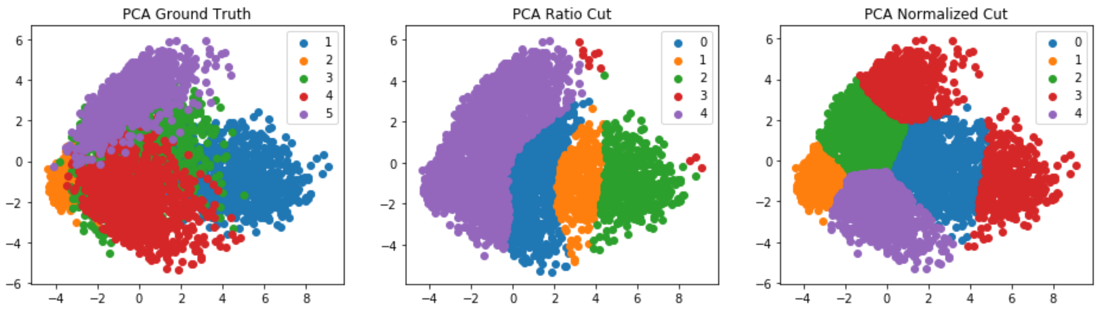
Homework# 7

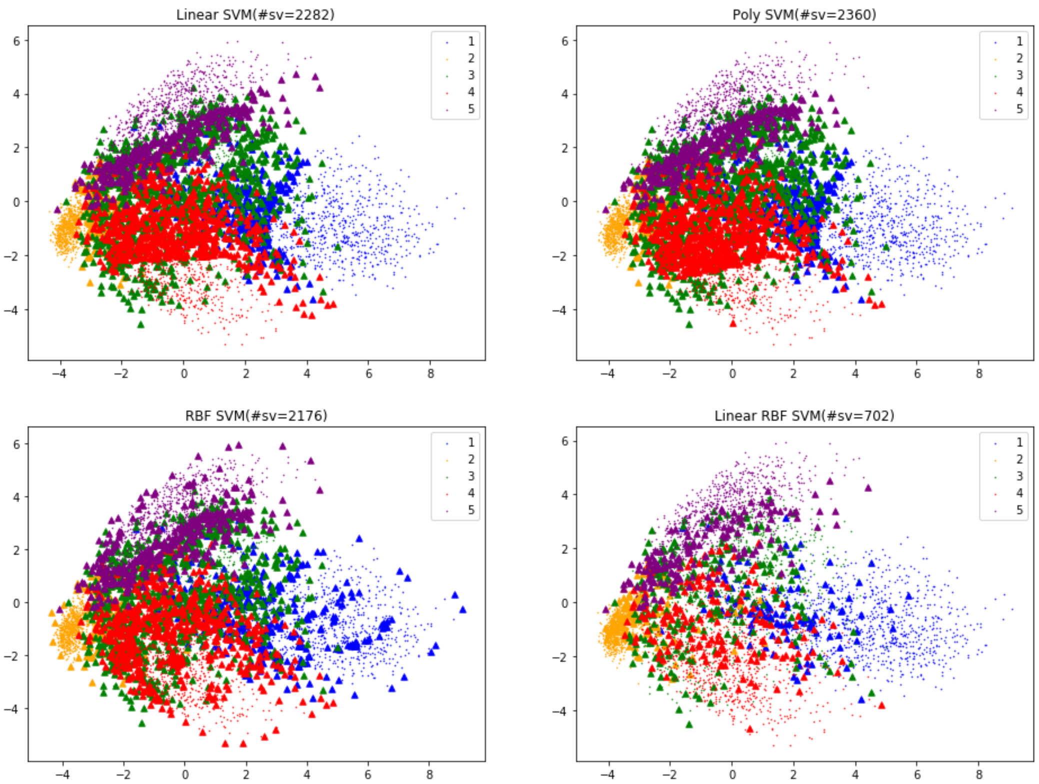
0756079 陳冠聞

1. PCA projection and clustering



可以發現 normalized cut 不同 cluster 間的個數較為平均。

2. PCA projection and SVM classification

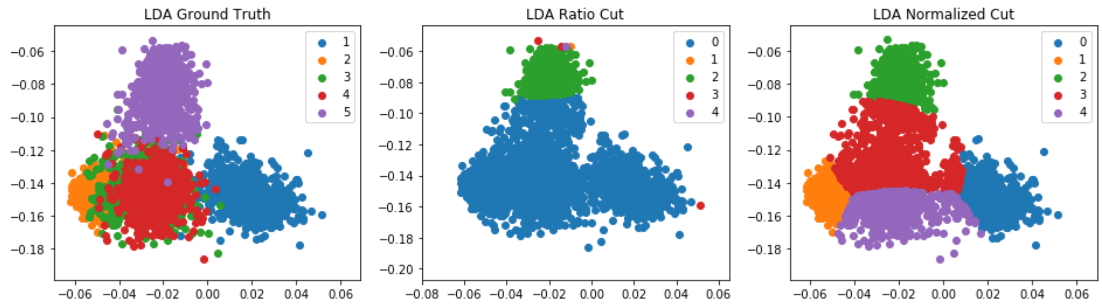


可以發現使用 Linear + RBF kernel 所用的 support vector 最少 (support vector 越少代表 overfitting 的風險越小)

此處 kernel 參數採用 default

(C=0, degree=3 for poly, coef0=0.0, gamma = 1/784)

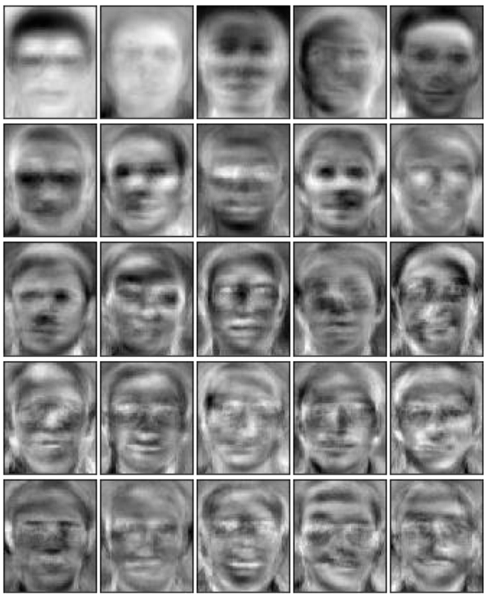
3. LDA projection



同上，可以發現 normalized cut 不同 cluster 間的個數較為平均。

4. EigenFace

Top 25 EigenFaces



Reconstruction

可以看到使用越多的 eigenFace 來重建，會越像本來的臉。

Code Detail

0. Read Data

```
df_X = pd.read_csv('X_train.csv', header=None)
df_Y = pd.read_csv('T_train.csv', header=None)

X_train = df_X.values
Y_train = df_Y.values
Y_train = Y_train.reshape(Y_train.shape[0])
```

1. Use PCA to project all your data X_train.csv onto 2D space

```
class PCA:
    def __init__(self, n_components=2):
        self.n_components = n_components

    def transform(self, X):
        X_high = np.copy(X)
        mean_mat = np.tile(self.mean_vec, (X.shape[0],1))
        diff_mat = X_high - mean_mat
        # Project from high to low
        X_low = np.matmul(diff_mat, self.W)
        return np.real(X_low)

    def fit(self, X):
        X_high = np.copy(X)
        mean_vec = np.mean(X_high, 0)
        mean_mat = np.tile(mean_vec, (X.shape[0],1))
        diff_mat = X_high - mean_mat
        cov_mat = np.cov(diff_mat.T)
        self.mean_vec = mean_vec

        # Compute eigenpairs of cov mat
        eigenValues, eigenVectors = np.linalg.eig(cov_mat)
        idx = eigenValues.argsort()[::-1]
        W = eigenVectors[:,idx][:, :self.n_components]
        W = W * -1
        self.W = W
        return self
pca = PCA(n_components=2)
X_low_pca = pca.fit(X_train).transform(X_train)
```

Compute covariance matrix

Compute first-k eigenvector of covariance matrix

2. Use LDA to project all your data X_train.csv onto 2D space

```python
class LDA:
    def __init__(self, n_components=2):
        self.n_components = n_components
        self.mean = 0
        self.std = 1

    def transform(self, X):
        X_high = np.copy(X)
        X_high = (X_high - self.mean) / self.std
        # Project from high to low
        X_low = np.matmul(X_high, self.W)
        return np.real(X_low)

    def fit(self, X, Y):
        N, dim = X.shape
        X_high = np.copy(X)
        self.mean = X_high.mean()
        self.std = X_high.std()
        X_high = (X_high - self.mean) / self.std
        # Compute mean for each class (mj, nj)
        mean_vectors = []
        for c in set(Y):
            mean_vectors.append( np.mean(X_high[Y==c], axis=0) )
        self.mean_vectors = mean_vectors

        # Compute within-class scatter
        SW = np.zeros( (dim,dim) )
        for c, mv in zip(set(Y), mean_vectors):
            within_class_scattter = np.zeros((dim, dim))
            for xi in X_high[Y==c]:
                xi = xi.reshape(-1, 1) # make vec to mat
                mj = mv.reshape(-1, 1) # make vec to mat
                within_class_scattter += np.matmul(xi-mj, (xi-mj).T)
            SW += within_class_scattter
        # Compute between-class scatter
        SB = np.zeros( (dim,dim) )
        m = np.mean(X_high, axis=0).reshape(-1, 1)
        for c, mv in zip(set(Y), mean_vectors):
            nj = X_high[Y==c].shape[0]
            mj = mv.reshape(-1, 1) # make vec to mat
            SB += nj * np.matmul((mj-m), (mj-m).T)

        # Compute W using first k eigenvetor of inv(SW)*SB
        mat = np.dot(np.linalg.pinv(SW), SB)
        eigenValues, eigenVectors = np.linalg.eig(mat)
        idx = eigenValues.argsort()[::-1]
        eigenValues = eigenValues[idx]
        eigenVectors = eigenVectors[:,idx]
        W = np.real(eigenVectors[:, 0:self.n_components])
        W /= np.linalg.norm(W, axis=0)
        self.W = W
        return self
lda = LDA(n_components=2)
X_low_lda = lda.fit(X_train, Y_train).transform(X_train)
```

Compute Within-class scatter

Compute Within-class scatter

Compute first-k largest eigvector of inv(SW) * SB

3. RatioCut

```python
class Spectral_clustering_rationCut:

    def __init__(self, n_cluster=2, gamma=1):
        self.n_cluster = n_cluster
        self.gamma = gamma

    def fit(self, X, kernel='rbf'):
        # 0. Define similarity matrix W and D
        W = []

        if kernel == 'linear':
            W = compute_linear_kernel(X)
        elif kernel == 'rbf':
            W = compute_RBF_kernel(X, self.gamma)
        elif kernel == 'rbf_linear':
            W = kernel = compute_linear_rbf_kernel(X, self.gamma)

        D = np.zeros((X.shape[0], X.shape[0]))
        for d in range(X.shape[0]):
            D[d][d] = W[d].sum()

        # 1. Graph Laplacian L = D - W
        L = D - W

        # Get first k eigenvector
        eigenValues, eigenVectors = LA.eig(L)
        idx = eigenValues.argsort()
        eigenValues = eigenValues[idx]
        eigenVectors = eigenVectors[:,idx]
        U = eigenVectors[:, 1:self.n_cluster+1]

        # Do k-means
        kmeans = k_means_clustering(k=self.n_cluster)
        membership = kmeans.fit(U)
        return membership
```

4. Normalized Cut

```python
class Spectral_clustering_normCut:

    def __init__(self, n_cluster=2, gamma=1):
        self.n_cluster = n_cluster
        self.gamma = gamma

    def fit(self, X, kernel='rbf'):
        # 0. Define similarity matrix W and D
        W = []

        if kernel == 'linear':
            W = compute_linear_kernel(X)
        elif kernel == 'rbf':
            W = compute_RBF_kernel(X, self.gamma)
        elif kernel== 'rbf_linear':
            W = kernel = compute_linear_rbf_kernel(X, self.gamma)
        D = np.zeros((X.shape[0], X.shape[0]))
        for d in range(X.shape[0]):
            D[d][d] = W[d].sum()

        # 1. Graph Laplacian L = D - W, L_norm = D^(-1/2) L D^(-1/2)
        L = D - W
        D_inv_sqrt = np.linalg.pinv(sqrtm(D))
        L = np.matmul(np.matmul(D_inv_sqrt, L), D_inv_sqrt)

        # Get first k eigenvector
        eigenValues, eigenVectors = LA.eig(L)
        idx = eigenValues.argsort()
        eigenValues = eigenValues[idx]
        eigenVectors = eigenVectors[:,idx]
        U = eigenVectors[:, 1:self.n_cluster+1]

        # Do k-means
        kmeans = k_means_clustering(k=self.n_cluster)
        membership = kmeans.fit(U)
        return membership
```

5. SVM

```python
from sklearn import svm
```

```python
svm_linear = svm.SVC(kernel='linear')
svm_linear.fit(X_low_pca, Y_train)
y_pred_svm_linear = svm_linear.predict(X_low_pca)
sv_linear_index = svm_linear.support_
print(svm_linear.n_support_)
```

Linear SVM

```python
svm_poly = svm.SVC(kernel='poly')
svm_poly.fit(X_low_pca, Y_train)
y_pred_poly_rbf = svm_poly.predict(X_low_pca)
sv_poly_index = svm_poly.support_
print(svm_poly.n_support_)
```

Poly SVM

```python
svm_rbf = svm.SVC(kernel='rbf')
svm_rbf.fit(X_low_pca, Y_train)
y_pred_svm_rbf = svm_rbf.predict(X_low_pca)
sv_rbf_index = svm_rbf.support_
print(svm_rbf.n_support_)
```

RBF SVM

```python
svm_linear_rbf = svm.SVC(kernel='precomputed')
kernel = compute_linear_rbf_kernel(X_train, gamma=1/784)
svm_linear_rbf.fit(kernel, Y_train)
# y_pred_svm_rbf = svm_linear_rbf.predict(X_low_pca)
sv_linear_rbf_index = svm_linear_rbf.support_
print(svm_linear_rbf.n_support_)
```

Linear + RBF SVM

6. EigenFaces

Read Images

```
imgs = []
for dir_path in img_dirs:
    imgs += read_dir_imgs(dir_path)

imgs = np.array(imgs)
imgs = imgs.reshape( (prod(imgs.shape[:1]), prod(imgs.shape[1:])) ).T
```

Compute Mean Face
And covariance matrix

```
mean_vector = imgs.mean(1)
mean_face = vec2img(mean_vector)
plt.imshow(mean_face, cmap = 'gray'), plt.show()

diff_imgs = imgs - np.tile(np.array([mean_vector]).T, (1, 400))
T_trans_T = np.cov(imgs.T)
```

Get 25 largest eigenvector
And project mean face to
eigenspace to get 25 eigen
faces

```
eigenValues, eigenVectors = np.linalg.eig(T_trans_T)

idx = eigenValues.argsort()[::-1]
eigenValues = eigenValues[idx]
eigenVectors = eigenVectors[:,idx]
print(eigenVectors.shape)

k = 25
eigenValues = eigenValues[:k]
eigenVectors = eigenVectors[:, 0:k]

# get real eigen vector(eigen faces)
A = diff_imgs
eigenVectors = np.matmul(A, eigenVectors)
eigen_faces = np.copy(eigenVectors)
```

For more detail, please check EigenFace.py and HW7.py