# hw7

## A. Code

### Kernel Eigenfaces

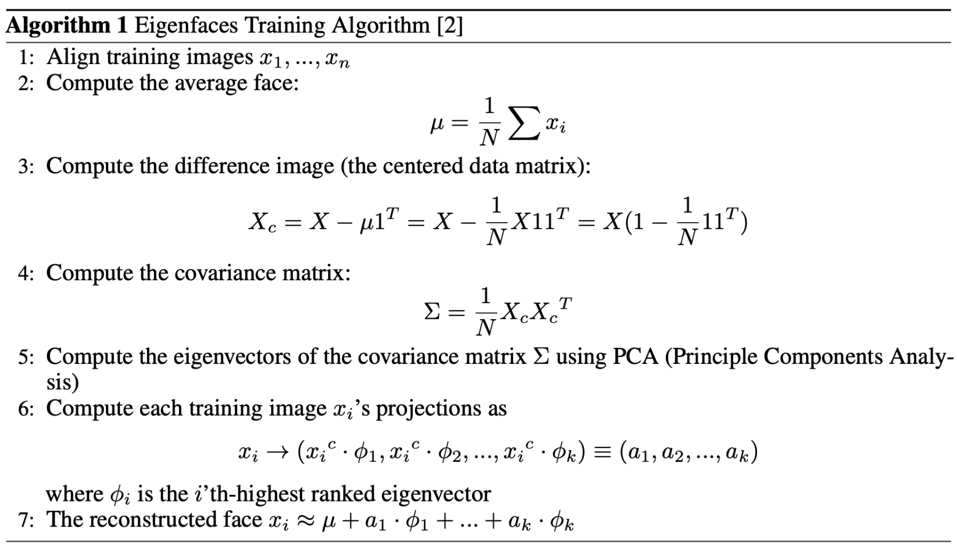
#### Preprocess

- The images' size are reduces to 100 x 100. Otherwise my RAM will be run out QQ.

```
(rw, rh) = (100, 100)
img_size = rw * rh

def read_faces(dir):
    ...
    for i, fname in enumerate(files):
        im = Image.open(f'{dir}/{fname}')
        # convert to gray scale
        im = im.convert("L")
        # use ANTIALIAS to avoid losing quality in resize step
        im = im.resize((rh, rw), Image.ANTIALIAS)
        ...
    return faces, labels
```

#### Part1

- PCA
  - reference



```
Algorithm 1 Eigenfaces Training Algorithm [2]
1: Align training images $x_1, ..., x_n$
2: Compute the average face:
       $$\mu = \frac{1}{N}\sum x_i$$
3: Compute the difference image (the centered data matrix):
       $$X_c = X - \mu 1^T = X - \frac{1}{N}X11^T = X(1 - \frac{1}{N}11^T)$$
4: Compute the covariance matrix:
       $$\Sigma = \frac{1}{N}X_c{X_c}^T$$
5: Compute the eigenvectors of the covariance matrix $\Sigma$ using PCA (Principle Components Analysis)
6: Compute each training image $x_i$'s projections as
       $$x_i \rightarrow ({x_i}^c \cdot \phi_1, {x_i}^c \cdot \phi_2, ..., {x_i}^c \cdot \phi_k) \equiv (a_1, a_2, ..., a_k)$$
   where $\phi_i$ is the $i$'th-highest ranked eigenvector
7: The reconstructed face $x_i \approx \mu + a_1 \cdot \phi_1 + ... + a_k \cdot \phi_k$
```

  - show the first 25 eigenfaces

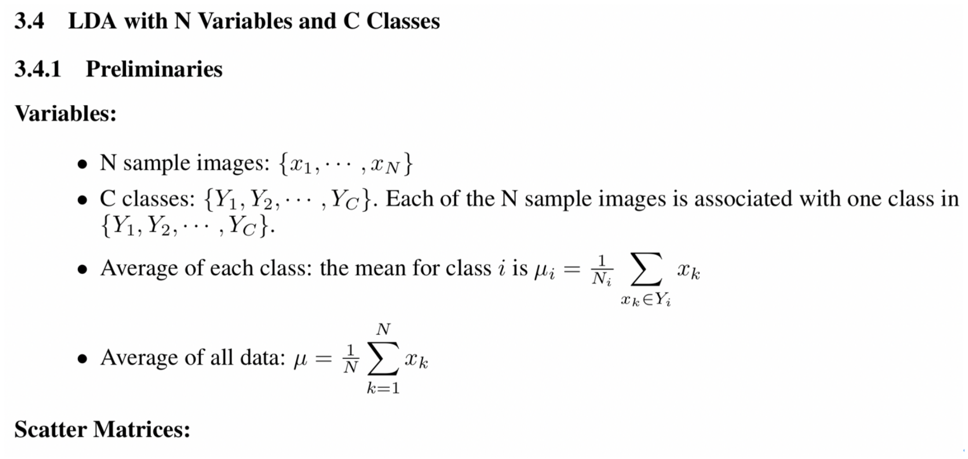
```
def PCA(X, X_test, train_labels, test_labels):
    # step2
    mu = np.mean(X, axis=0)

    # step 3
    # X_mu.shape = (img_size, num)
    X_mu = (X - mu).T

    # step 4
    C = X_mu @ X_mu.T / len(X)

    # step 5
    eig_vals, eig_vecs = getEigen(C, method="PCA", loadMode=False,
sym=True)
    eif_vecs = eig_vecs / np.linalg.norm(eig_vecs, axis=0)

    # faces contains eigen vectors [u1, u2, ... u25]
    sort_idx = np.argsort(-eig_vals)
    U = eig_vecs[:, sort_idx[:25]]
    faces = U.T.reshape(25, rh, rw)
    show_faces(faces, 5, 5, "PCA : 25 eigenfaces")
```

- randomly pick 10 images to show theit reconstruction

```
def PCA(X, X_test, train_labels, test_labels):
    ...
    idx = np.random.choice(len(X), 10)
    for k in [25]:
        print(f'k = {k}')
        # U : select k eigen vectors
        U = eig_vecs[:, sort_idx[:k]]

        # step 6
        # W : project X to U, (10, k)
        W = (X[idx] - mu) @ U

        # step 7
        # reconstruct_faces's = μ + Σwiui
        reconstruct_faces = (mu + W @ U.T).reshape(10, rh, rw)
        show_faces(reconstruct_faces, 2, 5, f'PCA : 10 reconstruction
with {k}')
```

---

- LDA
    - reference



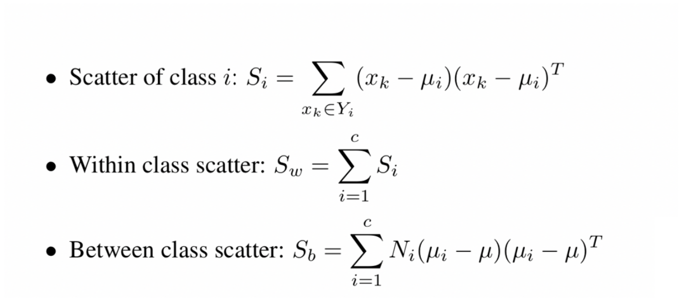
### 3.4 LDA with N Variables and C Classes

#### 3.4.1 Preliminaries

**Variables:**

- N sample images: $\{x_1, \cdots, x_N\}$
- C classes: $\{Y_1, Y_2, \cdots, Y_C\}$. Each of the N sample images is associated with one class in $\{Y_1, Y_2, \cdots, Y_C\}$.
- Average of each class: the mean for class $i$ is $\mu_i = \frac{1}{N_i} \sum\limits_{x_k \in Y_i} x_k$
- Average of all data: $\mu = \frac{1}{N} \sum\limits_{k=1}^{N} x_k$

**Scatter Matrices:**



- Scatter of class $i$: $S_i = \sum\limits_{x_k \in Y_i} (x_k - \mu_i)(x_k - \mu_i)^T$
- Within class scatter: $S_w = \sum\limits_{i=1}^{c} S_i$
- Between class scatter: $S_b = \sum\limits_{i=1}^{c} N_i(\mu_i - \mu)(\mu_i - \mu)^T$

- show the first 25 eigenfaces

```
def LDA(X, X_test, train_labels, test_labels):
    numClass = 15
    mu = np.mean(X, axis=0)
    Sb = np.zeros((img_size, img_size))
    Sw = np.zeros((img_size, img_size))

    # Build Sw and Sb
    for i in range(1, numClass+1):
        faces_i = train_faces[train_labels == i]
        mu_i = np.mean(faces_i, axis=0)
        # (xk - µi)
        xk_mui = (faces_i - mu_i).T
        # Sw += Si = (xk - µi)(xk - µi).T
        Sw += xk_mui @ xk_mui.T

        # (µi - µ)
        mui_mu = (mu_i - mu).T
        # Sb += Ni(µi - µ)(µi - µ).T
        Sb += len(faces_i) * mui_mu @ mui_mu.T
```

We want to learn a projection $W$ such that it converts all the points from $x \in \mathbb{R}^m$ to a new space $z \in \mathbb{R}^n$, where in general $m$ and $n$ are unknown:

$$z = w^T x, x \in \mathbb{R}^m, z \in \mathbb{R}^n$$

After the projection, the between class scatter is $\widetilde{S}_B = W^T S_B W$, where $W$ and $S_B$ are calculated from our original dataset. The within class scatter is $\widetilde{S}_W = W^T S_W W$. So the objective becomes:

$$W_{opt} = \underset{W}{\operatorname{argmax}} \frac{\left|\widetilde{S}_B\right|}{\left|\widetilde{S}_W\right|} = \underset{W}{\operatorname{argmax}} \frac{|W^T S_B W|}{|W^T S_W W|}$$

Again, after applying Lagrange multipliers we obtain a generalized eigenvector problem, where we have:

$$S_B w_i = \lambda_i S_W w_i, i = 1, \ldots, m$$

- 
- The solution to this maximization problem is $S_W^{-1}S_Bw = \lambda w$
- Yet, $Rank(S_B) ≤ C-1$ and $Rank(S_w) ≤ N-C$, so pseudo inverse is applied instead of inverse.

```
def LDA(X, X_test, train_labels, test_labels):
    ...
    S = np.linalg.pinv(Sw) @ (Sb)
    np.save('npy/LDA_S.npy',S)

    eig_vals, eig_vecs = getEigen(S, method="LDA", loadMode=False,
sym=True)
    eif_vecs = eig_vecs / np.linalg.norm(eig_vecs, axis=0)

    # Part1 : show the first 25 eigenfaces
    # same as PCA above
    sort_idx = np.argsort(-eig_vals)
    U = (eig_vecs[:, sort_idx[:25]]).real
    faces = (U.T).reshape(25, rh, rw)
    show_faces(faces, 5, 5, "LDA : 25 eigenfaces")
```

- randomly pick 10 images to show theit reconstruction

```
same as the process in PCA
```

---

## Part2

- PCA
    - Do face recognition and compute the perfomance
    - 
```
def PCA(){
    ...
    # Part 2 : do face recognition and compute the performance
    U = eig_vecs[:, sort_idx[:25]]
    W_train = (X - mu) @ U
    W_test = (X_test - mu) @ U
    recognize_faces(W_train, W_test)
}
```

```
def recognize_faces(train_w, test_w, k=5):
    err = 0.0
    cnt_test = len(test_labels)
    cnt_train = len(train_labels)

    dist = np.zeros(cnt_train)
    for i in range(cnt_test):
        for j in range(cnt_train):
            dist[j] = cdist([test_w[i]], [train_w[j]], 'sqeuclidean')
        # select k nearest coordinates
        k_nearst = np.argsort(dist)[:k]
        predict = np.argmax(np.bincount(train_labels[k_nearst]))
        err += 1 if test_labels[i] != predict else 0
    print("Testing Acc = ", (cnt_test - err) / cnt_test)
```

- LDA
  - Do face recognition and compute the perfomance

  -
    ```
    same as the process in PCA
    ```

---

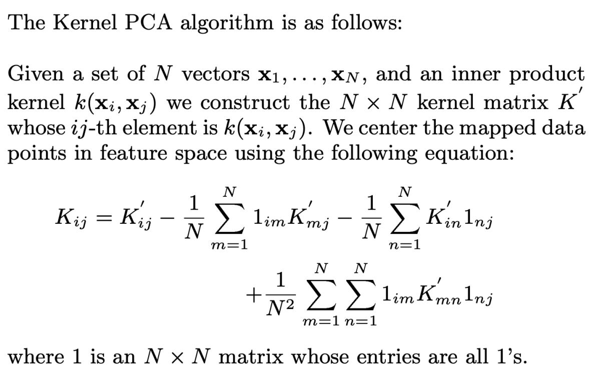
## Part3

- kernal

```
def RBF_kernel(xi, xj, gamma=1e-7):
    return np.exp( -gamma * cdist(xi, xj,'sqeuclidean'))
```

  - note that normal case of PCA, LDA can be seen as linear kernal

- kernal PCA
  - reference



The Kernel PCA algorithm is as follows:

Given a set of $N$ vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$, and an inner product kernel $k(\mathbf{x}_i, \mathbf{x}_j)$ we construct the $N \times N$ kernel matrix $K'$ whose $ij$-th element is $k(\mathbf{x}_i, \mathbf{x}_j)$. We center the mapped data points in feature space using the following equation:

$$K_{ij} = K'_{ij} - \frac{1}{N}\sum_{m=1}^{N} 1_{im}K'_{mj} - \frac{1}{N}\sum_{n=1}^{N} K'_{in}1_{nj} + \frac{1}{N^2}\sum_{m=1}^{N}\sum_{n=1}^{N} 1_{im}K'_{mn}1_{nj}$$

where 1 is an $N \times N$ matrix whose entries are all 1's.

  -

-
  ```
  def kernel_PCA(X, X_test, train_labels, test_labels):
      mu = np.mean(X, axis=0)
      X_mu = (X - mu).T
      # use X_mu instead of X just because it performs better
      K = RBF_kernel(X_mu, X_mu)
      lnn = np.ones((img_size, img_size)) / img_size
      S = K - (lnn @ K) - (K @ lnn) + (lnn @ K @ lnn)

      eig_vals, eig_vecs = getEigen(S, method="PCA_Kernal", 
  loadMode=False, sym=True)
      eif_vecs = eig_vecs / np.linalg.norm(eig_vecs, axis=0)
  ```
- reference

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

  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

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

-
  ```
  def kernel_LDA(X, X_test, train_labels, test_labels):

      K = RBF_kernel(X.T, X.T)
      mu_star = np.sum(K, axis = 0) / len(train_faces)

      numClass = 15
      Sb = np.zeros((img_size, img_size))
      Sw = np.zeros((img_size, img_size))
      for i in range(1, numClass+1):
          idx = np.where(train_labels == i)[0]
          Kj = K[idx]
          lj = len(idx)
  ```

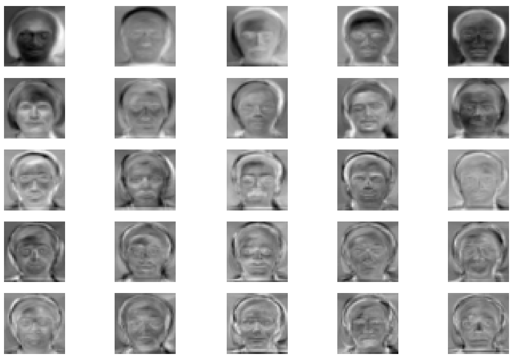
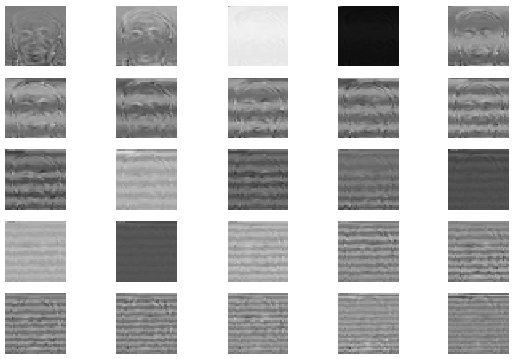
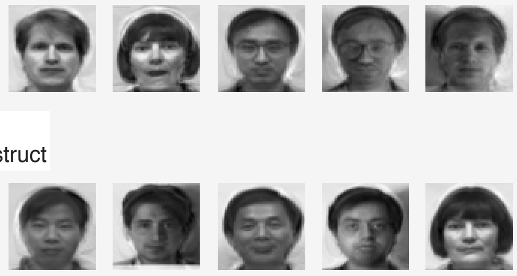
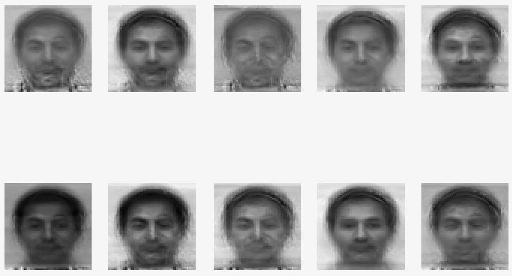
```
        mu_i = np.mean(Kj, axis=0)

        # 1_ij : all entries equal to 1/lj
        l_lj = np.full((lj, lj), 1.0 / lj)
        # Sw += Kj (I - 1_lj) Kj.T
        Kj = Kj.T
        Sw += Kj @ (np.eye(lj) - l_lj) @ Kj.T

        # (μi - μ*)
        mui_mu = (mu_i - mu_star).T
        # Sb += li(μi - μ*)(μi - μ*).T
        Sb += lj * mui_mu @ mui_mu.T

    S = np.linalg.pinv(Sw) @ (Sb)
    eig_vals, eig_vecs = getEigen(S, method="kernal_LDA",
loadMode=False, sym=True)
    eif_vecs = eig_vecs / np.linalg.norm(eig_vecs, axis=0)
```

---

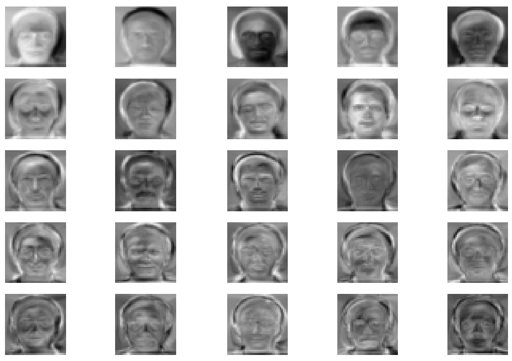
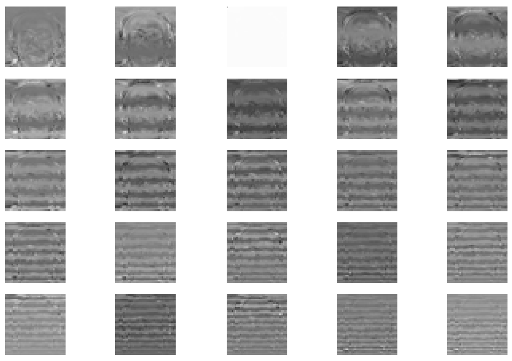
## B. Experiments Results & Discussion

### Kernel Eigenfaces

#### Part1 & part2

| <br> | PCA | LDA |
|---|---|---|
| 25 faces |  |  |
| 10 reconstruct |  |  |
| accuracy | 0.9 | 0.86 |

### Part3

| <br> | PCA | LDA |
|---|---|---|
| 25 faces |  |  |
| 10 reconstruct | | |
| accuracy | 0.86 | 0.86 |

## C. Observations and Discussion

- I think I do something wrong on LDA, but I can't figure them out QQ
- Not only because my LDA looks strange, but also, according to the reference and their reference, LDA should perform better especially in small component cases. Yet, in my case, PCA performs better than not only LDA, but also kernal ones, which is also different from the result I saw in the reference.

**TAGS:** `MACHINE LEARNING`