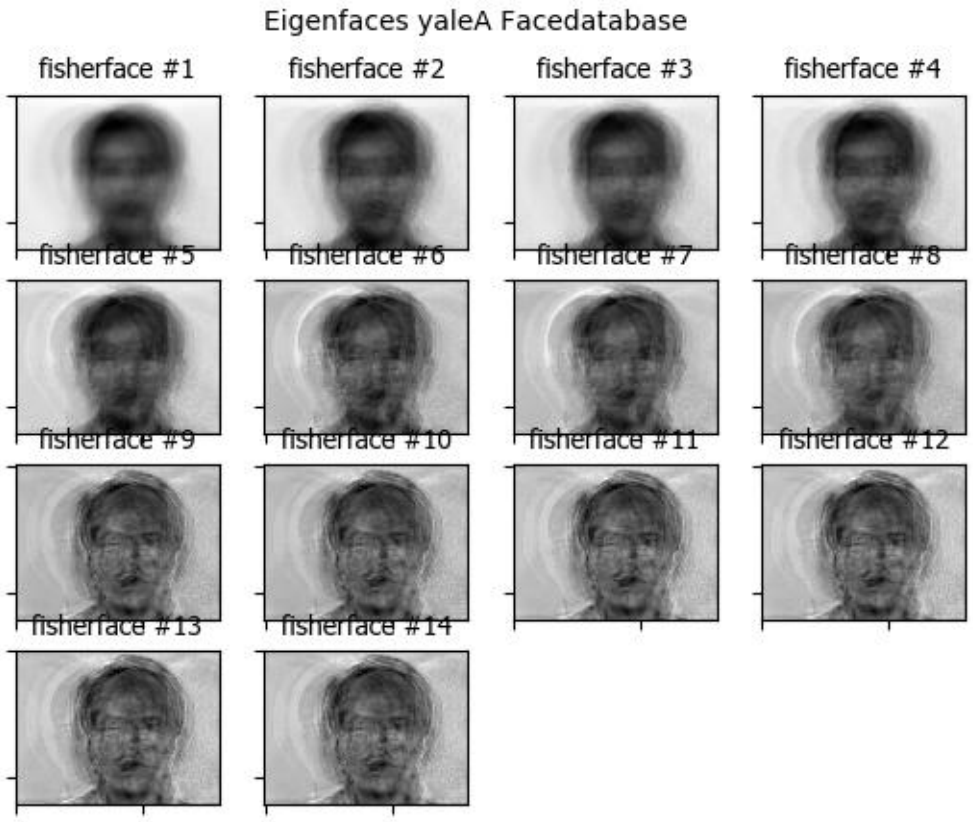


Experiments: Fisherfaces

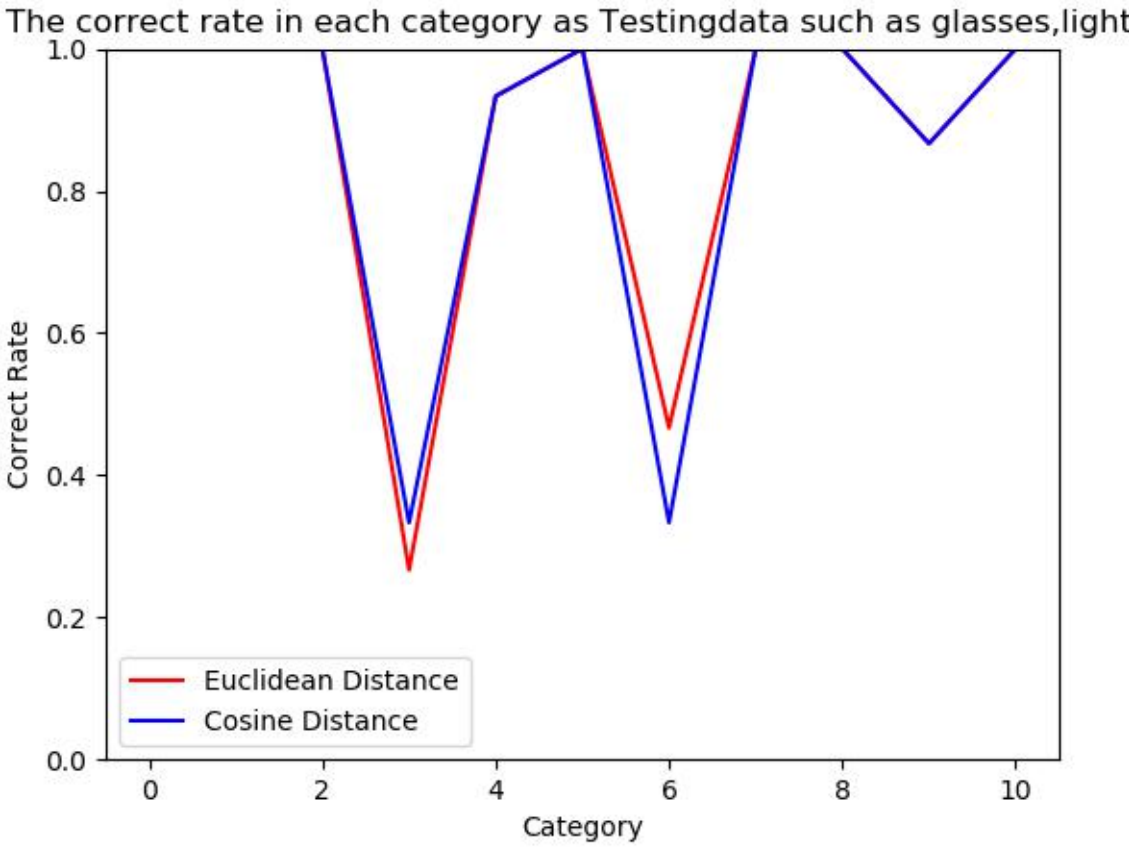


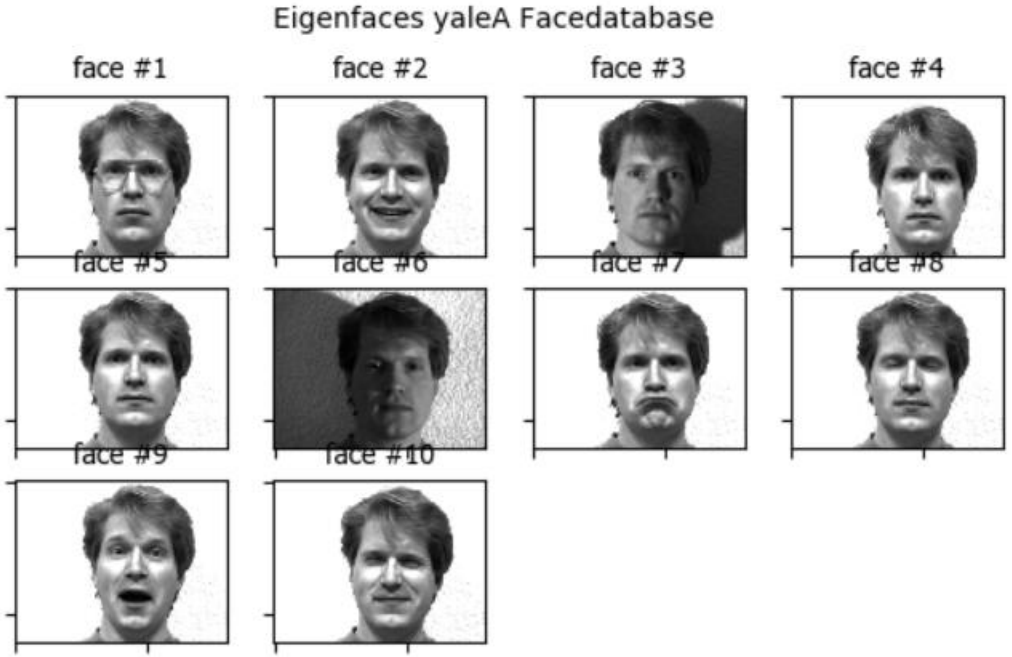


Experiments: Reconstruction



Experiments: Accuracy





Can variation in lighting conditions be accommodated if some of the individuals are only observed under one lighting condition?

Peter N. Belhumeur, Jo~ao P. Hespanha, and David J.Kriegman

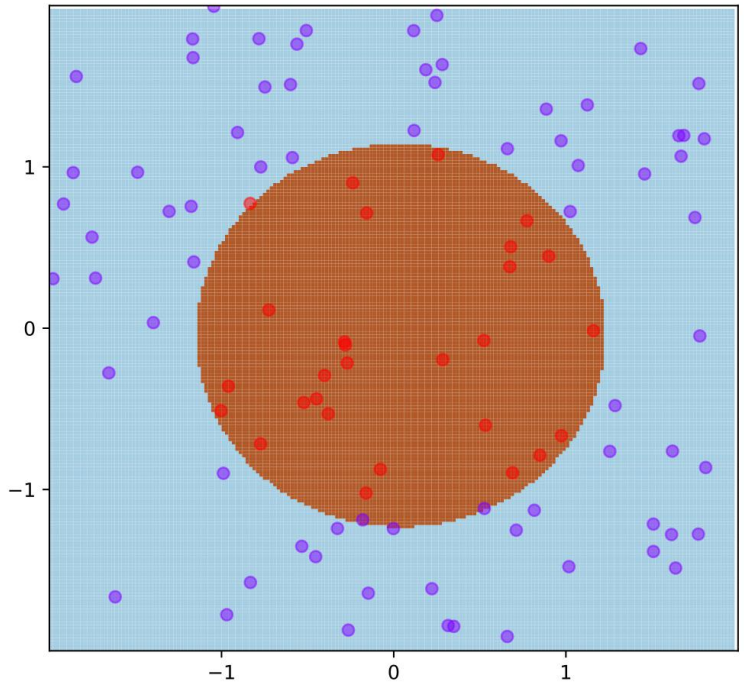
PCA Algorithmic Implement

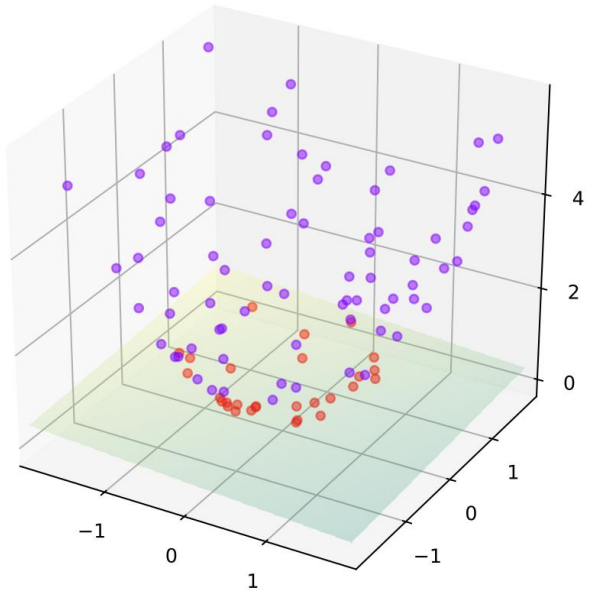
```
def pca(X, y, num_components=0):
    [n, d] = X.shape
    if (num_components <= 0) or (num_components > n):
        num_components = n
    mu = X.mean(axis=0)
    X = X - mu
    if n > d:
        C = np.dot(X.T, X)
        [eigenvalues, eigenvectors] = np.linalg.eigh(C)
    else:
        C = np.dot(X, X.T)
        [eigenvalues, eigenvectors] = np.linalg.eigh(C)
        eigenvectors = np.dot(X.T, eigenvectors)
        for i in range(n):
            eigenvectors[:,i] = eigenvectors[:, i] / np.linalg.norm(eigenvectors[:, i])
    idx = np.argsort(-eigenvalues)
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:, idx]
    eigenvalues = eigenvalues[0:num_components].copy()
    eigenvectors = eigenvectors[:,0:num_components].copy()
    return [eigenvalues, eigenvectors, mu]
```

LDA Algorithmic Implement

```
def lda(X, y, num_components=0):
    y = np.asarray(y)
    [n,d] = X.shape
    c = np.unique(y)
    if(num_components <= 0) or (num_components > (len(c) - 1)):
        num_components = (len(c) - 1)
    mean_total = X.mean(axis=0)
    Sw = np.zeros((d, d), dtype=np.dtype(np.float32))
    Sb = np.zeros((d, d), dtype=np.dtype(np.float32))
    for i in c:
        Xi = X[np.where(y == i)[0], :]
        mean_class = Xi.mean(axis=0)
        Sw = Sw + np.dot((Xi - mean_class).T, (Xi-mean_class))
        Sb = Sb + n * np.dot((mean_class - mean_total).T, (mean_class - mean_total))
    eigenvalues, eigenvectors = np.linalg.eig(np.linalg.inv(Sw) * Sb)
    idx = np.argsort(-eigenvalues.real)
    eigenvalues, eigenvectors = eigenvalues[idx], eigenvectors[:, idx]
    eigenvalues = np.array(eigenvalues[0: num_components].real, dtype=np.dtype(np.float32), copy=True)
    eigenvectors = np.array(eigenvectors[0:, 0:num_components].real, dtype=np.dtype(np.float32), copy=True)
    return [eigenvalues, eigenvectors]
```

Improvement



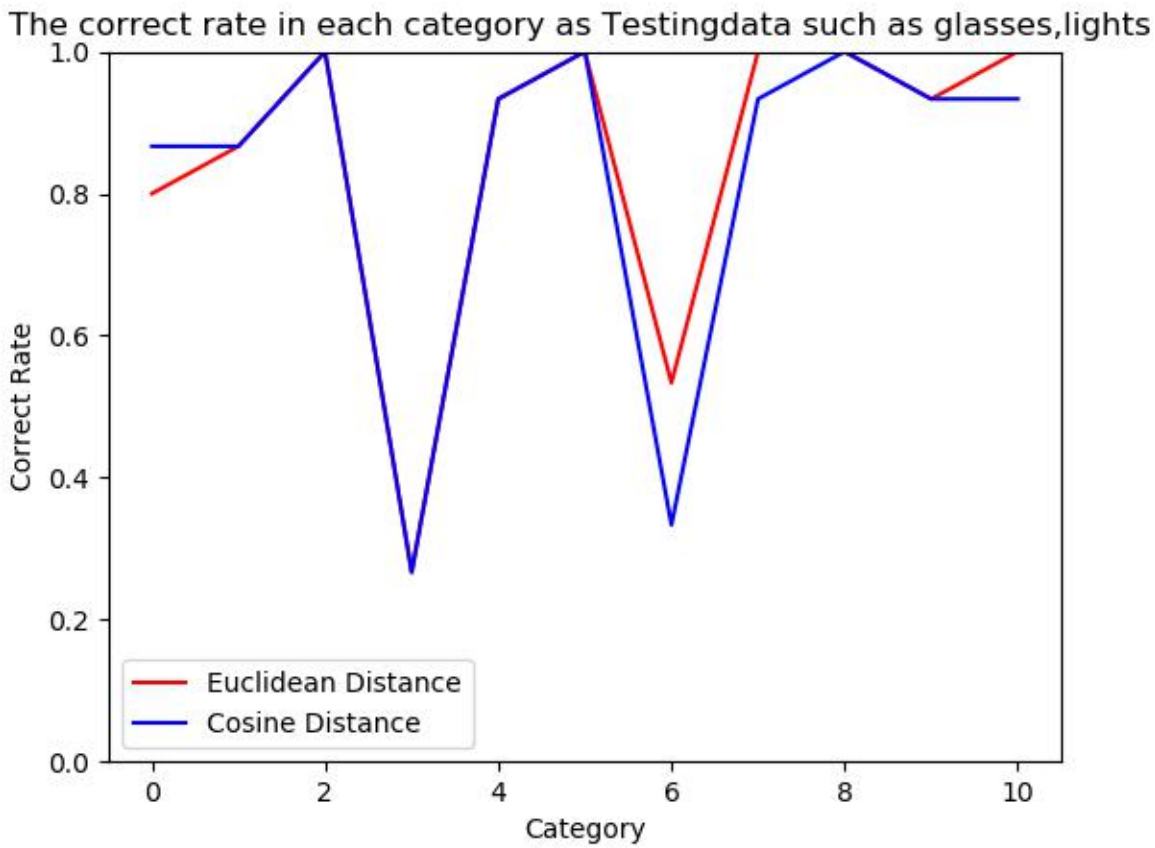


https://en.wikipedia.org/wiki/Kernel_method

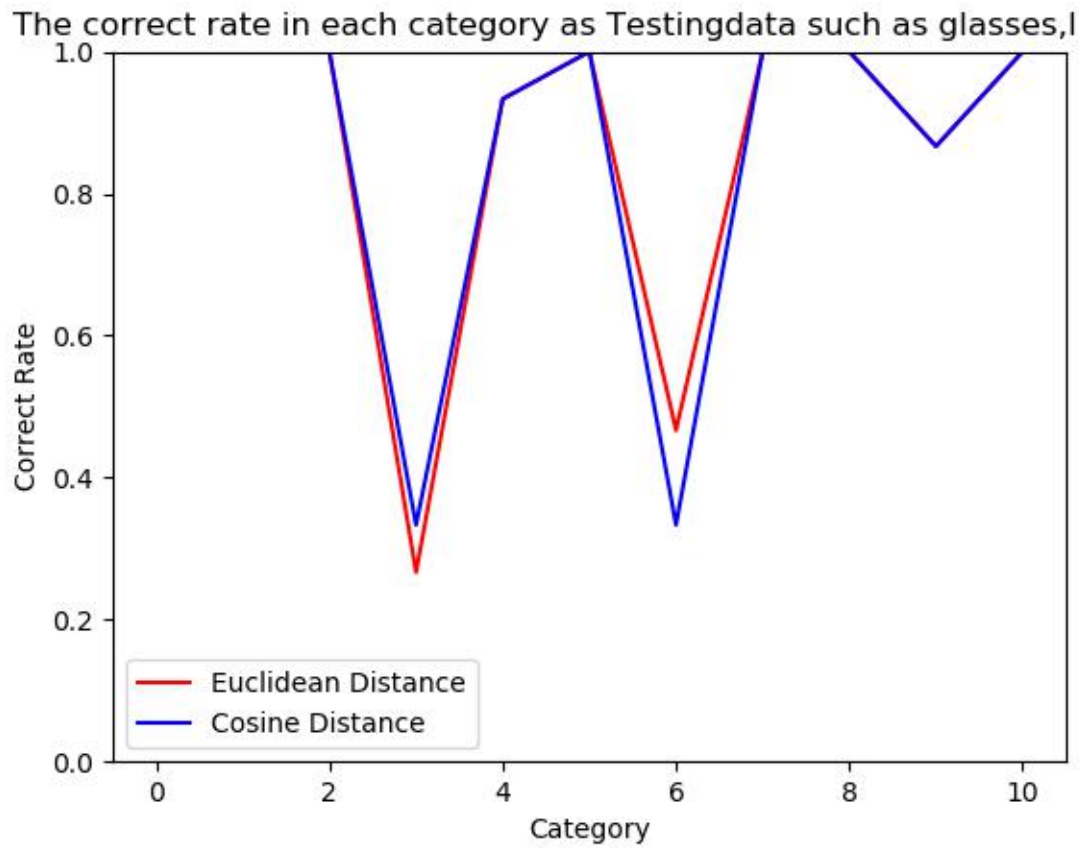
Distance Function

We want to see the difference of Euclidean Distance and C

Eigenfaces

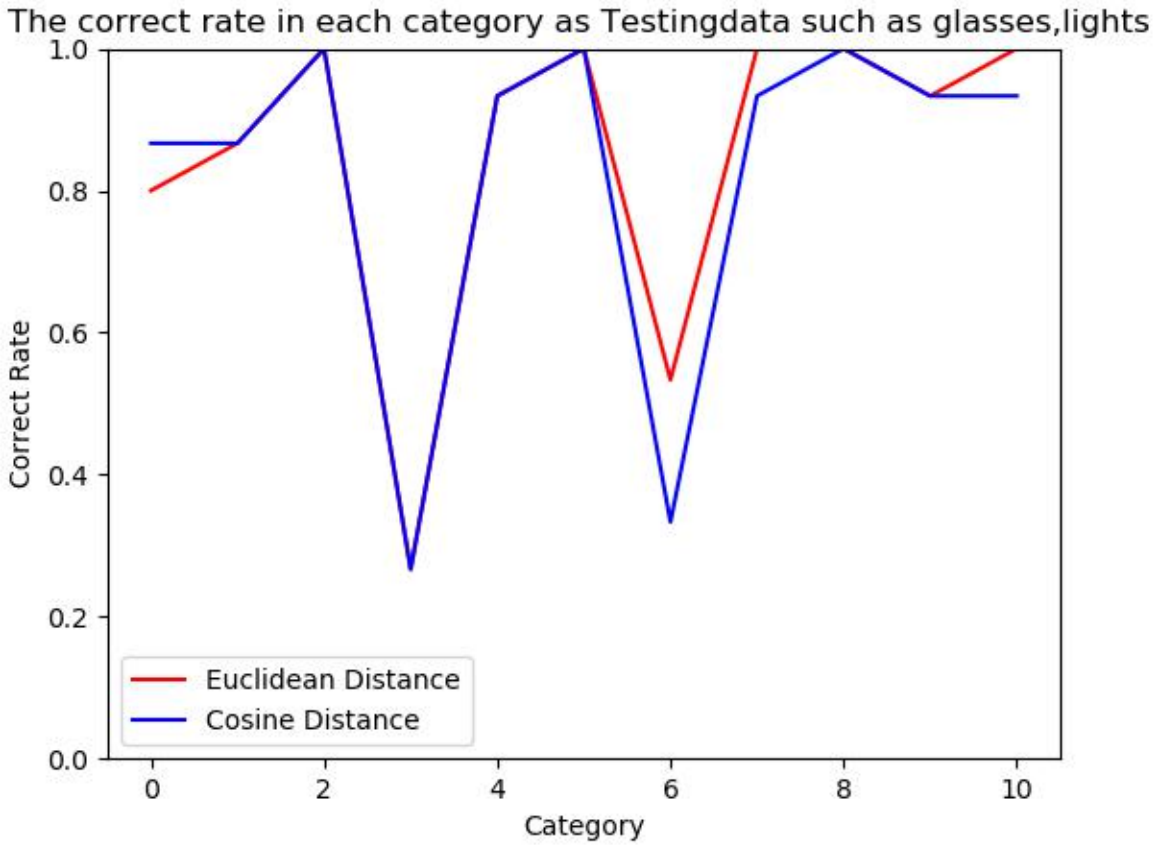


Fisherfaces

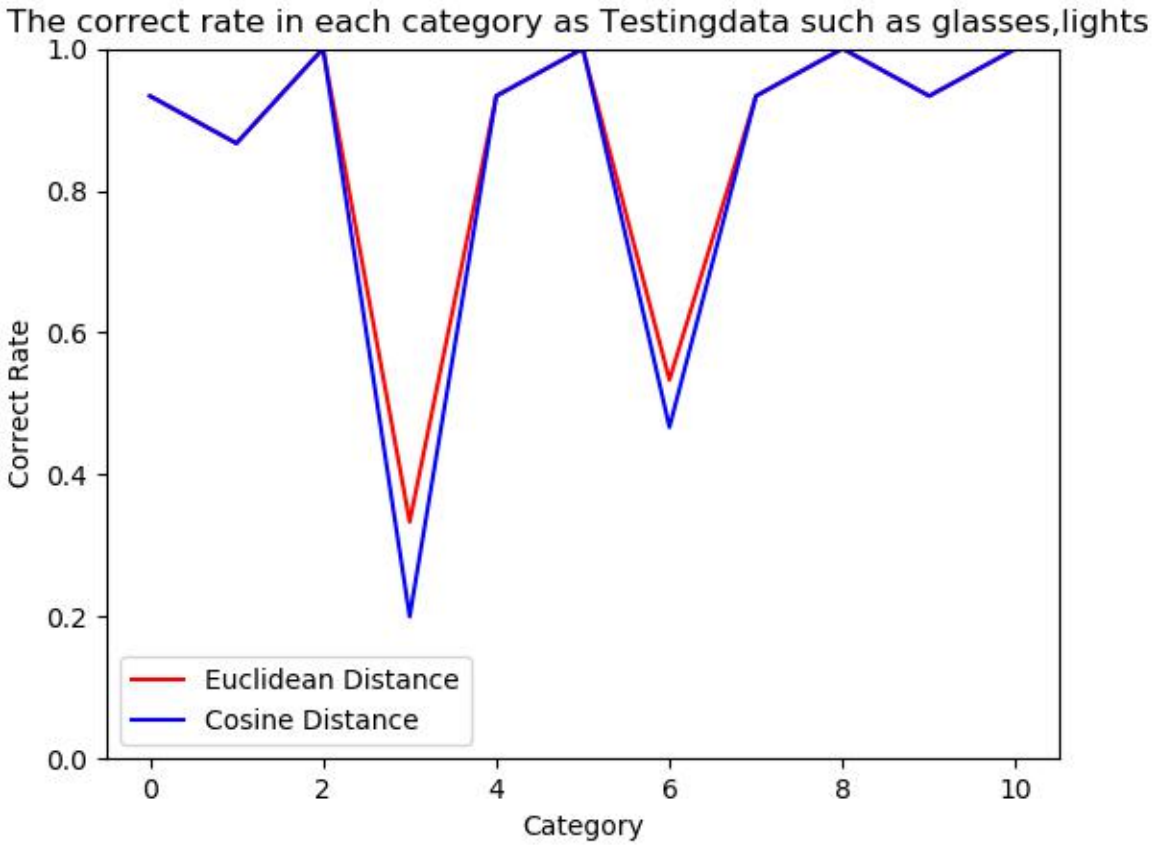


Cut off one vector: Eigenface

All Eigenvectors



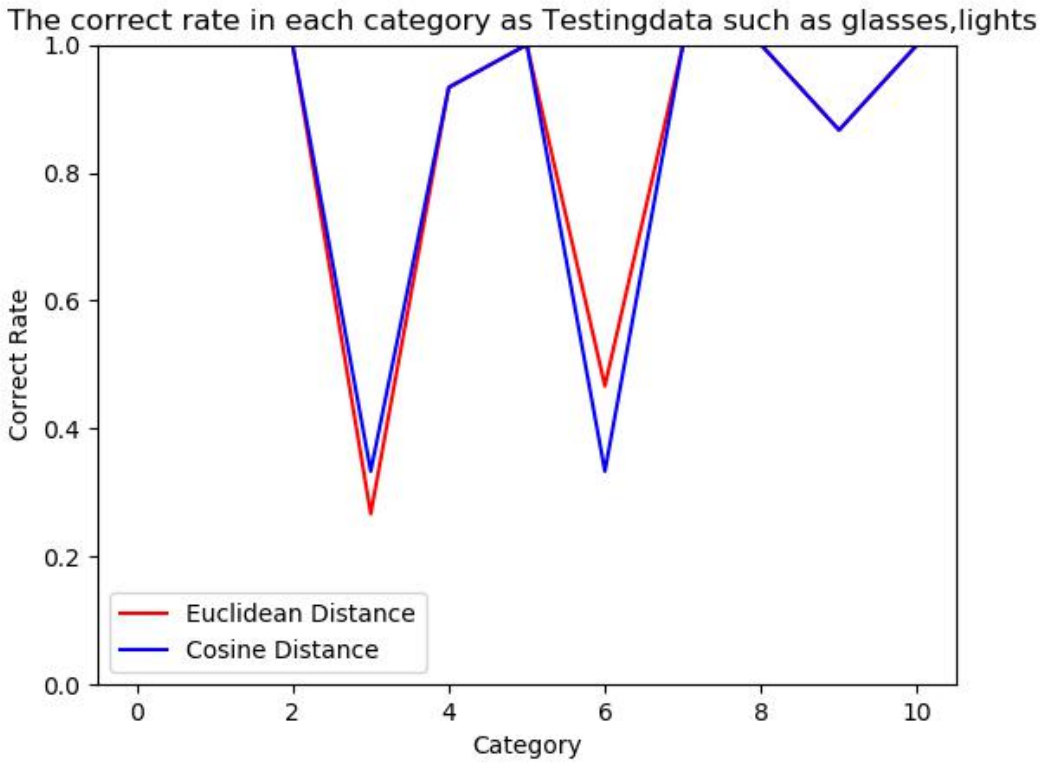
Cut off the first vector



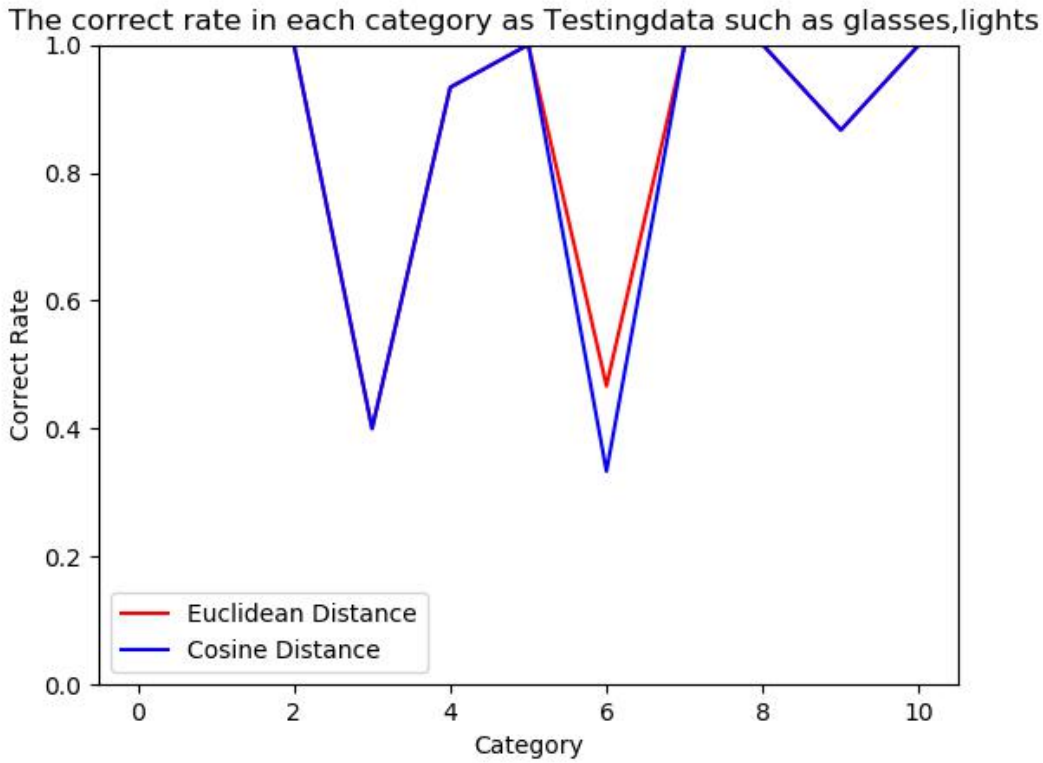
We can see that the accuracies in the third and sixth categories are improved which coincide with illumination.

Cut off one vector: Fisherface

All Fishervectors

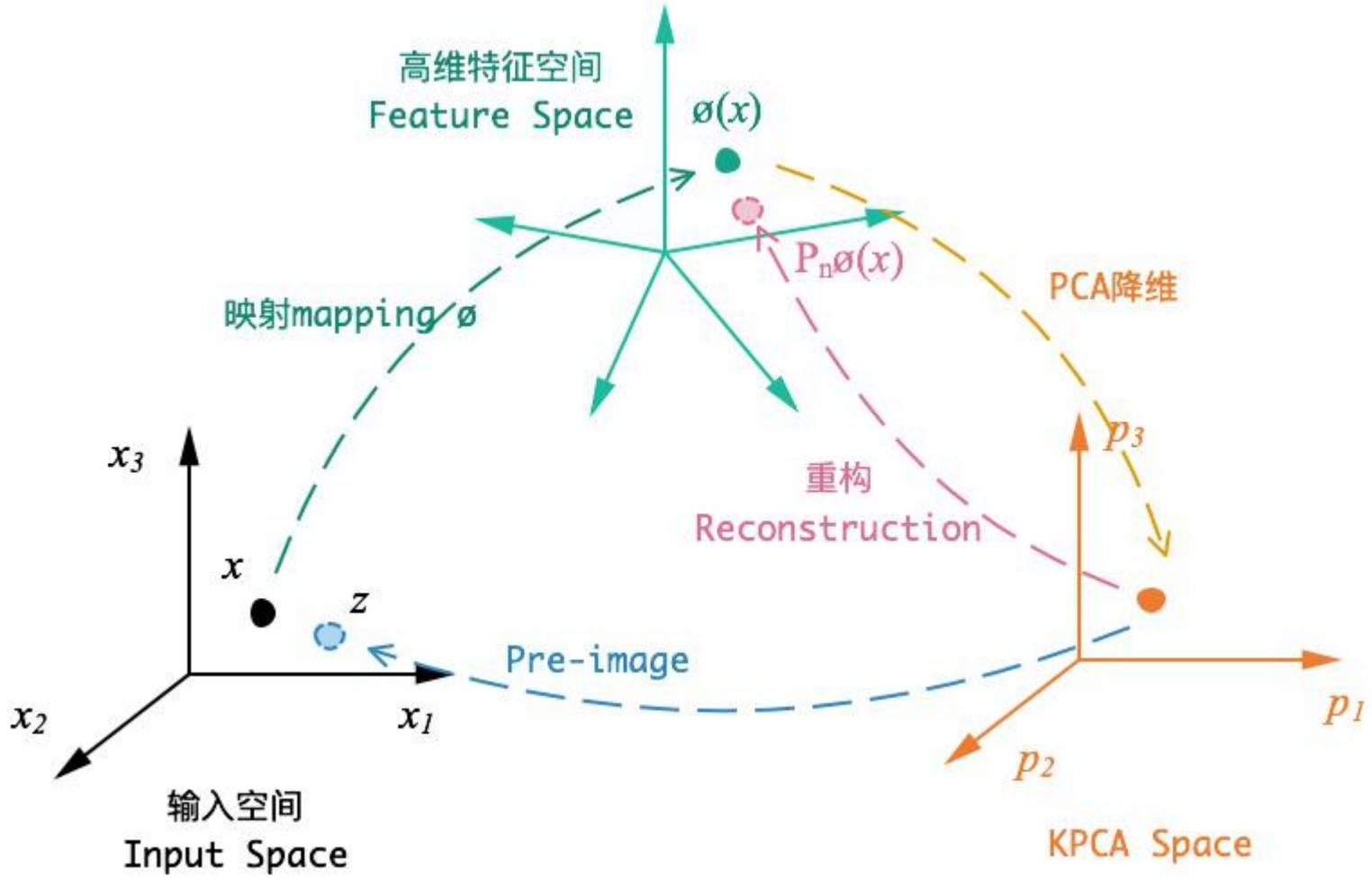


Cut off the first vector



We can see that the accuracies in the third categories are improved which coincide with illumination.

Kernel method: Kernel PCA



https://zhuanlan.zhihu.com/p/59775730

Kernel method: Kernel PCA

$$\phi(\mathbf{X}) = [\phi(\mathbf{x}_1), \phi(\mathbf{x}_2), \ldots, \phi(\mathbf{x}_N)]$$

$$\mathbf{C}_{\mathcal{F}} = \frac{1}{N}\phi(\mathbf{X})[\phi(\mathbf{X})]^T = \frac{1}{N}\sum_{i=1}^{N}\phi(\mathbf{x_i})\,\phi(\mathbf{x_i})^T$$

$$\mathbf{C}_{\mathcal{F}}\mathbf{p} = \lambda\mathbf{p}$$

$$\phi(\mathbf{X})[\phi(\mathbf{X})]^T\mathbf{p} = \sum_{i=1}^{N}\phi(\mathbf{x_i})\,\phi(\mathbf{x_i})^T\,\mathbf{p} = \lambda\mathbf{p}$$

$$\mathbf{p} = \frac{1}{\lambda}\sum_{i=1}^{N}\left(\phi(\mathbf{x}_i)\left[\phi(\mathbf{x}_i)^T\,\mathbf{p}\right]\right)$$

$$\mathbf{p} = \sum_{i=1}^{N}\alpha_i\phi(\mathbf{x_i}) = \phi(\mathbf{X})\alpha$$

$$[\phi(\mathbf{X})]^T\phi(\mathbf{X})[\phi(\mathbf{X})]^T\phi(\mathbf{X})\alpha = \lambda[\phi(\mathbf{X})]^T\phi(\mathbf{X})\alpha$$

$$\mathbf{K} = [\phi(\mathbf{X})]^T\phi(\mathbf{X})$$

$$\mathbf{K}\alpha = \lambda\alpha$$

主成分 $$\mathbf{t_a} = \phi(\mathbf{X})^T\mathbf{p_a} = \phi(\mathbf{X})^T\phi(\mathbf{X})\alpha_\mathbf{a} = \mathbf{K}\alpha_\mathbf{a} = \lambda_a\alpha_\mathbf{a}$$

中心化

$$\widetilde{\mathbf{K}} = \mathbf{K} - \mathbf{K}\cdot\mathbf{1_N} - \mathbf{1_N}\cdot\mathbf{K} + \mathbf{1_N}\cdot\mathbf{K}\cdot\mathbf{1_N}$$

$$\widetilde{\mathbf{K}}_{\text{test}} = \widetilde{\phi}(\mathbf{X}_{\text{test}})^T\,\widetilde{\phi}(\mathbf{X}) = \mathbf{K}_{\text{test}} - \mathbf{K}_{\text{test}}\cdot\mathbf{1_N} - \mathbf{1}_{\text{NL}}^T\cdot\mathbf{K} + \mathbf{1}_{\text{NL}}^T\cdot\mathbf{K}\cdot\mathbf{1_N}$$

Kernel method: Kernel PCA

```python
def CalculateK(X,Y):
    '''
    calculate Kernel Matrix
    :param X: Input Data
    :param Y: Other Data
    :return: kernel Matrix
    '''
    K = np.zeros((X.shape[1], Y.shape[1]), dtype=np.dtype(np.float32))
    for i in range(X.shape[1]): #xi,xj
        for j in range(Y.shape[1]):
            K[i,j] = kernelfunc(X[:,i],Y[:,j])
    return np.matrix(K)

def Training_NormalizeK(K):
    '''
    归一化训练数据的kernel矩阵
    :param K: kernel matrix for training data
    :return: normalization result
    '''
    OneNOne = np.ones(K.shape[0],dtype=np.dtype(np.float32))
    OneN = np.matrix(OneNOne).T.dot(np.matrix(OneNOne))/K.shape[0]
    return K - K.dot(OneN) - OneN.dot(K)+OneN.dot(K).dot(OneN)

def Testing_NormalizeK(K_test,K_training):
    OneNOne = np.ones(K_test.shape[1], dtype=np.dtype(np.float32))
    OneLOne = np.ones(K_test.shape[0], dtype=np.dtype(np.float32))
    OneNL = np.matrix(OneNOne).T.dot(np.matrix(OneLOne)) / K_test.shape[1]
    OneN = np.matrix(OneNOne).T.dot(np.matrix(OneNOne)) / K_test.shape[1]
    return K_test - K_test*OneN - OneNL.T*K_training + OneNL.T*K_training*OneN
```

```python
def Kernelpca(X,tX,num_components = 0):
    '''
    Calculate the project matrix
    :param X: Training Data
    :param tX: Testing Data
    :return: W,W_test
    '''
    if num_components == 0 :
        num_components = X.shape[1]-1
    K = CalculateK(X,X)
    K = Training_NormalizeK(K)
    [eigenvalues, eigenvectors] = np.linalg.eigh(K)
    idx = np.argsort(-eigenvalues)
    eigenvalues = eigenvalues[idx]
    for i in range(len(eigenvalues)-1,0,-1):
        if eigenvalues[i]>0:
            num_components = min(num_components,i)
            break
    print(num_components)
    eigenvectors = eigenvectors[:, idx]
    eigenvalues = eigenvalues[0:num_components].copy()
    eigenvectors = eigenvectors[:, 0:num_components].copy()
    eigenvectors = eigenvectors/np.sqrt(eigenvalues)
    W = np.array(eigenvalues)*np.array(eigenvectors) #weight matrix
    W = np.matrix(W)
    K_test = CalculateK(tX,X)
    K_test = Testing_NormalizeK(K_test,K)
    W_test = K_test * np.matrix(eigenvectors)
    return [W,W_test]
```

Kernel method: Kernel PCA

Unfortunately, the effect is not satisfactory.

```
expected =  0 / predicted =  4
expected =  1 / predicted =  4
expected =  2 / predicted =  10
expected =  3 / predicted =  9
expected =  4 / predicted =  4
expected =  5 / predicted =  5
expected =  6 / predicted =  3
expected =  7 / predicted =  7
expected =  8 / predicted =  1
expected =  9 / predicted =  4
expected =  10 / predicted =  4
expected =  11 / predicted =  5
expected =  12 / predicted =  4
expected =  13 / predicted =  4
expected =  14 / predicted =  12
correct rate: 0.2
```

We did a lot of testing, but the best result is this. We are very disappointed.

```
return 1/(1+pow(np.linalg.norm(xi-xj,ord = 2),0.5)/0.1)#1/(1+pow((xi-xj).T.dot(xi-xj),1)/1000)#pow((xi-xj).T.dot(xi-xj)+10,0.5)#1/(1+pow(np.linalg.norm(xi-xj,ord = 2),0.5)/0.1)#-1*math.log(1+pow(np.linalg.norm(xi-xj,ord = 2),1.5))#tanh(0.00001*xi.T.dot(xj))#math.exp(-1*np.linalg.norm(xi-xj,ord=1))#pow(np.dot(xi.T,xj)*0.001+5,20)#math.exp(-1*pow(np.linalg.norm(xi-xj,ord=2),2)*10000)
```

But we insisted finishing the Kernel Fisher Discriminant Analysis.

Cauchy Kernel: $k(x,y) = \frac{1}{\|x-y\|^2/\sigma+1}$

Kernel method: Kernel Fisher Discriminant Analysis

Multi-class KFD [edit]

The extension to cases where there are more than two classes is relatively straightforward.[2][6][7] Let c be the number of classes. Then multi-class KFD involves projecting the data into a $(c-1)$-dimensional space using $(c-1)$ discriminant functions

$$y_i = \mathbf{w}_i^{\mathrm{T}} \phi(\mathbf{x}) \qquad i = 1, \ldots, c-1.$$

This can be written in matrix notation

$$\mathbf{y} = \mathbf{W}^{\mathrm{T}} \phi(\mathbf{x}),$$

where the $\mathbf{w}_i$ are the columns of $\mathbf{W}$.[6] Further, the between-class covariance matrix is now

$$\mathbf{S}_B^{\phi} = \sum_{i=1}^{c} l_i (\mathbf{m}_i^{\phi} - \mathbf{m}^{\phi})(\mathbf{m}_i^{\phi} - \mathbf{m}^{\phi})^{\mathrm{T}},$$

where $\mathbf{m}^{\phi}$ is the mean of all the data in the new feature space. The within-class covariance matrix is

$$\mathbf{S}_W^{\phi} = \sum_{i=1}^{c} \sum_{n=1}^{l_i} (\phi(\mathbf{x}_n^i) - \mathbf{m}_i^{\phi})(\phi(\mathbf{x}_n^i) - \mathbf{m}_i^{\phi})^{\mathrm{T}},$$

The solution is now obtained by maximizing

$$J(\mathbf{W}) = \frac{\left|\mathbf{W}^{\mathrm{T}} \mathbf{S}_B^{\phi} \mathbf{W}\right|}{\left|\mathbf{W}^{\mathrm{T}} \mathbf{S}_W^{\phi} \mathbf{W}\right|}.$$

The kernel trick can again be used and the goal of multi-class KFD becomes[7]

$$\mathbf{A}^* = \underset{\mathbf{A}}{\operatorname{argmax}} = \frac{\left|\mathbf{A}^{\mathrm{T}} \mathbf{M} \mathbf{A}\right|}{\left|\mathbf{A}^{\mathrm{T}} \mathbf{N} \mathbf{A}\right|},$$

where $A = [\alpha_1, \ldots, \alpha_{c-1}]$ and

$$M = \sum_{j=1}^{c} l_j (\mathbf{M}_j - \mathbf{M}_*)(\mathbf{M}_j - \mathbf{M}_*)^{\mathrm{T}}$$

$$N = \sum_{j=1}^{c} \mathbf{K}_j (\mathbf{I} - \mathbf{1}_{l_j}) \mathbf{K}_j^{\mathrm{T}}.$$

The $\mathbf{M}_i$ are defined as in the above section and $\mathbf{M}_*$ is defined as

$$(\mathbf{M}_*)_j = \frac{1}{l} \sum_{k=1}^{l} k(\mathbf{x}_j, \mathbf{x}_k).$$

$\mathbf{A}^*$ can then be computed by finding the $(c-1)$ leading eigenvectors of $\mathbf{N}^{-1}\mathbf{M}$.[7] Furthermore, the projection of a new input, $\mathbf{x}_t$, is given by[7]

$$\mathbf{y}(\mathbf{x}_t) = (\mathbf{A}^*)^{\mathrm{T}} \mathbf{K}_t,$$

where the i^{th} component of $\mathbf{K}_t$ is given by $k(\mathbf{x}_i, \mathbf{x}_t)$.

https://en.wikipedia.org/wiki/Kernel_Fisher_discriminant_analysis

Kernel method: Kernel Fisher Discriminant Analysis

```python
def kernellda(X,y,num_components=0):
    y = np.asarray(y)
    [n,d] = X.shape
    c = np.unique(y)
    if (num_components <= 0) or (num_components > (len(c) - 1)):
        num_components = (len(c) - 1)
    M = np.zeros((n,1),dtype=np.dtype(np.float32))
    Mi = np.zeros((n,1),dtype=np.dtype(np.float32))
    Pk = np.zeros((n, 1), dtype=np.dtype(np.float32))
    Sw = np.zeros((n, n), dtype=np.dtype(np.float32))
    Sb = np.zeros((n, n), dtype=np.dtype(np.float32))
    for i in c:
        Xi = X[np.where(y==i)[0],:]
        for j in range(n):
            Mi[j,0] = 0
            M[j,0] = 0
            for k in range(Xi.shape[0]):
                Mi[j,0] += kernelfunc(X[j,:],Xi[k,:])
            for k in range(n):
                M[j,0] += kernelfunc(X[j,:],X[k,:])
            Mi[j,0] /= Xi.shape[0]
            M[j,0] /= n
        Sb += Xi.shape[0]*np.dot((Mi - M), (Mi - M).T)
        for k in range(Xi.shape[0]):
            for j in range(n):
                Pk[j,0] = kernelfunc(X[j,:],Xi[k,:])
            Sw += np.dot((Pk-Mi),(Pk-Mi).T)
    eigenvalues, eigenvectors = np.linalg.eig(np.linalg.inv(Sw) * Sb)
    idx = np.argsort(-eigenvalues.real)
    eigenvalues, eigenvectors = eigenvalues[idx], eigenvectors[:, idx]
    eigenvalues = np.array(eigenvalues[0: num_components].real, dtype=np.dtype(np.float32), copy=True)
    eigenvectors = np.array(eigenvectors[0:, 0:num_components].real, dtype=np.dtype(np.float32), copy=True)
    return [eigenvalues, eigenvectors]
```

```python
def kernelfisher_faces(X,y,num_components=0):
    y = np.asarray(y)
    [n, d] = X.shape
    c = len(np.unique(y))
    [eigenvalues_pca, eigenvectors_pca, mu_pca] = pca(X, y, (n - c))
    [eigenvalues_lda, eigenvectors_lda] = kernellda(project(eigenvectors_pca, X, mu_pca), y, num_components)
    #eigenvectors = np.dot(eigenvectors_pca, eigenvectors_lda)
    return [eigenvalues_lda, eigenvectors_pca,eigenvectors_lda, mu_pca]
```

```python
class KernelfisherfacesModel(BaseModel):
    def __init__(self, X=None, y=None, dist_metric=EuclideanDistance(), num_components=0):
        super(KernelfisherfacesModel, self).__init__(X=X, y=y, dist_metric=dist_metric, num_components=num_components)

    def compute(self, X, y):
        [D, self.W_Pca,self.W, self.mu] = kernelfisher_faces(asRowMatrix(X), y, self.num_components)
        self.y = y
        self.new_x = project(self.W_Pca, asRowMatrix(X), self.mu)
        new_x = self.new_x
        for xi in new_x:
            Xj = np.zeros((new_x.shape[0],1),dtype=np.dtype(np.float32))
            for k in range(new_x.shape[0]):
                Xj[k,0] = kernelfunc(xi,new_x[k,:])
            self.projections.append(project(self.W, Xj.T))

    def predict(self, X):
        minDist = np.finfo('float').max
        minClass = -1
        Q = project(self.W_Pca, X.reshape(1, -1), self.mu)
        Xj = np.zeros((self.new_x.shape[0], 1), dtype=np.dtype(np.float32))
        for k in range(self.new_x.shape[0]):
            Xj[k, 0] = kernelfunc(Q.T, self.new_x[k, :])
        Q = project(self.W,Xj.T)
        for i in range(len(self.projections)):
            dist = self.dist_metric(self.projections[i], Q)
            if dist < minDist:
                minDist = dist
                minClass = self.y[i]
        return minClass
```

Kernel method: Kernel Fisher Discriminant Analysis

Also, the effect is not satisfactory.

```
expected =  0 / predicted =  3
expected =  1 / predicted =  9
expected =  2 / predicted =  2
expected =  3 / predicted =  3
expected =  4 / predicted =  8
expected =  5 / predicted =  5
expected =  6 / predicted =  12
expected =  7 / predicted =  3
expected =  8 / predicted =  3
expected =  9 / predicted =  3
expected =  10 / predicted =  10
expected =  11 / predicted =  6
expected =  12 / predicted =  13
expected =  13 / predicted =  13
expected =  14 / predicted =  8
correct rate =  0.3333333333333333
```

Polynomial kernel: $k(x, y) = (ax^t y + c)^d$

In the beginning, we thought that the Gaussian kernel will work well but the fact is that the predicted is all the same. Maybe our parameters are worse.

```
expected =  0 / predicted =  14
expected =  1 / predicted =  14
expected =  2 / predicted =  14
expected =  3 / predicted =  14
expected =  4 / predicted =  14
expected =  5 / predicted =  14
expected =  6 / predicted =  14
expected =  7 / predicted =  14
expected =  8 / predicted =  14
expected =  9 / predicted =  14
expected =  10 / predicted =  14
expected =  11 / predicted =  14
expected =  12 / predicted =  14
expected =  13 / predicted =  14
expected =  14 / predicted =  14
correct rate: 0.06666666666666667
```

Future

1. 3D EIGENFACES
2. CNN
3. …