Q1 For the implementation, we consider three classes with size 121, 109, 144 respectively, and train/test set split of ratio 75 to 25. For each pair of images, we consider $N = 8$, i.e. the pairwise evaluation only involves neighbouring patches within distance $\sqrt{2}$ between pair of centroids (z_i, z_j). Then the misclassification error with respect to the two hyperparameters are plotted individually as follows,

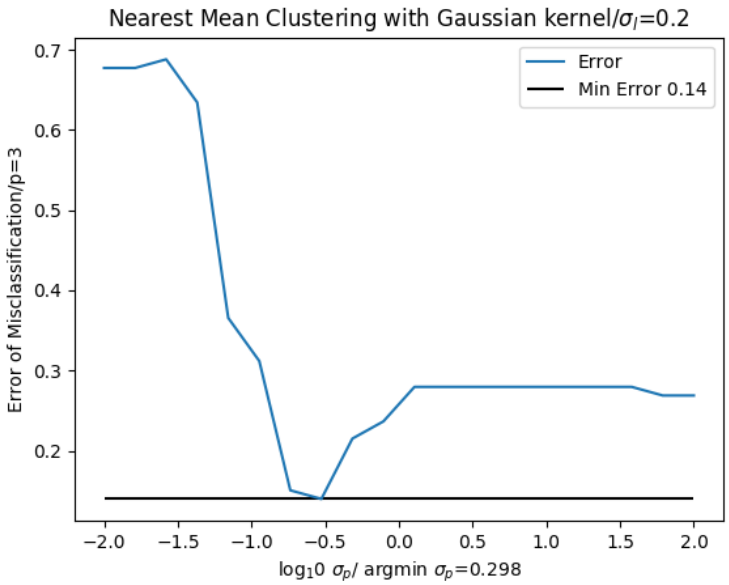


Figure 1: Misclassification error verses $log_{10}\sigma_p$ with fixed $\sigma_l = 0.2$.

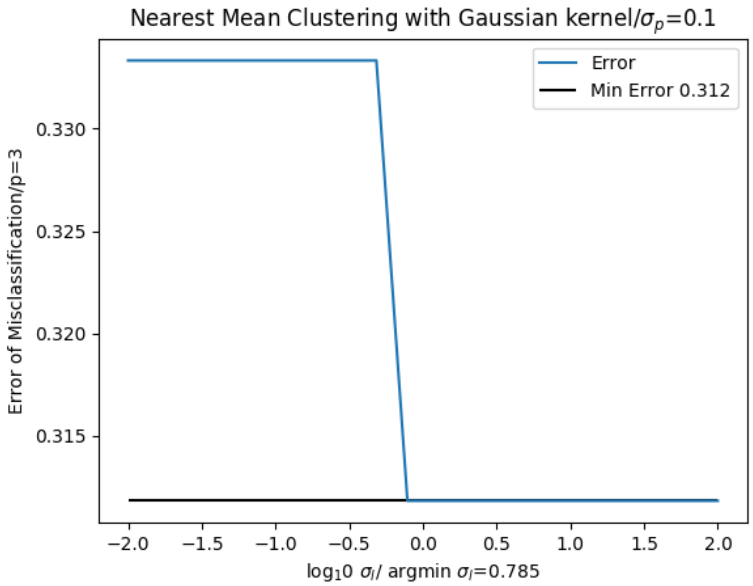


Figure 2: Misclassification error verses $log_{10}\sigma_l$ with fixed $\sigma_p = 0.1$.

Then the misclassification error is plotted for a 2d grid value of σ_p and σ_l as follows

We see that the classification performs the best for $log_{10}\sigma_p \approx -0.67$ and $log_{10}\sigma_l < 0$. And the performance is not good for small σ_p irrespective the choice of σ_l.

Q2 We first simulate random points with Gaussian noise from two concentric ellipses with $(4, 3.2)$ and

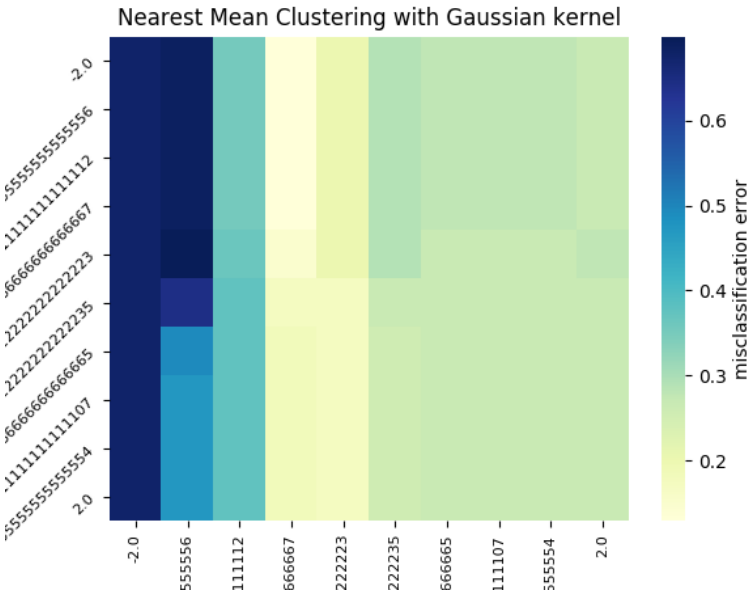


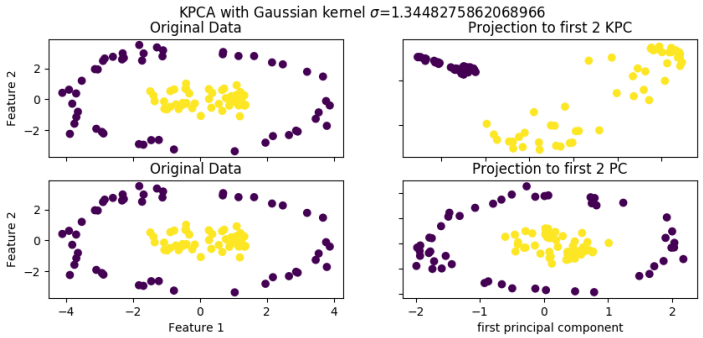
Figure 3: Misclassification error for $log_{10}\sigma_p$ (x-axis) verses $log_{10}\sigma_l$ (y-axis).

$(1.2, 0.5)$, the radius of semi-major axis and semi-minor axis for each ellipse. The power iteration is computed for the leading eigenvector and eigenvalue of K_p by iteratively updating the following

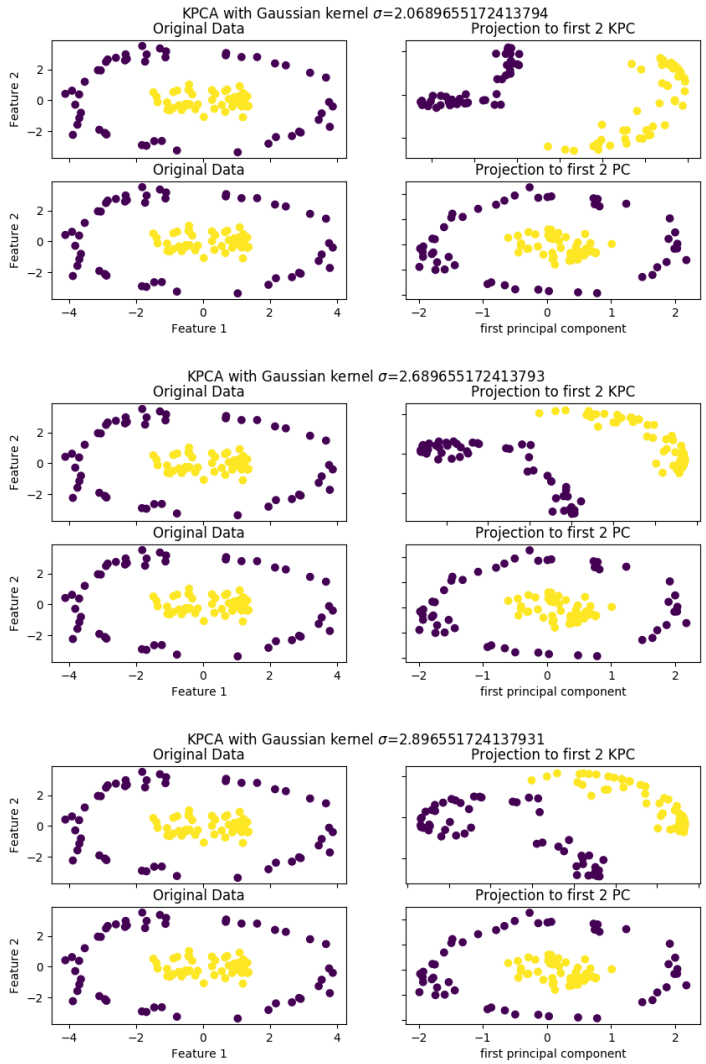
$$z_p^{(i)} \leftarrow K_p v_p^{(i-1)},\ v_p^{(i)} \leftarrow \frac{z_p^{(i)}}{\|z_p^{(i)}\|_2},\ \lambda_p^{(i)} \leftarrow v_p^{(i)T} K_p v_p^{(i)}, \quad \text{with } v_p^{(0)} = v_0$$

where $i = 1, \cdots, 700$, $v_0 \in \mathbb{R}^n$ random initialisation and $K_p = K - \sum_{i=1}^{p-1} \lambda_p v_p v_p^T$ is updated after each λ_p and v_p pair is found.

Then for the kernel PCA, we implement with Gaussian (RBF) kernel with varying hyperparameter $\sigma \in (0, 3)$, and visualise the projection of the data onto the two leading kernel principle components found. The result is compared with the PCA, having data projected onto the two leading principle components.



By experimentation, the seperation of the two classes are the best for $\sigma \in [1, 2.8)$ with the simulated data. And since we only have a 2d dataset, the principle components are just corresponding to the two axis of ellipses. And since rotation is not applied for the simulation, the projection onto first two PCs looks exactly the same as the data. However the projection onto first two KPC gives clear seperation of the two classes.



Q3 The kernel regression objective is as follows,

$$\min_{f\in\mathcal{F}} \Phi(f(x_1),\dots,f(x_n),\|f\|_{\mathcal{F}}^2) = \min_{f\in\mathcal{F}} \frac{1}{n}\sum_{i=1}^{n}(f(x_i)-y_i)^2 + \lambda\|f\|_{\mathcal{F}}^2$$

1. We consider for $\hat{\mu}_X = 0$ and $\hat{\mu}_Y = 0$. Since the V is monotonically increasing with respect to the norm $\|f\|_{\mathcal{F}}$, we may invoke the Representer Theorem. Therefore the solution admits the form $f = \sum_{i=1}^{n}\alpha_i k(x_i,\cdot)$, for some $\boldsymbol{\alpha}\in\mathbb{R}^n$.

   Therefore, $\|f\|_{\mathcal{F}}^2 = \boldsymbol{\alpha}^T K\boldsymbol{\alpha}$, where $K_{ij} = k(x_i,x_j)$. And the objective is equivalent to

   $$\min_{\boldsymbol{\alpha}\in\mathbb{R}^n} V = \min_{\boldsymbol{\alpha}\in\mathbb{R}^n} \frac{1}{n}(K\boldsymbol{\alpha}-\boldsymbol{y})^T(K\boldsymbol{\alpha}-\boldsymbol{y}) + \lambda\boldsymbol{\alpha}^T K\boldsymbol{\alpha}$$

   To find the minimum, we find the gradient of V in the direction of $\boldsymbol{\alpha}$ and set it to 0, thus

   $$\nabla_\alpha V = \frac{2}{n}K(K\boldsymbol{\alpha}-\boldsymbol{y}) + 2\lambda K\boldsymbol{\alpha} = 0 \quad \Longrightarrow \quad \boldsymbol{\alpha} = (\frac{1}{n}K + \lambda I_n)^{-1}\boldsymbol{y}$$

   Since the objective is strictly convex, we know the $\boldsymbol{\alpha}$ obtained is the minimum.

2. Let $g\in\mathcal{G}$ s.t. $\langle g, h(y_i,\cdot)\rangle = g(y_i) = y_i$, e.g. $I_g(y) = y$ the identity function. And for simplicity, we asusme that the empirical covariance operators are given by

   $$\hat{S}_{XX} = \frac{1}{n}\sum_{i=1}^{n} k(x_i,\cdot)\otimes k(x_i,\cdot),\ \hat{S}_{YY} = \frac{1}{n}\sum_{i=1}^{n} h(y_i,\cdot)\otimes h(y_i,\cdot),\ \hat{S}_{XY} = \frac{1}{n}\sum_{i=1}^{n} k(x_i,\cdot)\otimes h(y_i,\cdot)$$

Therefore, for $\mathbf{Id}_{\mathcal{F}} : \mathcal{F} \to \mathcal{F}$ the identity operator $\mathbf{Id}_{\mathcal{F}} f = f$

$$\begin{aligned}\Phi(f) &= \frac{1}{n}\sum_{i=1}^{n}(\langle f, k(x_i,\cdot)\rangle_{\mathcal{F}} - \langle g, h(y_i,\cdot)\rangle_{\mathcal{G}})^2 + \langle f, \lambda \mathbf{Id}_{\mathcal{F}} f\rangle \\ &= \frac{1}{n}\sum_{i=1}^{n}[\langle f, k(x_i,\cdot)\rangle_{\mathcal{F}}^2 + \langle g, h(y_i,\cdot)\rangle_{\mathcal{G}}^2 - 2\langle f, k(x_i,\cdot)\rangle_{\mathcal{F}}\langle g, h(y_i,\cdot)\rangle_{\mathcal{G}}] + \langle f, \lambda \mathbf{Id}_{\mathcal{F}} f\rangle \\ &= \langle f, (\hat{S}_{XX} + \lambda \mathbf{Id}_{\mathcal{F}}) f\rangle_{\mathcal{F}} + \langle g, \hat{S}_{YY} g\rangle_{\mathcal{G}} - 2\langle f, \hat{S}_{XY} g\rangle_{\mathcal{F}}\end{aligned}$$

3. We find the functional gradient in the direction of $h \in \mathcal{F}$, and $\epsilon \in \mathbb{R}^+$,

$$\begin{aligned}\Phi(f+\epsilon h) &= \langle f+\epsilon h, (\hat{S}_{XX} + \lambda \mathbf{Id}_{\mathcal{F}})(f+\epsilon h)\rangle_{\mathcal{F}} + \langle g, \hat{S}_{YY} g\rangle_{\mathcal{G}} - 2\langle f+\epsilon, \hat{S}_{XY} g\rangle_{\mathcal{F}} \\ &= \Phi(f) + 2\epsilon\langle h, (\hat{S}_{XX} + \lambda \mathbf{Id}_{\mathcal{F}}) f - \hat{S}_{XY} g\rangle_{\mathcal{F}} + O(\epsilon^2) \\ \implies \nabla\Phi(f+\epsilon h) &= 2(\hat{S}_{XX} + \lambda \mathbf{Id}_{\mathcal{F}}) f - 2\hat{S}_{XY} g\end{aligned}$$

Therefore, by setting the functional gradient to 0 we obtain the minimum

$$f = (\hat{S}_{XX} + \lambda \mathbf{Id}_{\mathcal{F}})^{-1} \hat{S}_{XY} g.$$

4. The kernel CCA on the sample $(x_i, y_i)_{i=1}^n$ has the below objective

$$\begin{aligned}&\sup_{f\in\mathcal{F}, g\in\mathcal{G}} \Psi(f(x_1), \cdots, f(x_n), g(x_1), \cdots, g(x_n), \|f\|_{\mathcal{F}}, \|g\|_{\mathcal{G}}) \\ =& \sup_{f\in\mathcal{F}, g\in\mathcal{G}} \frac{\langle f, \hat{S}_{XY} g\rangle}{\left(\langle f, \hat{S}_{XX} f\rangle + \lambda\|f\|_{\mathcal{F}}^2\right)^{1/2}\left(\langle g, \hat{S}_{YY} g\rangle + \lambda\|g\|_{\mathcal{G}}^2\right)^{1/2}}\end{aligned}$$

5. WLOG, fixing $f \in \mathcal{F}$, the $\sup_{g\in\mathcal{G}} \Psi(f,g)$ is obtained for $g = I_g$ the identity function in $\mathcal{G}$. Then for such $I_g \in G$, $\Psi(f,g)^{-1}$ is monotonically increasing with $\|f\|_{\mathcal{F}}$, we may invoke the Representer Theorem, then $f = \sum_{i=1}^n \alpha_i k(x_i,\cdot)$, for some $\boldsymbol{\alpha} \in \mathbb{R}^n$, and $I_g = \sum_{i=1}^n \beta_i k(x_i,\cdot)$, for some $\boldsymbol{\beta} \in \mathbb{R}^n$.

   Since $I_g(y) = \boldsymbol{\beta}^T \boldsymbol{y} y = y$, it implies $\boldsymbol{\beta}^T \boldsymbol{y} = 1$, and therefore $\|I_g\|_{\mathcal{G}} = (\boldsymbol{y}^T\boldsymbol{\beta})^T(\boldsymbol{y}^T\boldsymbol{\beta}) = 1$ and $\langle g, \hat{S}_{YY} g\rangle_{\mathcal{G}} + \lambda\|I_g\|_{\mathcal{G}} = \frac{1}{n}\boldsymbol{y}^T\boldsymbol{y} + \lambda$, which is a constant.

   With $\lambda_n = n\lambda$ and ignoring the constants, the objective in (4) can then be rewritten as

$$\begin{aligned}\sup_{f\in\mathcal{F}, g\in\mathcal{G}} \Psi(f,g) &= \sup_{\boldsymbol{\alpha}\in\mathbb{R}^n} \frac{\boldsymbol{y}^T K \boldsymbol{\alpha}}{(\boldsymbol{\alpha}^T(K^2 + \lambda_n K)\boldsymbol{\alpha})^{1/2}} \\ &= \sup_{\boldsymbol{\alpha}\in\mathbb{R}^n} \frac{(\boldsymbol{y}^T K \boldsymbol{\alpha})^T(\boldsymbol{y}^T K \boldsymbol{\alpha})}{(\boldsymbol{\alpha}^T(K^2 + \lambda_n K)\boldsymbol{\alpha})^{1/2}} \\ &= \sup_{\boldsymbol{a}\in\mathbb{R}^n} \frac{\boldsymbol{a}^T K(K^2+\lambda_n K)^{-1/2}\boldsymbol{y}\boldsymbol{y}^T K(K^2+\lambda_n K)^{-1/2}\boldsymbol{a}}{\|\boldsymbol{a}\|_2}\end{aligned}$$

   where $\boldsymbol{a} = (K^2 + \lambda_n K)^{1/2}\boldsymbol{\alpha}$. Then the above objective is again formulated as an eigen problem, which has the solution of the leading eigenvector of $\boldsymbol{a}^T K(K^2+\lambda_n K)^{-1/2}\boldsymbol{y}\boldsymbol{y}^T K(K^2+\lambda_n K)^{-1/2}\boldsymbol{a}$, denoted by $\boldsymbol{a} = v_1$. Therefore the original problem has the solution $\boldsymbol{\alpha} = (K^2 + \lambda_n K)^{1/2} v_1$.

6. By assuming the sample variance of $\{y_i\}_{i\geq 1}$, we have the variance $\langle g, \hat{S}_{YY} g\rangle_{\mathcal{G}} = \frac{1}{n}\sum_{i=1}^n y_i^2 = 1$, again assuming $g = I_g$. If we ignore the regularisation term, the objective reads

$$\sup_{f\in\mathcal{F}, g\in\mathcal{G}} \Psi(f,g) = \sup_{f\in\mathcal{F}} \frac{\langle f, \hat{S}_{XY} I_g\rangle}{\left(\langle f, \hat{S}_{XX} f\rangle + \lambda\|f\|_{\mathcal{F}}^2\right)^{1/2}}$$

   which is the same for the objective in (5) if the constant term is ignored. And it gives the same solution as the original construction.

Appendices

```
# Q1 Nearest Mean clustering ------------------------------------------------------------

# Image pre-processing
def Wrapper_Dat(cla, dat):
    """Wrapper function to process image data from lfw into patched format.
    If teo classes are inputed, relabel the data into class +/-1
    -----------
    Input
    cla: list; containing a pair or one label of class in lfw
    dat: 3d array; containing images in row
    p: integer; dimension of the patch
    -----------
    Output
    processed images a/na relabeled classes
    """
    target = dat.target

    cla_index = np.concatenate([np.where(target == i)[0] for i in cla])
    arr_dat = dat.images[cla_index]

    # scale the original image
    arr_dat = ((arr_dat - np.mean(arr_dat, axis=(-1,-2), keepdims=True))
                     /np.std(arr_dat, axis=(-1,-2), keepdims=True))
    proc_imag = Wrapper_extra_patches3(arr_dat)

    # change the label to +/-1; if input class is 2-dim
    if len(cla) is 2:

        labels = target[cla_index]
        labels = [-1 if i==cla[0] else 1 for i in labels]

        return proc_imag, np.array(labels)

    elif len(cla) is 1:

        return proc_imag
    else:

        return proc_imag, np.array(target[cla_index])

def extra_patches3(imag):
    """Extracts 3 by 3 patches of any nd array in place using strides,
    with stepsize 1
    -----------
    Input
    imag: np array storing images
    -----------
    Returns
     patches : strided 4d array
     4d array indexing patches in row and column order in the first two axes
     and containing patches on the last 2 axes.
    """
    import numpy as np
    from numpy.lib.stride_tricks import as_strided

    # pad the image with np.nan, width=1
    imag = np.pad(imag, 1, mode='constant', constant_values=np.nan)
```

```
    # dimension of image and patches
    i_h, i_w = imag.shape
    p = 3
    n_patches = (i_h-p+1) * (i_w-p+1)

    imag_ndim = imag.ndim
    patch_shape = tuple([p] * imag_ndim)
    patch_strides = imag.strides

    # the shape of produced array of patches
    patch_indices_shape = np.array(imag.shape) - np.array(patch_shape) + 1
    # output of strided array of patches
    shape_out = tuple(list(patch_indices_shape) + list(patch_shape))
    strides_out = tuple(list(patch_strides) * 2)
    patches = as_strided(imag, shape=shape_out, strides=strides_out)

    # renormalise by deviding the norm
    return patches/np.sqrt(np.sum(patches**2, axis=(-1,-2), keepdims=True)) # 4d array


def Wrapper_extra_patches3(arr_imag):
    """Wrapper function of extra_patches3 for each image in the data set
    """
    from itertools import starmap, product

    return np.array(list(starmap(extra_patches3, product(arr_imag))))

# kernel function
def ker_Imag(X, Y, kernel, hyper, sigma):
    """Compute the distance between patches in one image with the other only if
    the overlap between two patches are above 50%
    -----------
    Input
    X, Y: 4d array; storing patched image rocessed by 'extra_patches3'
    kernel: fuction; kernel function for patches
    hyper: scalar; hyperparameter of the kernel function
    sigma: sclar; hyper parameter of the Gaussian kernel of distances between patches
    -----------
    Return:
    scalar; mean of kernel evaluation between patches
    """
    # X, Y 4d array of same dimension, n, m, p, p
    n, m = X.shape[:2]
    X_true = X[1:(n-1), 1:(m-1), :, :]

    # distance^2 between >100%, <50% overlapped patch
    ker = [kernel(X_true, Y[1:(n-1), 1:(m-1), :, :], hyper)]
    # take centroids as the reference position for the patches
    for i in range(-1, 2, 2):
        # for the nearest overlaps, can find directly the distance between centroids of
            patches
        ker += [kernel(X_true, Y[(1+i):(n-1+i), 1:(m-1), :, :], hyper) * np.exp(-1/sigma**2)]
        ker += [kernel(X_true, Y[1:(n-1), (1+i):(m-1+i), :, :], hyper) * np.exp(-1/sigma**2)]
        ker += [kernel(X_true, Y[(1+i):(n-1+i), (1+i):(m-1+i), :, :], hyper) *
            np.exp(-2/sigma**2)]
        ker += [kernel(X_true, Y[(1-i):(n-1-i), (1+i):(m-1+i), :, :], hyper) *
            np.exp(-2/sigma**2)]

    return np.nanmean(np.array(ker))
```

```
def ker_Gauss3(u, v, l):
    """Gaussian kernel for vectors
    ----------
    Input
    u, v: 4d array; each position of first two axis stores patches as from 'extra_patches3'
    l: positive scalar; hyperparameter sigma
    ----------
    Output
    kernel evaluation between u and v
    """
    # u, v 3 by 3 array
    return np.exp(-np.sum((u - v)**2, axis=(-1,-2))/l**2)

def Wrapper_ker_Gauss(X_arr, Y_arr, l, sigma):
    """Wrapper function for evaluating the between image kernel value with 'ker_Imag'
    for Guassian kernel
    ----------
    Input
    X_arr, Y_arr: 5d array; first axis stores patched images in class X and Y respectively
    l: positive scalar; hyperparameter of Gaussian kernel
    -----------
    Output
    2d array; storing between image kernel evaluations from two classes X and Y.
    """
    from itertools import starmap, product

    ker = np.array(list(starmap(ker_Imag, product(X_arr, Y_arr, [ker_Gauss3],
                                                  np.array([l]), np.array([sigma])))))

    return ker.reshape(X_arr.shape[0], Y_arr.shape[0])

# nearest mean algorithm
def train_test_split(dat, lab, test_ratio=0.25, seed=17671):
    """Split dataset into train and test set (stratified)
    ----------------
    Input:
    dat: n darray; X
    lab: 1 darray; y
    test_ratio: [0,1]; test to train ratio
    ----------------
    Output:
    dat_train, dat_test, lab_train, lab_test
    """
    import numpy.random as rm
    rs = rm.RandomState(seed=seed)

    ind_test_lis = []; ind_train_lis = []

    for i in np.unique(lab):
        index = np.where(lab==i)[0]
        ind_test = rs.choice(index, round(len(index) * test_ratio), replace=False)

        ind_test_lis += list(ind_test)
        ind_train_lis += list(index[~np.isin(index, ind_test)])


    dat_test = dat[ind_test_lis]; dat_train = dat[ind_train_lis]
    lab_test = lab[ind_test_lis]; lab_train = lab[ind_train_lis]

    return dat_train, dat_test, lab_train, lab_test
```

```
def NNmean(dat_train, dat_test, lab_train, lab_test, kernel, hyper, sigma):
    """Find the nearest mean to the training set
    """
    n_cla = np.unique(lab_test)
    dist_lis = []
    # compute the corresponding distance of test set to class mean of train set
    for i in n_cla:

        index_train = np.where(lab_train==i)[0]
        samp_train = dat_train[index_train]

        ker_test = kernel(dat_test, dat_test, hyper, sigma)

        dist_lis += [np.diag(ker_test)
                     - 2 * np.mean(kernel(dat_test, samp_train, hyper, sigma), axis=1)
                     + np.mean(kernel(samp_train, samp_train, hyper, sigma))]

    dist_arr = np.array(dist_lis).reshape((len(n_cla), len(lab_test)))
    lab_pred = np.choose(np.argmin(dist_arr, axis=0), n_cla)

    # confusion matrix
    conf_mat = pd.crosstab(pd.Series(lab_test, name="actual"),
                           pd.Series(lab_pred.astype('int'), name="predict"))

    # print('confusion matrix', conf_mat)
    misclass = 1 - np.diag(conf_mat.values).sum()/conf_mat.values.sum()

    return {'lab_pred': lab_pred, 'conf_mat': conf_mat, 'error': misclass}


# visualisations
def Wrapper_Plot(kernel_dic, hyper_dic, arr_train, arr_test, lab_train, lab_test, p=3):
    """Wrapper function to plot the missclassification rate with varying hyper
    parameters.
    -------------
    Input:
    kernel_dic: dictionary; storing kernel name and corresponding function
    hyper_dic: dictionary; storing hyper parameter name and corresponding list
    of values
    -------------
    Output:
    Plot of misclassification rate against hyper parameter values
    """
    ker_name, kernel = list(kernel_dic.items())[0]

    # for j in range(len(hyper_dic)):
    hyper_name, hyper_lis = list(hyper_dic.items())[0]
    sigma_name, sigma_lis = list(hyper_dic.items())[1]
    # error_lis = []

    for i in hyper_lis:
        pred_class = NNmean(arr_train, arr_test, lab_train, lab_test, kernel, i, 0.2)
        error_lis += [pred_class['error']]

    min_error = min(error_lis)
    argmin = hyper_lis[error_lis.index(min_error)]

    log_hyper_lis = np.log10(hyper_lis)
    plt.figure()
    plt.plot(log_hyper_lis, error_lis, label='Error')
```

```
    plt.hlines(min_error, xmin=log_hyper_lis.min(), xmax=log_hyper_lis.max(),
            label='Min Error {}'.format(round(min_error,3)))
    plt.title('Nearest Mean Clustering with {a} kernel/{b}=0.2'.format(a=ker_name,
        b=sigma_name))
    plt.xlabel('$\log10 sigma_p^2$/ argmin $sigma_p^2$={}'.format(round(argmin,3)))
    plt.ylabel('Error of Misclassification/p={}'.format(p))
    plt.legend()
    plt.savefig('{a}_{b}.png'.format(a=hyper_name, b=p))
    plt.show()
    plt.close()

    error_lis = []
    for i in sigma_lis:
        pred_class = NNmean(arr_train, arr_test, lab_train, lab_test, kernel, 0.1, i)
        error_lis += [pred_class['error']]

    min_error = min(error_lis)
    argmin = sigma_lis[error_lis.index(min_error)]

    log_sigma_lis = np.log10(sigma_lis)
    plt.figure()
    plt.plot(log_sigma_lis, error_lis, label='Error')
    plt.hlines(min_error, xmin=log_sigma_lis.min(), xmax=log_sigma_lis.max(),
            label='Min Error {}'.format(round(min_error,3)))
    plt.title('Nearest Mean Clustering with {a} kernel/{b}=0.1'.format(a=ker_name,
        b=hyper_name))
    plt.xlabel('$\log10 sigma_l^2$/ argmin $sigma_l^2$={}'.format(round(argmin,3)))
    plt.ylabel('Error of Misclassification/p={}'.format(p))
    plt.legend()
    plt.savefig('{a}_{b}.png'.format(a=sigma_name, b=p))
    plt.show()
    plt.close()

    error_lis = []

    for k in range(len(hyper_lis)):
        i = hyper_lis[k]
        error_hyper = []

        for j in sigma_lis:
            pred_class = NNmean(arr_train, arr_test, lab_train, lab_test, kernel, i, j)
            error_hyper += [pred_class['error']]

        error_lis[k] = error_hyper

    log_hyper_lis = np.log10(hyper_lis)
    log_sigma_lis = np.log10(hyper_lis)

    plt.figure()
    plt.plot(log_hyper_lis, log_sigma_lis, np.array(error_lis), label='Error')
    plt.title('Nearest Mean Clustering with {a} kernel'.format(a=ker_name))
    plt.xlabel('$\log_{c}$ {a}'.format(c=10, a=hyper_name))
    plt.ylabel('$\log_{c}$ {a}'.format(c=10, a=sigma_name))
    plt.legend()
    plt.savefig('contour_error.png')
    plt.show()
    plt.close()


if __name__=='__main__':
    import matplotlib.pyplot as plt
```

```
    import numpy as np
    import pandas as pd
    import time

    # can we use this function, but what should be the ‘resize‘ argument? By default?
    from sklearn.datasets import fetch_lfw_people
    # Download the data, if not already on disk and load it as numpy arrays
    lfw_people = fetch_lfw_people(min_faces_per_person=40, resize=0.3)
    target = lfw_people.target
    # print('target has classes', np.unique(target, return_counts=True))

    # preprocessing
        ------------------------------------------------------------------------------
    # classes 1, 6, 10, 13, 14
    # with # images 42, 44, 42, 41, 41
    # cla=[1, 6, 10, 13, 14]; test_ratio = 0.25

    # classes 3, 5, 17
    # with # images 121, 109, 144
    cla=[3, 5, 17]; test_ratio = 0.25

    proc_imag, labels = Wrapper_Dat(cla, lfw_people)
    # print("dim of classes", np.unique(labels, return_counts=True))
    dat_train, dat_test, lab_train, lab_test = train_test_split(proc_imag,
                                            labels, test_ratio, seed=21203)
    print("dim of train ", dat_train.shape, '\n')
    print("dim of test ", dat_test.shape, '\n')
    print("train to test ratio ", np.unique(lab_train, return_counts=True),
                               np.unique(lab_test, return_counts=True))

    # # Varying hyperparams
        ------------------------------------------------------------------------

    kernel_dic = {
        'Gaussian': Wrapper_ker_Gauss,
        # 'VovkPolynomial': Wrapper_ker_Poly
        }
    hyper_dic = {
        '$\sigma_p$': np.logspace(-2,2,20),
        '$\sigma_l$': np.logspace(-2,2,20)
        }

    # a) Nearest Mean algorithm
    Wrapper_Plot(kernel_dic, hyper_dic, dat_train, dat_test, lab_train, lab_test)

# Q2 KPCA ------------------------------------------------------------------------------
def ker_Gauss(X, Y, l):
    n = X.shape[0]
    m = Y.shape[0]

    # binomial formula ~ Smola's Blog
    tmp = np.tile(np.sum(X**2, axis=1, keepdims=True), m)
    tmp += np.tile(np.sum(Y**2, axis=1), (n,1))
    tmp -= 2 * X @ Y.T

    ker = np.exp(-tmp/(l**2))
    return ker

# generate samples around two concentric ellipse
def samp_ell(range_lis, n=40, seed=17671):
```

```
    import numpy.random as rm
    rs = rm.RandomState(seed=seed)
    lis_samp = []

    for i in range_lis:
        # generate random angle
        phi = rs.uniform(0, 2*np.pi, n)
        # random rotation
        theta = rs.uniform(0, 2*np.pi, 1)

        samp = np.array([np.cos(phi) * i[0], np.sin(phi) * i[1]]).reshape((2, n))
        # rot_samp += [rot_mat @ samp + rs.normal(0, 0.3, size=(2, n))]
        lis_samp += [samp + rs.normal(0, 0.3, size=(2, n))]

    return lis_samp

def power_iter1(gram_mat, maxiter=700, seed=17671):

    # randomly generate a vector to initialise
    rs = rm.RandomState(seed=seed)
    v = rs.uniform(size=(gram_mat.shape[0], 1))
    l = 0

    # recursively compute the update
    n = 0
    while n < maxiter :
        n += 1

        z = gram_mat @ v
        v = z/LA.norm(z)
        l = v.T @ gram_mat @ v

    return l[0][0], v.flatten()

def power_iter(gram, k, maxiter=700, seed=17671):
    eval_lis = []
    evec_lis = []

    for i in range(k):
        l, v = power_iter1(gram, maxiter=maxiter, seed=17671)
        # print(v, '\n')
        eval_lis += [l]
        evec_lis += [v]

        # project on to subspace of v
        gram -= l * np.outer(v, v)

    # return the top k evecs and evals
    eval_arr = np.array(eval_lis)
    print('verify', eval_arr)
    evec_arr = np.array(evec_lis)/np.sqrt(eval_arr)[:, np.newaxis]

    return eval_arr, evec_arr

def centering(ker):
    n, m = ker.shape

    return ((np.eye(n) - np.ones((n,n))/n) @ ker @ (np.eye(m) - np.ones((m,m))/m))

def kPCA(dat, test, lab, kernel, hyper, k=2, plot=True):
    m = dat.shape[1]
```

```
    # ker_mat = ((np.eye(m) - np.ones((m,m))/m) @ kernel(dat.T, dat.T, hyper)
    #                                     @ (np.eye(m) - np.ones((m,m))/m))
    ker_mat = centering(kernel(dat.T, dat.T, hyper))

    eval_k, evec_k = power_iter(ker_mat, k, maxiter=700, seed=17671)

    ker_test = centering(kernel(dat.T, test.T, hyper))
    kpc = evec_k @ ker_test

    if plot:
        from sklearn.decomposition import PCA
        from sklearn.preprocessing import StandardScaler

        pca = PCA(n_components=2)
        X_std = StandardScaler().fit_transform(dat.T)
        X_pca = pca.fit_transform(X_std)

        plt.rcParams["figure.figsize"] = [10, 4]
        fig, axs = plt.subplots(2,2)
        fig.suptitle('KPCA with Gaussian kernel $\sigma$={}'.format(hyper))

        # visualise the clustering of the first two dimension
        plt.figure()
        axs[0,0].scatter(dat[0], dat[1], c=lab)
        axs[0,0].set_title('Original Data')

        axs[0,1].scatter(kpc[0], kpc[1], c=lab)
        axs[0,1].set_title('Projection to first 2 KPC')

        axs[1,0].scatter(dat[0], dat[1], c=lab)
        axs[1,0].set_title('Original Data')

        axs[1,1].scatter(X_pca[:, 0], X_pca[:, 1], c=lab)
        axs[1,1].set_title('Projection to first 2 PC')

        for ax in axs.flat[[0,2]]:
            ax.set(
                xlabel='Feature 1',
                ylabel='Feature 2')
        axs.flat[1].set(
                xlabel="first principal component",
                ylabel="second principal component")
        axs.flat[3].set(
                xlabel="first principal component",
                ylabel="second principal component")

        # Hide y ticks for right plots.
        for ax in axs.flat:
            ax.label_outer()

        fig.savefig('KPC{}.png'.format(hyper))
        # plt.show()
        plt.close()


    return eval_k, evec_k, kpc

def Wrapper_plot(dat, kernel, hyper, k=2):

    eval_2, evec_2 = kPCA(dat, kernel, hyper, k=2)
```

```
    proj_dat

if __name__=='__main__':
    import matplotlib.pyplot as plt
    import numpy as np
    import pandas as pd
    import time

    import numpy.random as rm
    from numpy import linalg as LA

    range_lis = [[4, 3.2], [1.2, 0.5]]
    n = 50

    samp = samp_ell(range_lis, n=n, seed=176)
    dat = np.concatenate((samp[0], samp[1]), axis=1)
    lab = np.repeat([0, 1], n)

    k = 2
    for hyper in np.linspace(0,3,num=30):
        eval_k, evec_k, emb_k = kPCA(dat, dat, lab, ker_Gauss, hyper, k=2, plot=False)
```