2018 ML HW7 Clustering

0756110 李東霖

This document made by HackMD, you can view here https://hackmd.io/s/BysklI4PeN (https://hackmd.io/s/BysklI4PeN)

implementation of Kernels

總共實作了四種 kernel :

- Linear
- Polynomial
- RBF
- Linear+RBF

```
def __Normal(x):
    return (x - np.min(x))/(np.max(x) - np.min(x))

def Linear(u,v):
    return np.matmul(u, v.T);

def Poly(u,v, gamma, coef0, degree):
    return ((gamma*np.matmul(u, v.T)) + coef0)**degree

def Euclidean(x,y):
    """
    calculate Euclidean distance
    parameters:
        x: n1 * d, y: n2 x d
    return:
        d: n1 * n2

    d(i, j) = |x(i) - y(j)|^2
    """
    if len(x.shape)==1:
        x = x[None,:]
    if len(y.shape)==1:
        y = y[None,:]
    return np.matmul(x**2, np.ones((x.shape[1],y.shape[0]))) \
    + np.matmul(np.ones((x.shape[0],x.shape[1])), (y**2).T) \
    - 2*np.dot(x,y.T)

def RBF(u,v,gamma=1):
    return np.exp(-1*gamma*Euclidean(u,v));

def LinearKernel():
    return lambda u,v:Linear(u,v)

def PolyKernel(gamma=1, coef0=1, degree=3):
    return lambda u,v:Poly(u,v,gamma,coef0,degree)

def RBFKernel(gamma=1):
    """
    generate callable function for RBF kernel
    parameters:
        gamma : default is 1
    return:
        lambda(u, v)

    rbf(u, v) = exp(-gamma|u-v|^2)
    """
    return lambda u,v:RBF(u,v,gamma)

def LinearRBFKernel(gamma=1):
    return lambda u,v:__Normal(Linear(u,v)) + RBF(u,v,gamma)
```

在 Linear+RBF kernel 中稍微做了點處理
因為 RBF kernel 的數值只會介在 $(0, 1]$ 之間，但是 Linear kernel 卻不是
如果直接相加會導致結果幾乎是等於 Linear kernel
因此先將 Linear kernel 線性變換到 $[0, 1]$ 再進行相加
在之後的使用可以看到 Linear+RBF 結果介在 Linear kernel 與 RBF kernel 之間

PCA & Kernel PCA

PCA 希望能找到一組投影向量能讓投影過去之後投影回來與原始資料的 MSE 最小

$$z = xW$$
$$MSE = ||x - zW^T||^2 = ||x - xWW^T||^2$$
$$= \text{constant} - \text{constant}W^T SW$$
$$S_{i,j} = \frac{\sum_{k=1}^{N}(x_k(i) - \bar{x(i)})(x_k(j) - \bar{x(j)})^T}{N}, x = \begin{bmatrix} x(0) \\ x(1) \\ \vdots \\ x(d) \end{bmatrix}$$

S 為 covariance matrix，經由上式得知要找到最小的 MSE 就是找到最大的 $W^T SW$
因此去解出 eigen problem $SW = \lambda W$，並取出擁有最大 eigen value 的 k 個 vector
就是 PCA 想找的投影向量也是該份資料的主成份

kernel trick

除了從原始資料空間進行投影，也可以嘗試從 feature space 去投影
但因為 data space 變換 feature space 不一定容易計算，因此使用 kernel trick 來計算

$$\frac{(\phi(X) - \frac{\sum_i^n \phi(Xi)}{N})(\phi(X) - \frac{\sum_i^n \phi(Xi)}{N})^T}{N} W = \lambda W, W = \phi(X)\alpha$$
$$\Rightarrow (K - 1_N K - K 1_N + 1_N K 1_N)\alpha = \lambda N \alpha$$
$$z = \phi(x)W = \phi(x)\phi(X)\alpha = K(x, X)\alpha$$

以下為實作的程式碼，eigen problem 使用 `numpy.linalg.eigh` 解

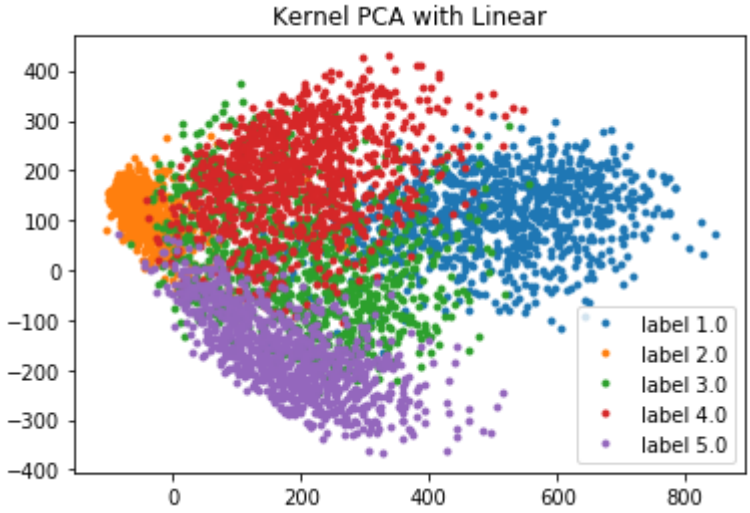
```python
def covariance(x, y):
    x1 = x - (np.sum(x, axis = 1) / x.shape[1])[:,None]
    y1 = y - (np.sum(y, axis = 1) / y.shape[1])[:,None]
    return np.matmul(x1, y1.T)/ x1.shape[1]

def PCA(datas, k, kernel=None):
    if callable(kernel):
        datas = kernel(datas, datas)
        N = datas.shape[0]
        N1 = np.ones((N,N))/N
        S = (datas
             - np.matmul(N1, datas)
             - np.matmul(datas, N1)
             + np.matmul(np.matmul(N1, datas), N1)
            )/N
    else:
        S = covariance(datas.T, datas.T)

    value, vector = np.linalg.eigh(S)

    start_idx = 0
    max_idxs = np.flip(np.argsort(value))
    W = np.concatenate([
        vector[:, max_idxs[i]][:,None]
        for i in range(start_idx,k+start_idx)
    ], axis=1)

    # whitening
    if callable(kernel):
        W /= np.sqrt(value[max_idxs[start_idx:start_idx+k]])[None, :]

    return np.matmul(datas, W), W, value, vector
```

回傳的第一個參數即是投影完的空間
使用 kernel PCA 要注意不是直接將計算得到的 eigen vector 使用在原本資料空間
而是要使用在 kernel matrix 上，因為實際上取得的 eigen vector 代表的是係數 $alpha$
這些係數乘上在 feature space 的所有資料點才會是投影向量 W

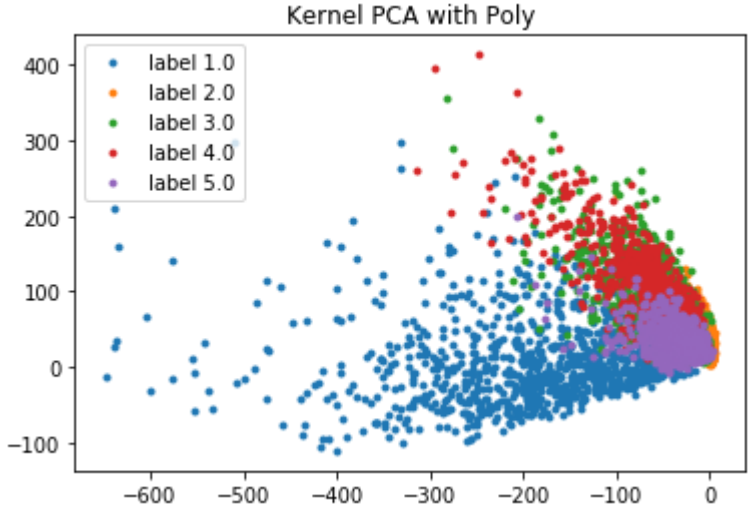
visualization

- PCA

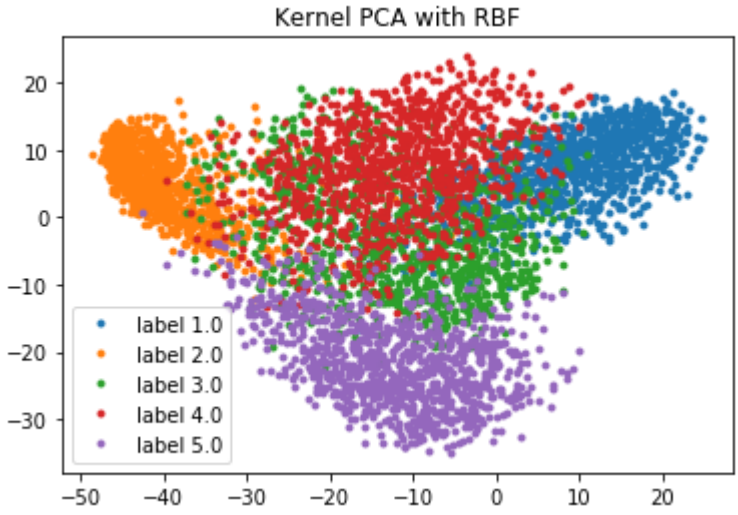
- PCA with Linear kernel



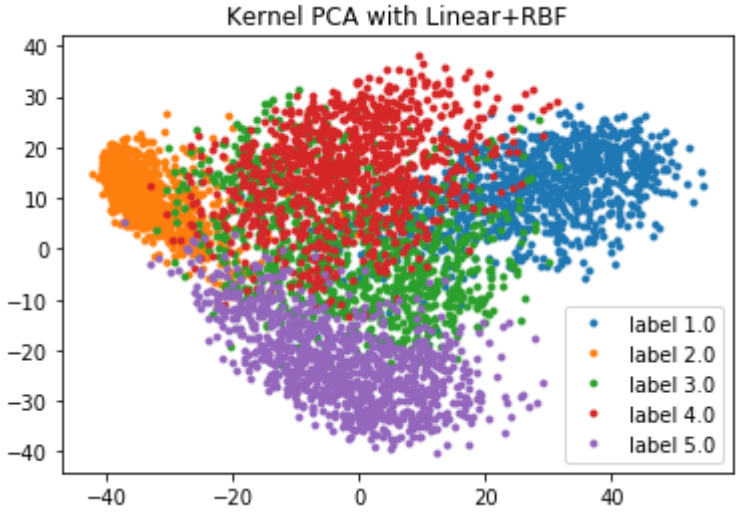
- PCA with Polynomial kernel
  $\gamma = 3/1000, \text{coef0} = 1, \text{degree} = 10$



- PCA with RBF kernel
  $\gamma = 1/100$



- PCA with Linear+RBF kernel
  $\gamma = 1/100$



my observation

這邊可以發現 PCA 與 PCA with Linear kernel 是非常相似
只是形狀上下左右顛倒，這是因為在 Linear kernel 中其實 feature space 與 data space 是一樣的
因此做出來會有類似結果也是合情合理

但畢竟計算過程與維度並不一樣因此座標軸範圍有差距

可以看到 RBF 的結果還不錯，同時 Linear+RBF 則確實介在 Linear 與 RBF 的結果

LDA & Kernel LDA

LDA 是一種 supervised learning，需要告知資料的標籤
求解過程也容易理解，希望找到一組投影向量 W 滿足以下條件

- 使得同一群的中心 (mean) 離所有資料的中心盡可能遠
  - maximize between-class-scatter S_B
- 同一群自己的變異數 variance 盡量小，也就是同群要集中
  - minimize within-class-scatter S_W

如何求解，如下

$$S_B = \sum_{j=1}^{k} |C_j|(m_j - m)(m_j - m)^T, m = \frac{\sum x}{n}, m_j = \frac{\sum_{x_i \in C_j} x_i}{|C_j|}$$

$$S_W = \sum_{j=1}^{k} \sum_{x_i \in C_j} (x_i - m_j)(x_i - m_j)^T$$

$$\$S_B W = \lambda S_W W \Rightarrow S_W^{-1} S_B W = \lambda W$$

因此我們只要解出 $S_W^{-1} S_B$ 的 eigen vector 即可找到投影向量

kernel trick

這樣一樣需要使用 kernel trick，推導來源參考 kernel LDA from wiki
(https://en.wikipedia.org/wiki/Kernel_Fisher_discriminant_analysis)

$$S_B^{\phi} = \sum_{j=1}^{k} |C_j|(m_j^{\phi} - m^{\phi})(m_j^{\phi} - m^{\phi})^T, m_j^{\phi} = \frac{\sum_{x_i \in C_j} \phi(x_i)}{|C_j|}$$

$$S_W^{\phi} = \sum_{j=1}^{k} \sum_{x_i \in C_j} (\phi(x_i) - m_j^{\phi})(\phi(x_i) - m_j^{\phi})^T$$

$$W = \sum_{i}^{w} \sum_{j}^{n} W_{i,j} \phi(x_j)$$

$$\frac{W^T S_B^{\phi} W}{W^T S_W^{\phi} W} = \frac{\alpha^T M \alpha}{\alpha^T N \alpha}$$

$$M = \sum_{j=1}^{k} |C_J|(M_j - M_{all})(M_j - M_{all})^T, M_j = \frac{\sum_{x_i \in C_j} \sum_{x_j}^{n} K(x_i, x_j)}{|C_j|}$$

$$N = \sum_{j=1}^{k} K_j (I - 1_{|C_j|}) K_j^T$$

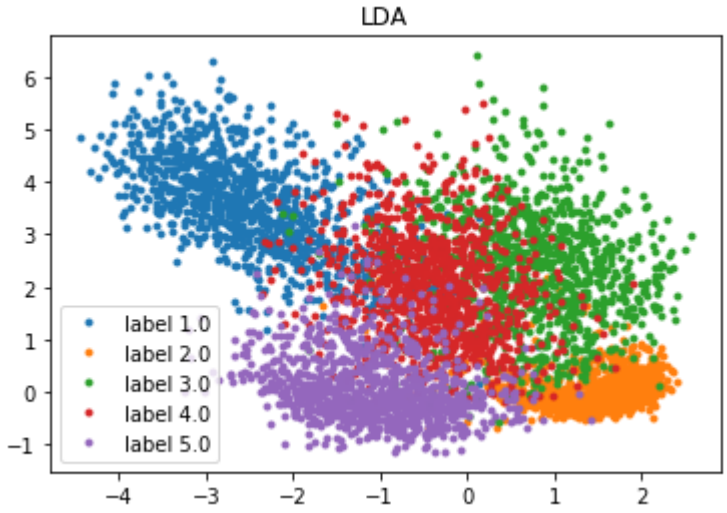
這邊是實作的程式碼

```
def LDA(datas, labels, k, kernel=None):
    C = np.zeros((datas.shape[0], len(np.unique(labels))))
    for idx, j in enumerate(np.unique(labels)):
        C[labels == j,idx] = 1;

    if callable(kernel):
        datas = kernel(datas, datas)

    Mj = np.matmul(datas.T, C) / np.sum(C, axis = 0)
    M = np.sum(datas.T, axis = 1) / datas.shape[0]

    B = Mj - M[:,None]
    SB = np.matmul(B * np.sum(C, axis = 0), B.T)

    W = datas.T - np.matmul(Mj, C.T)
    SW = np.zeros(SB.shape)
    for group in np.unique(labels):
        w = W[:, labels == group]
        SW += (np.matmul(w, w.T) / w.shape[1])

    try:
        SW_inv = np.linalg.inv(SW)
    except:
        SW_inv = np.linalg.pinv(SW)

    SBW = np.matmul(SW_inv, SB)
    value, vector = np.linalg.eigh(SBW)

    start_idx = 1 if callable(kernel) else 0
    max_idxs = np.flip(np.argsort(value))
    W = np.concatenate([
        vector[:, max_idxs[i]][:,None]
        for i in range(start_idx,start_idx+k)
    ], axis=1)

    return np.matmul(datas, W), W
```

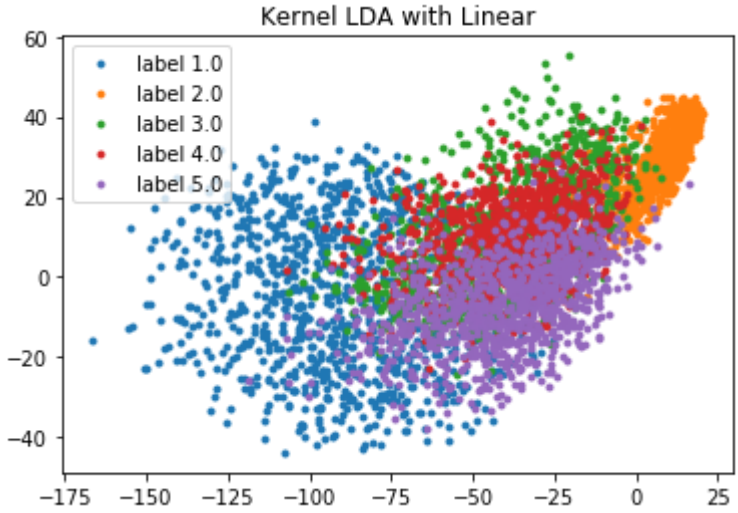
基本架構與 PCA 差不多，關鍵在於要找 eigen vector 的矩陣是誰
這樣有一個需要注意的地方是，因為需要計算矩陣的 inverse
因此要考量到如果不能 inverse 的時候要使用 pseudo inverse

visualization

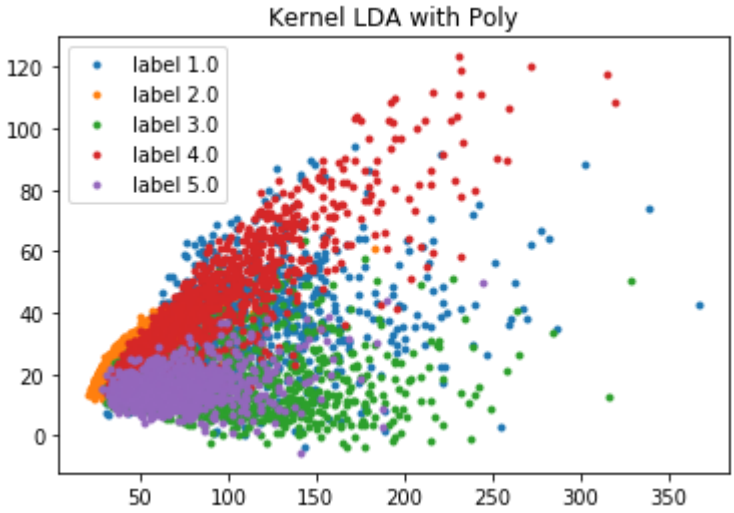
- LDA



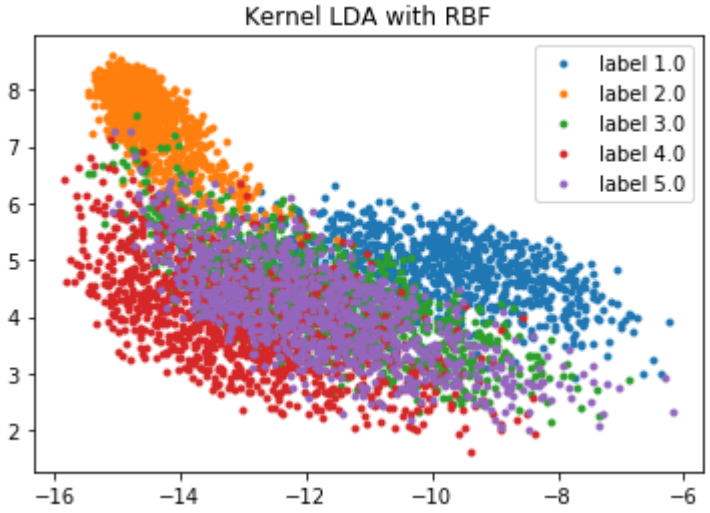
- LDA with Linear kernel



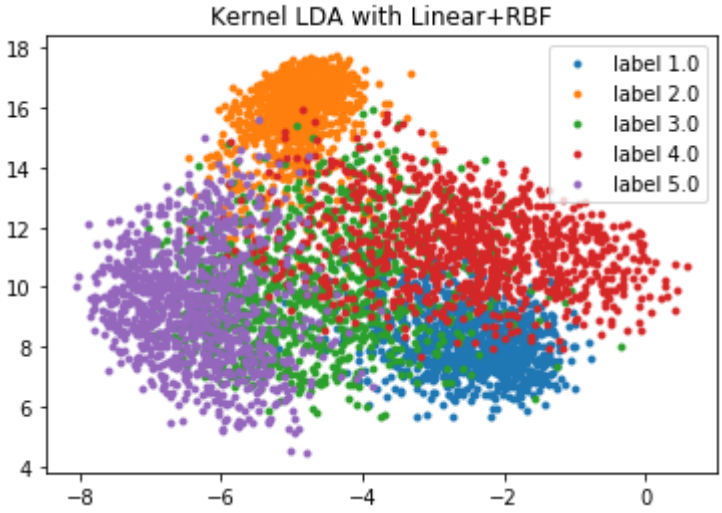
- LDA with Polynomial kernel



- LDA with RBF kernel



- LDA with Linear+RBF kernel



my observation

在單純的 LDA (without kernel)下，可以看到各群中心盡可能分開
且同一群盡可能靠近，這結果確實比 PCA 好，畢竟 LDA 是知道各群的分類

加入 kernel 之後，發現到 Linear kernel 的結果並沒有跟單純的 LDA 一樣
個人認為是我的實作有問題，因此反覆檢查與推敲也沒有找到答案

本以為 RBF 結果會不錯，但是也沒有
反倒是 Linear+RBF 看起來比較好

RatioCut & NormalizeCut

這邊參考我自己的 2018 ML HW6 Clustering (https://hackmd.io/s/rJ-O49ikE#steps-amp-my-implementation)
實作的程式，並且加入不同的 cut
改寫成以下程式

```
def __normalizecut(W,D):
    d = np.sqrt(np.linalg.inv(D))
    return np.matmul(np.matmul(d, D-W), d)

def __generalizecut(W,D):
    return np.matmul(np.linalg.inv(D), D-W)

__spectral_cut = {
    0 : lambda W,D:D-W,
    1 : lambda W,D:(D-W)/D.shape[0],
    2 : __normalizecut,
    3 : __generalizecut,
}
def Spectral(datas, kernel, k=None, cutType=0):
    """
    spectral clustering
    parameters:
        datas : data source n*d
        kernel : function for compute kernel
        k : mapping data to k-dim eigen space
        cutType:
            - 0 : simple cut, D-W
            - 1 : ratio cut, (D-W)/|V|
            - 2 : normalize cut, D^-1/2(D-W)D^-1/2
            - 3 : generalize cut, D^-1(D-W)
    return:
        U : data in eigen space n*k
        eigen values : n*1
        eigen vector : n*n
    """
    if not callable(kernel):
        raise AttributeError;
    W = kernel(datas, datas)
    D = np.sum(W, axis=1)*np.eye(W.shape[0])

    if not cutType in __spectral_cut:
        raise AttributeError;
    L = __spectral_cut[cutType](W,D)

    eig_values, eig_vectors = np.linalg.eigh(L)

    return eig_values, eig_vectors

def GetKeigen(eigen_values, eigen_vectors, k, isMin=True, startidx=1):
    sort_idxs = np.argsort(eigen_values)
    if not isMin:
        sort_idxs = np.flip(sort_idxs)
    return np.concatenate([
        eigen_vectors[:,sort_idxs[i]][:,None]
        for i in range(startidx, startidx+k)
    ], axis=1)
```

kernel 實作使用前面所寫的部份

這邊加入三種 cut 的算法，總共四種，如下

- SimpleCut
  - $\Rightarrow L = D - W$
- RatioCut
  - $\Rightarrow L_R = \frac{L}{|V|} = \frac{D-W}{|V|}$
- NormalizedCut
  - $\Rightarrow L_N = D^{-\frac{1}{2}} L D^{-\frac{1}{2}}$
- GeneralizeCut
  - $\Rightarrow L_G = D^{-1} L$

目標一樣是希望 最小化 cut，因此找出第二小之後的 k 個 eigen vectors
為什麼不找最小的，是怕出現全連接的情況因此避開(讓所有點都成為一群，cut 就會是 0)

我將 spectral clustering 分群過程分成三步(三個 function)，方便調整與觀察

1. 計算相似度矩陣並根據 cut 找出 eigen values 和 eigen vectors
2. 根據 eigen values 挑出 k 個 eigen vectors，並組成 eigen space
3. 將 eigen space 丟入 kmeans，得到分群結果

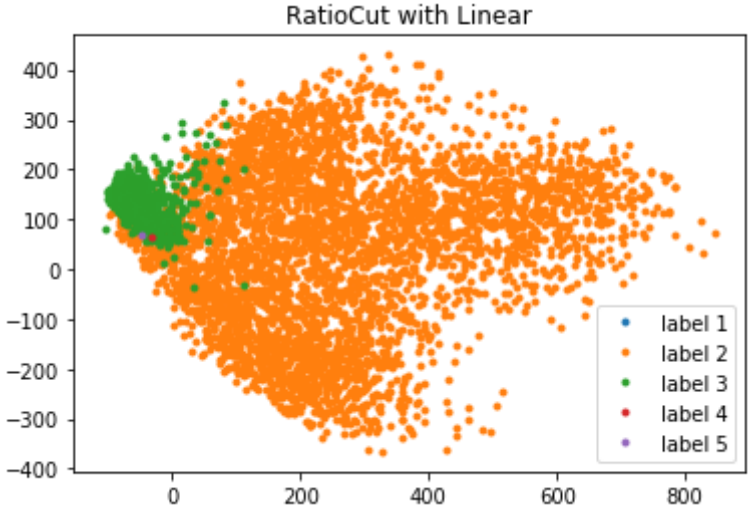
kmeans 實作直接拿 HW6 (https://hackmd.io/s/rJ-O49ikE#k-means--kernel-k-means) 就不重複贅述

以下將跑三種不同 kernel 搭配兩種不同的 cut
參數使用如下

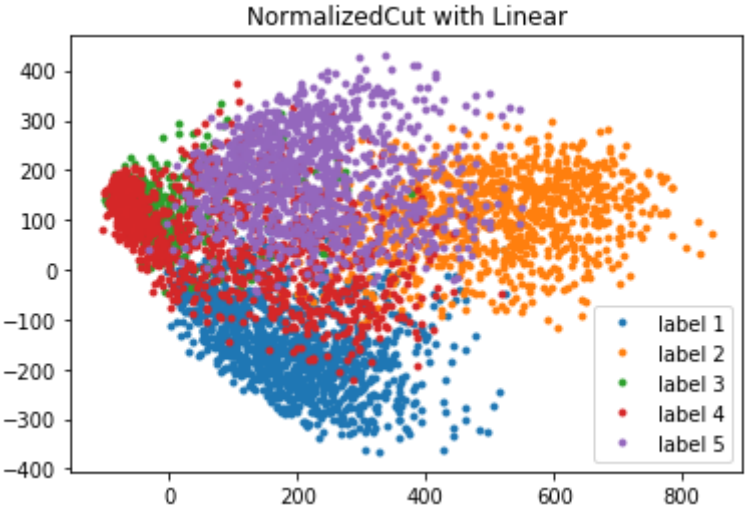
- gamma $\gamma = 0.01$

visualization (3 Kernel with 2 cut)

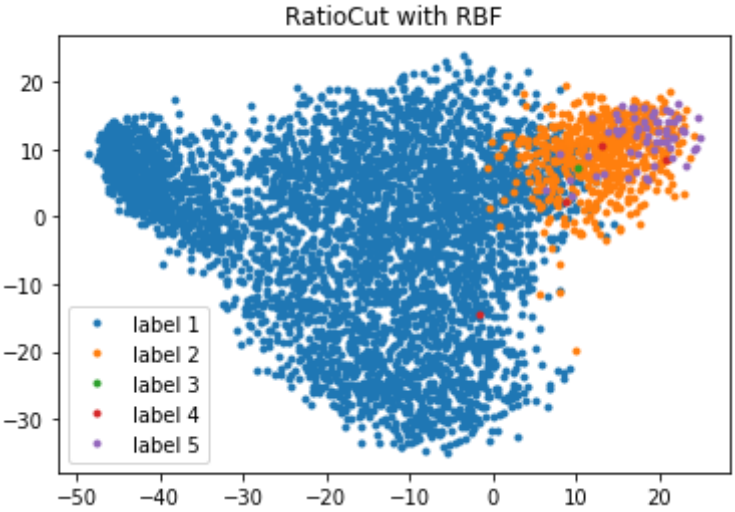
- Linear
  - RatioCut



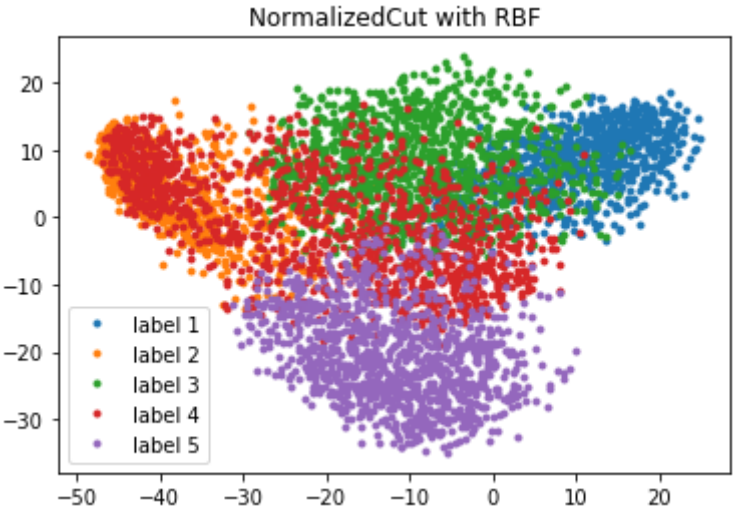
- NormalizeCut



- RBF
  - RatioCut

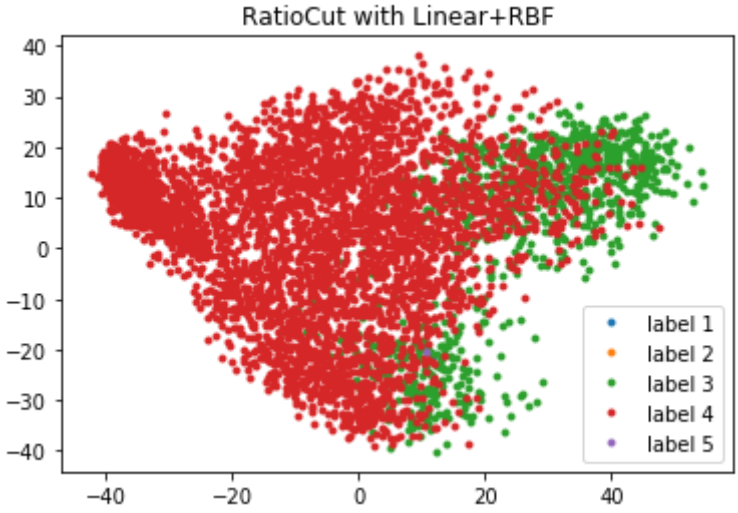


  - NormalizeCut

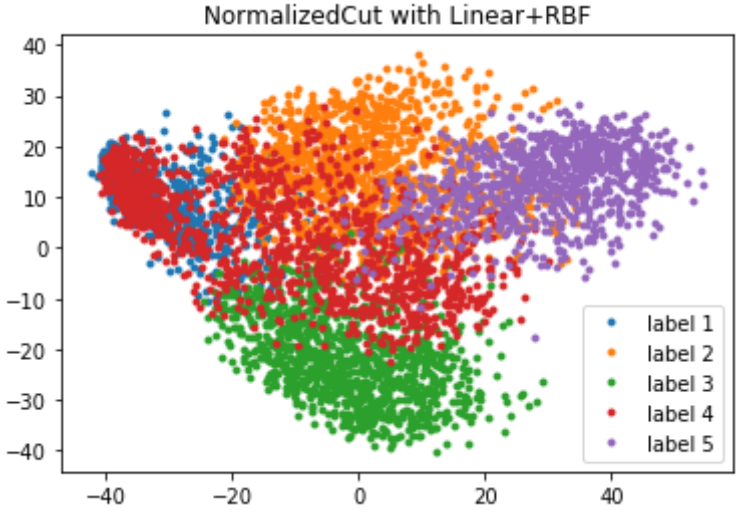


- Linear + RBF

- RatioCut



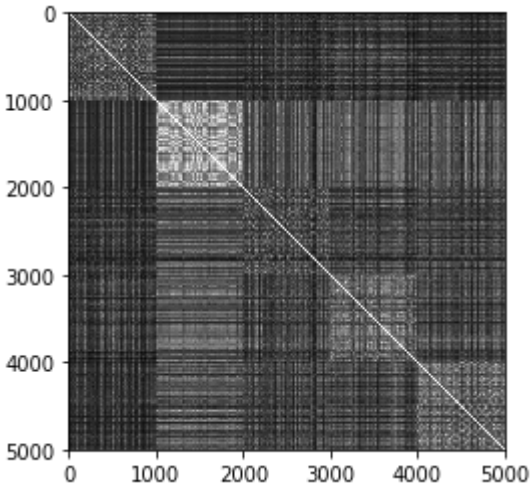
- NormalizeCut



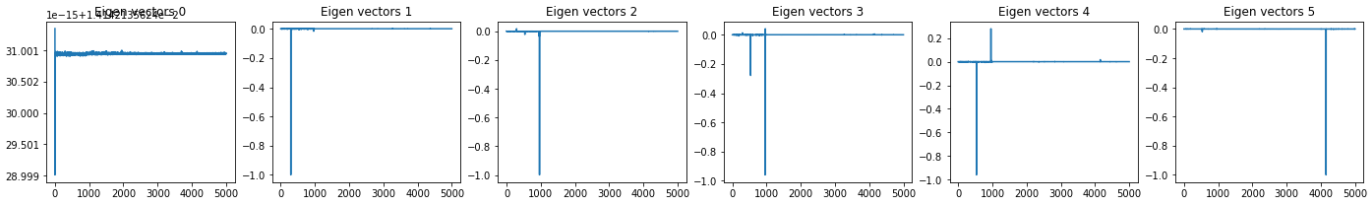
my observation

兩種不同的 cut 的結果非常不同，NormalizeCut 能很好在圖上看到 5 個群
而 RatioCut 則在各群之間的數量差距過大，導致直接看圖會覺得有些群消失了

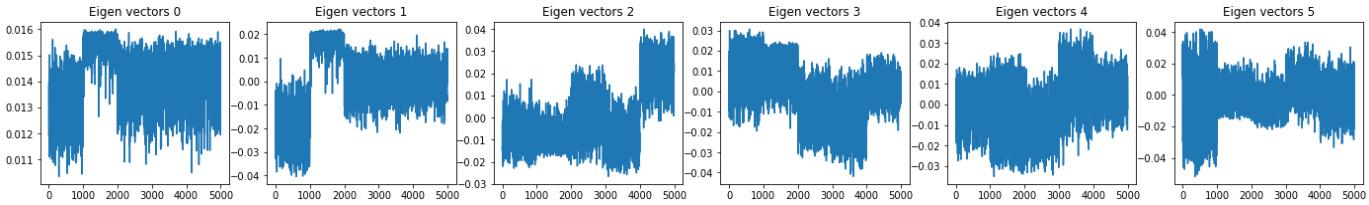
這樣的現象可以從相似度矩陣略知一二
下圖是 RBF kernel 計算得到的相似度矩陣



可以發現有一塊特別白(數值較高)，因此容易被分成一群
但是在沒有 normalize 的情況下，其他非這一塊的容易直接被分在同一群



因此加入 normalize 試圖在考慮相似度下去平衡每一群的數量
所以找到的 eigen vector 就盡量減少過度不平衡的情況

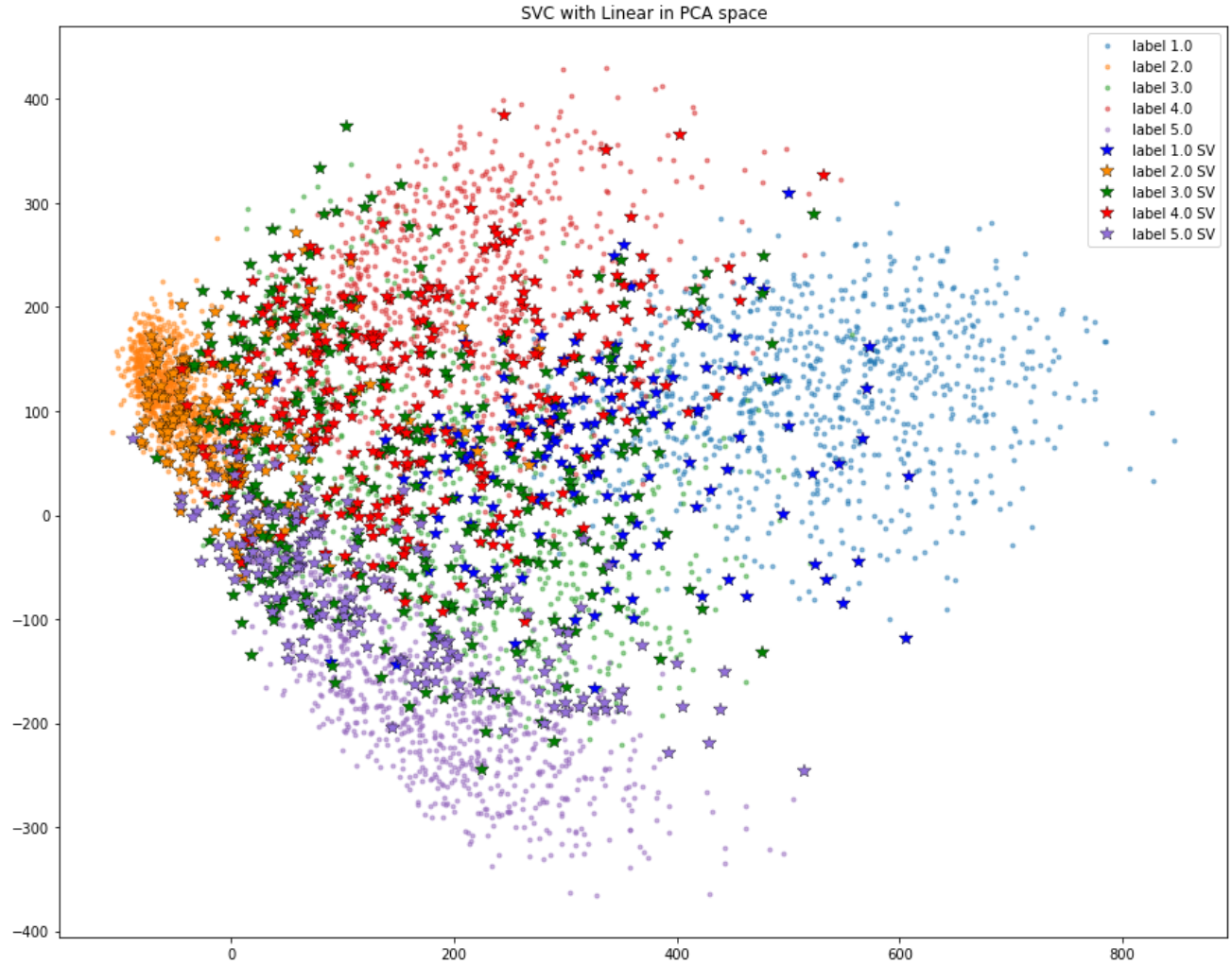


SVC result visualization

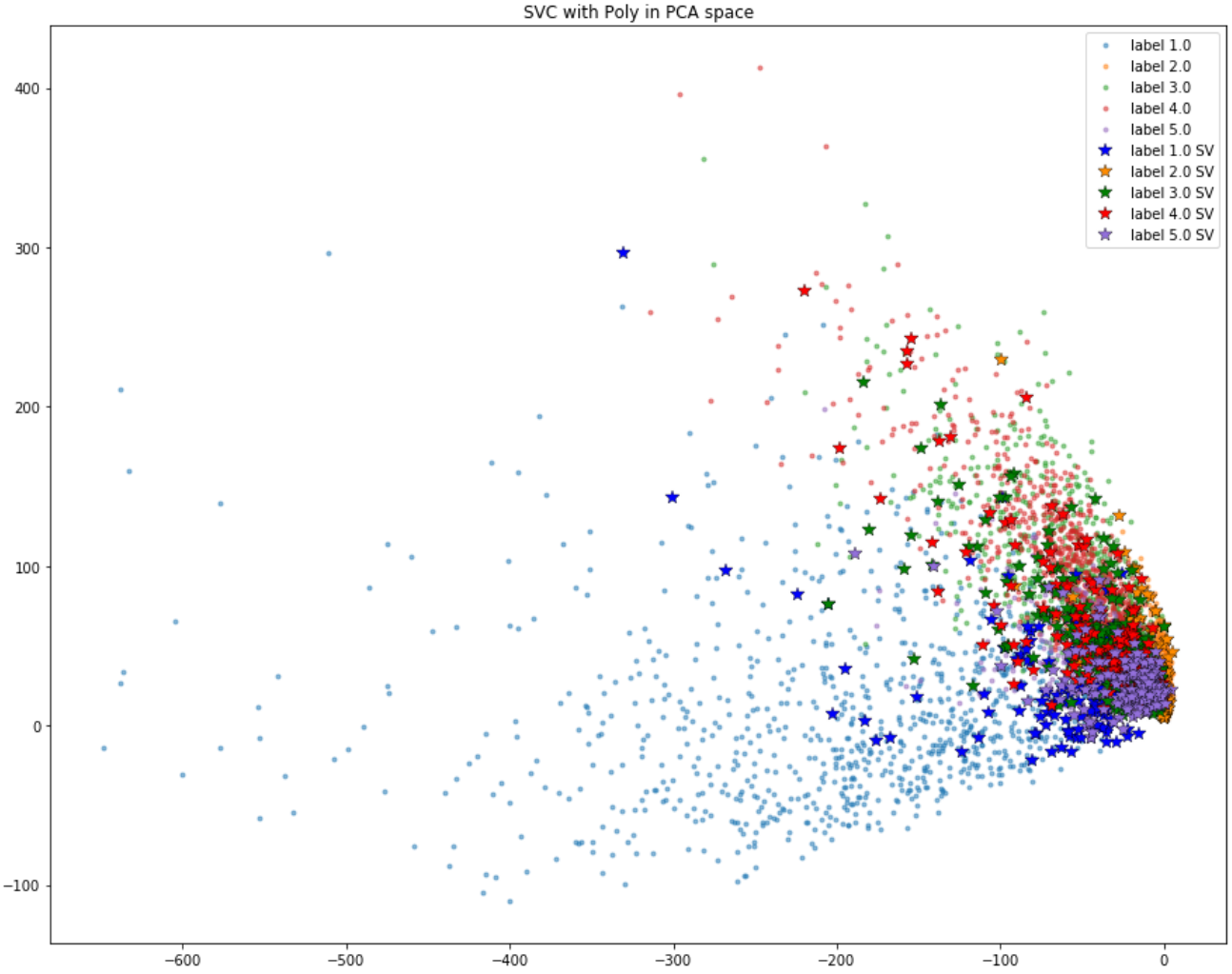
直接引用 HW5 的程式碼，將分群得出的結果與 support vector 呈現出來

In PCA space

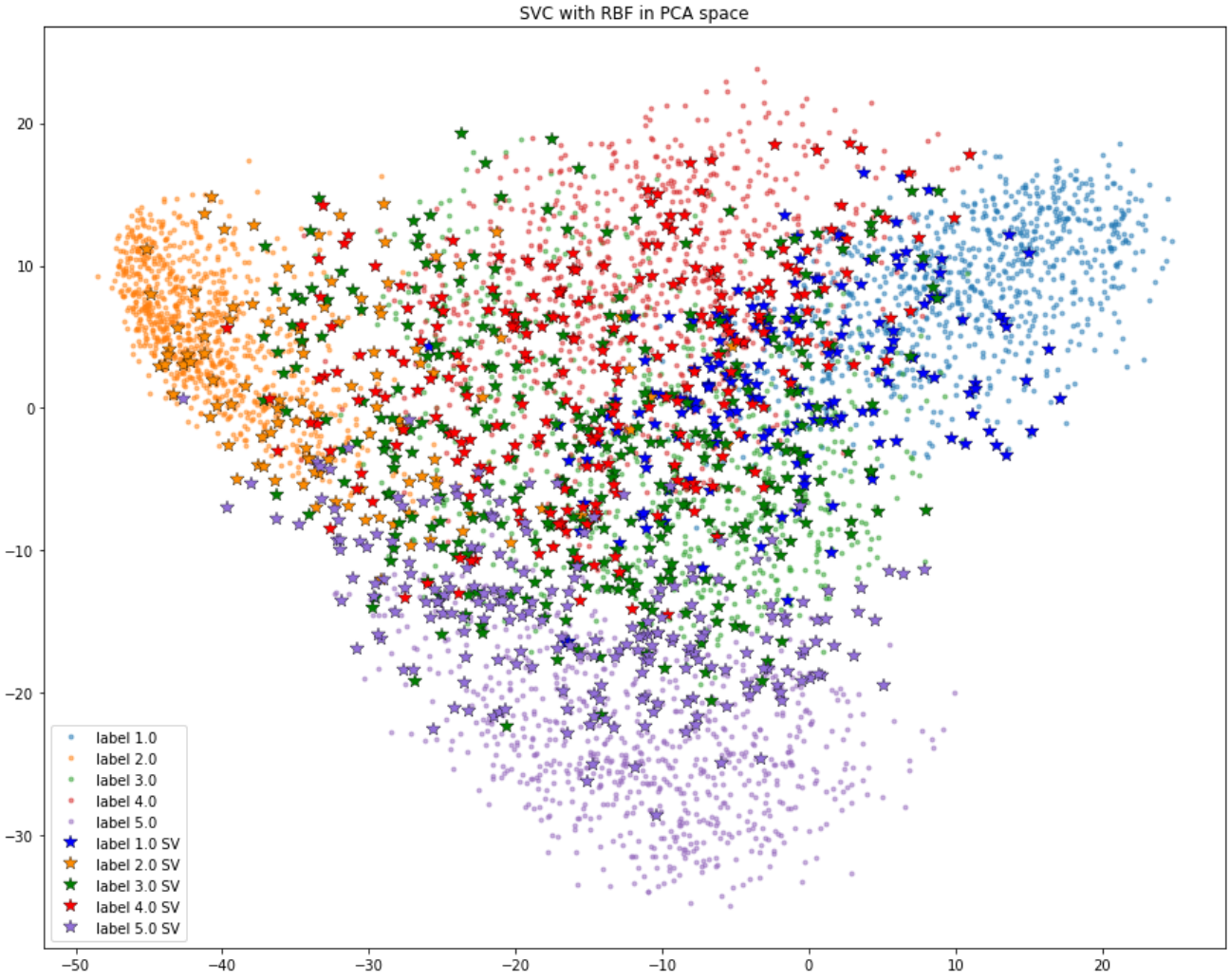
- Linear



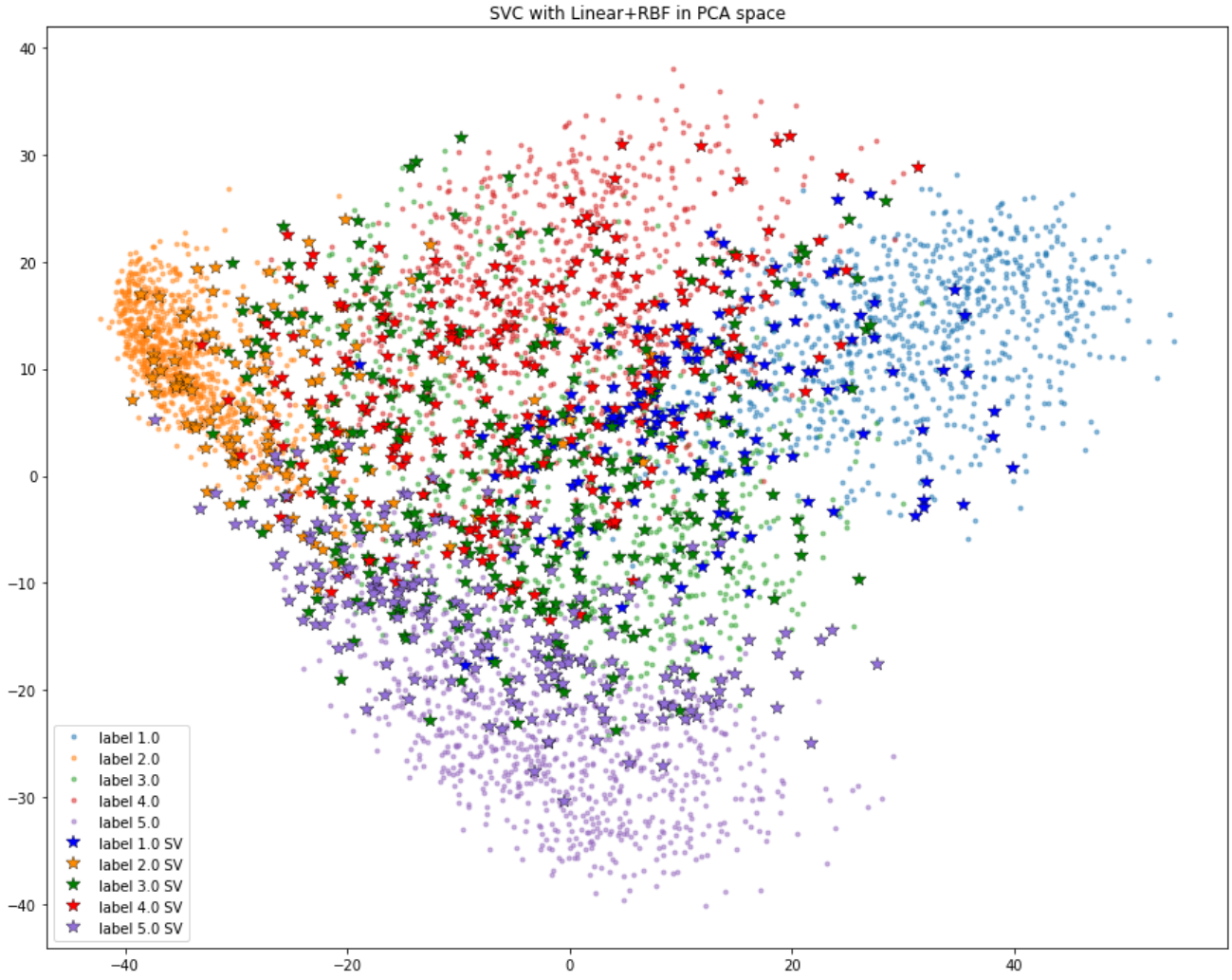
- Polynomial


SVC with Poly in PCA space
label 1.0
label 2.0
label 3.0
label 4.0
label 5.0
label 1.0 SV
label 2.0 SV
label 3.0 SV
label 4.0 SV
label 5.0 SV
400
300
200
100
0
−100
−600
−500
−400
−300
−200
−100
0

- RBF

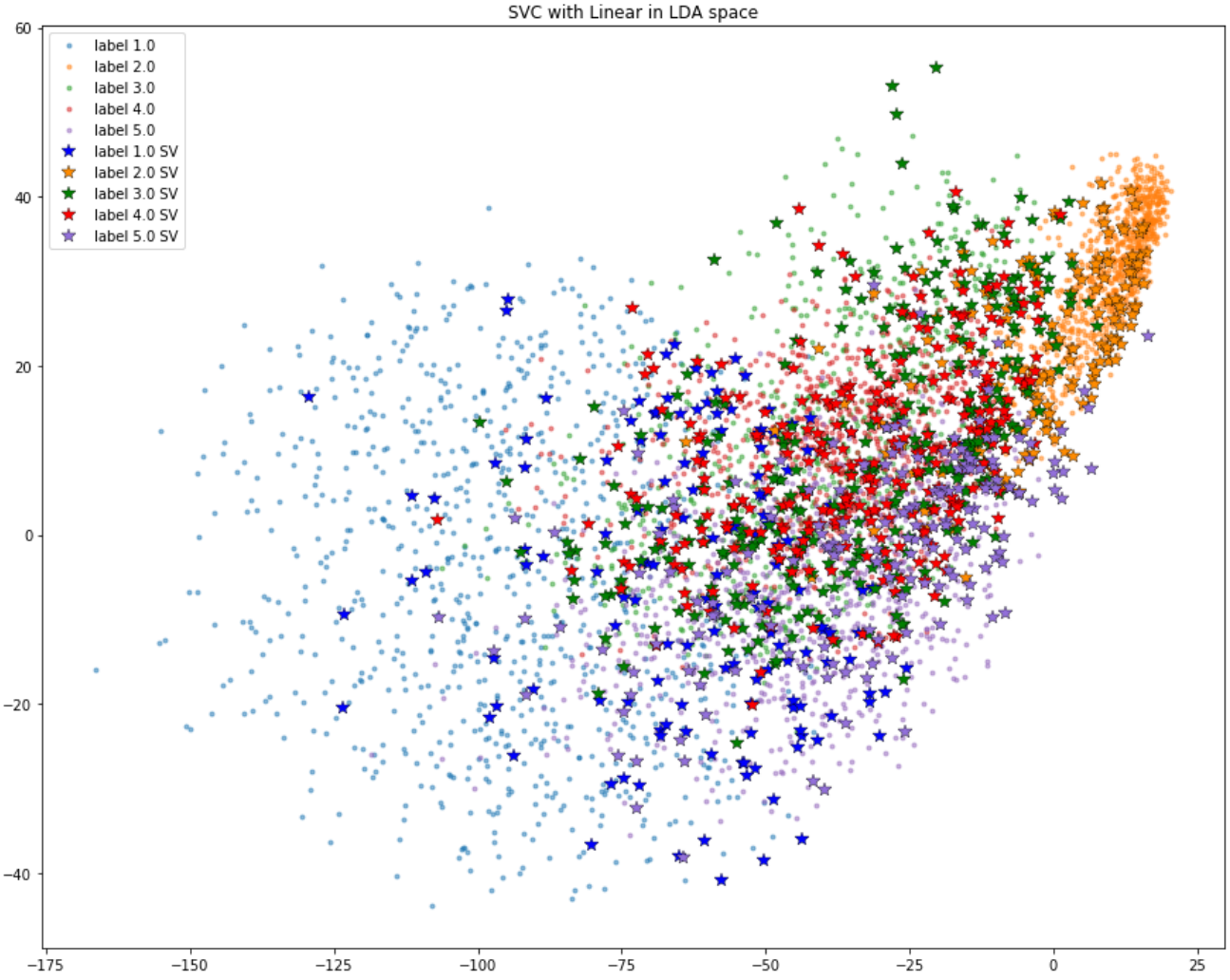

SVC with RBF in PCA space
20
10
0
−10
−20
−30
−50
−40
−30
−20
−10
0
10
20
label 1.0
label 2.0
label 3.0
label 4.0
label 5.0
label 1.0 SV
label 2.0 SV
label 3.0 SV
label 4.0 SV
label 5.0 SV

- Linear+RBF

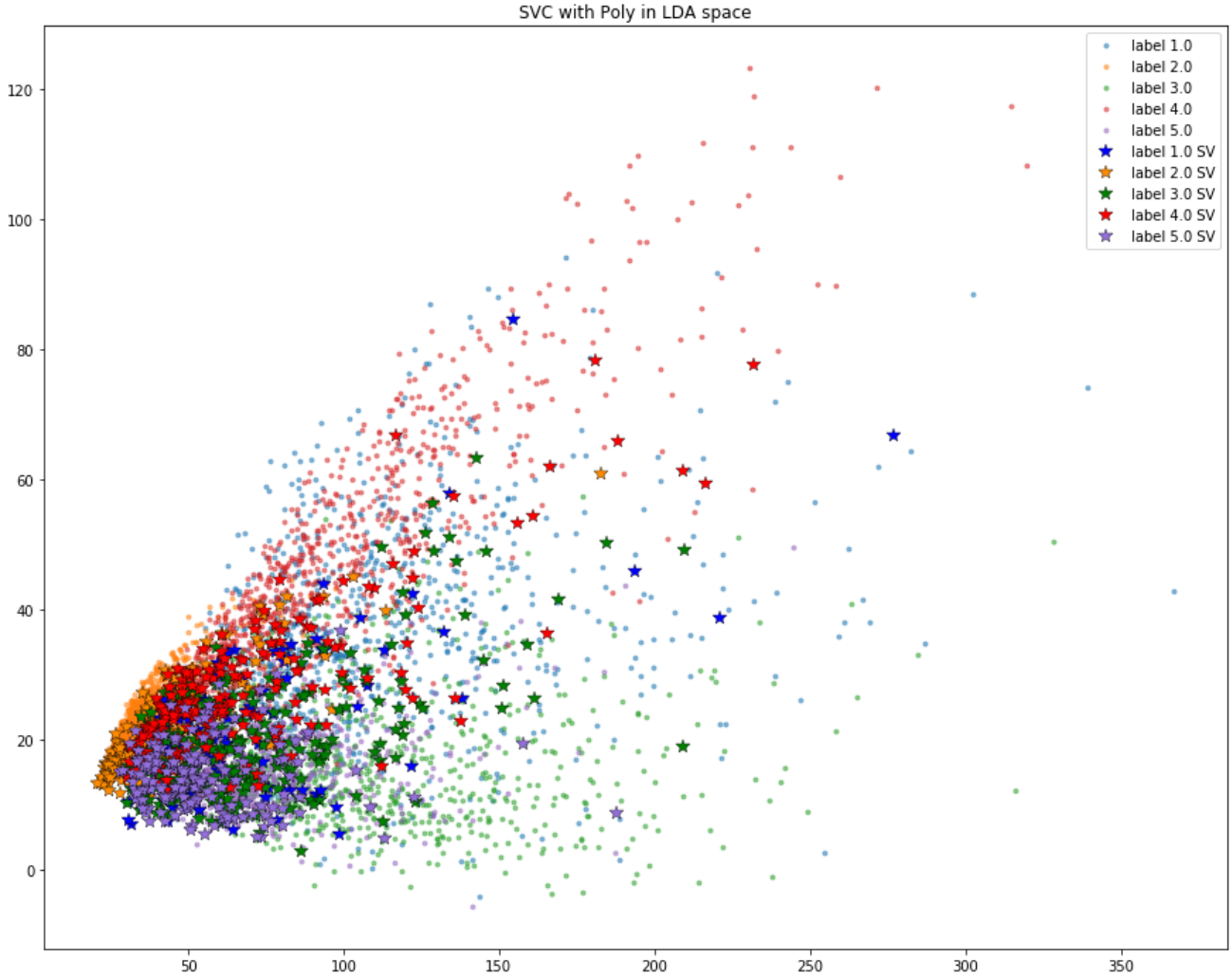


In LDA space

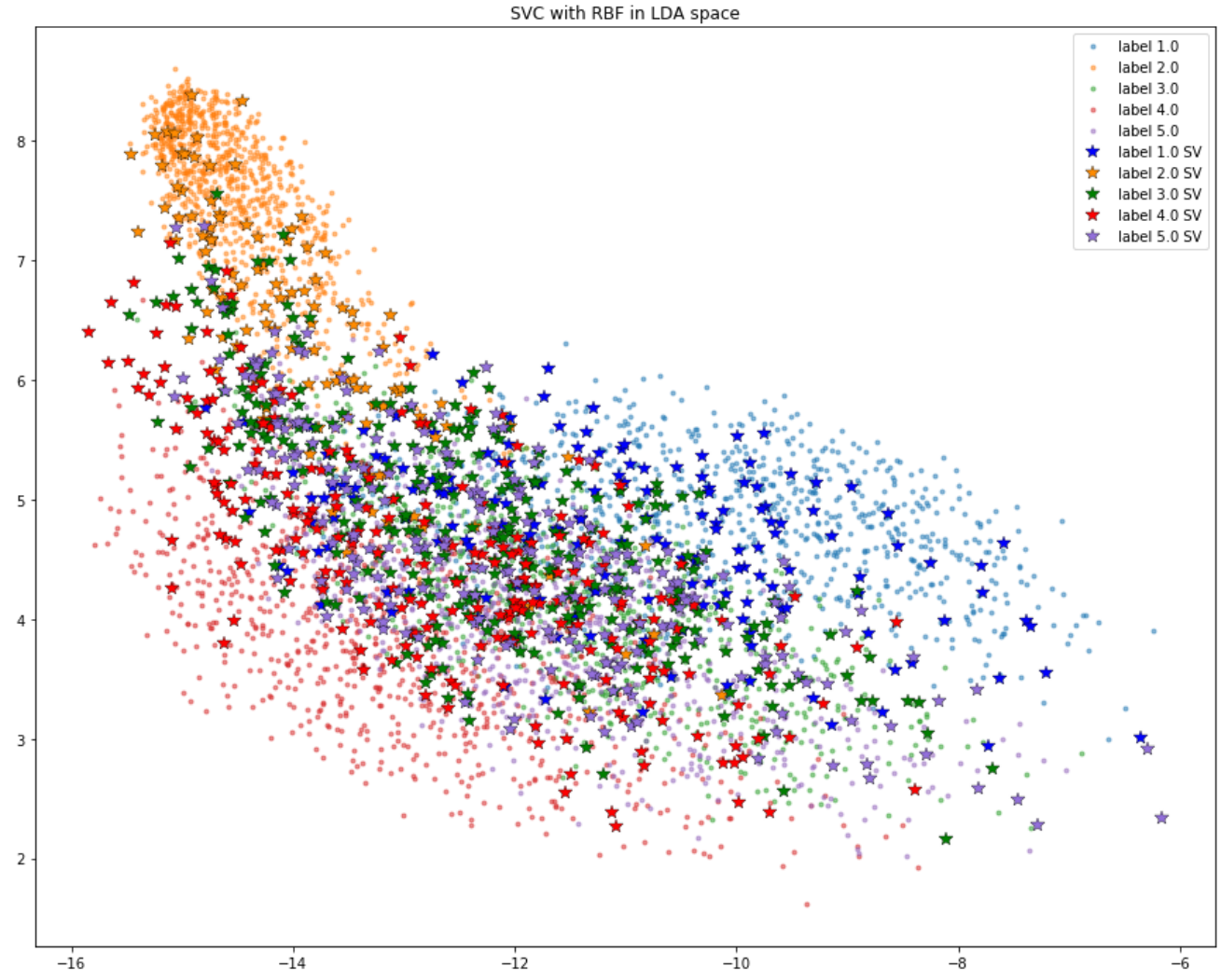
- Linear


SVC with Linear in LDA space
label 1.0
label 2.0
label 3.0
label 4.0
label 5.0
label 1.0 SV
label 2.0 SV
label 3.0 SV
label 4.0 SV
label 5.0 SV
60
40
20
0
−20
−40
−175
−150
−125
−100
−75
−50
−25
0
25

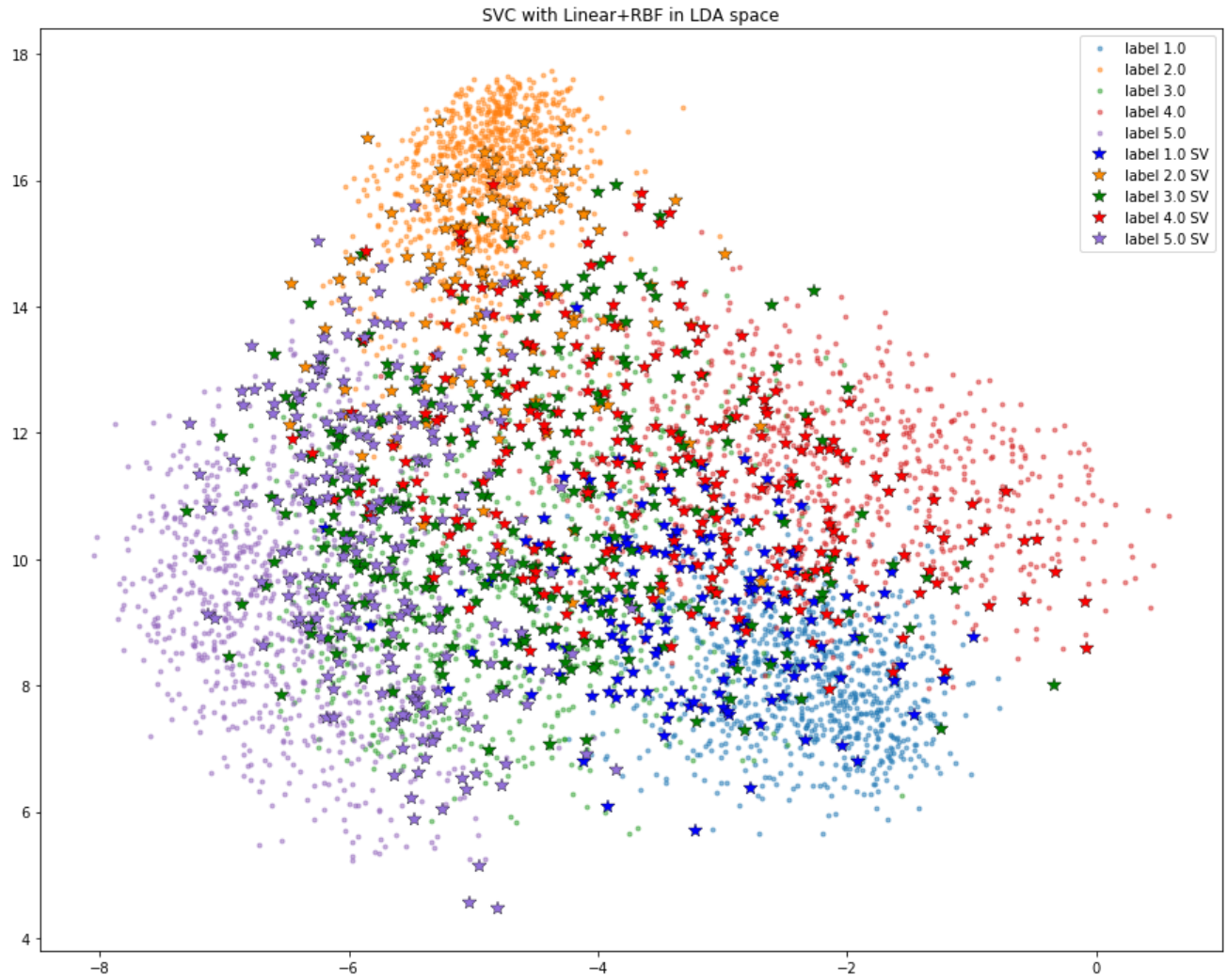
- Polynomial


SVC with Poly in LDA space
label 1.0
label 2.0
label 3.0
label 4.0
label 5.0
label 1.0 SV
label 2.0 SV
label 3.0 SV
label 4.0 SV
label 5.0 SV
120
100
80
60
40
20
0
50
100
150
200
250
300
350

- RBF


SVC with RBF in LDA space
label 1.0
label 2.0
label 3.0
label 4.0
label 5.0
label 1.0 SV
label 2.0 SV
label 3.0 SV
label 4.0 SV
label 5.0 SV
8
7
6
5
4
3
2
−16
−14
−12
−10
−8
−6

- Linear+RBF



my observation

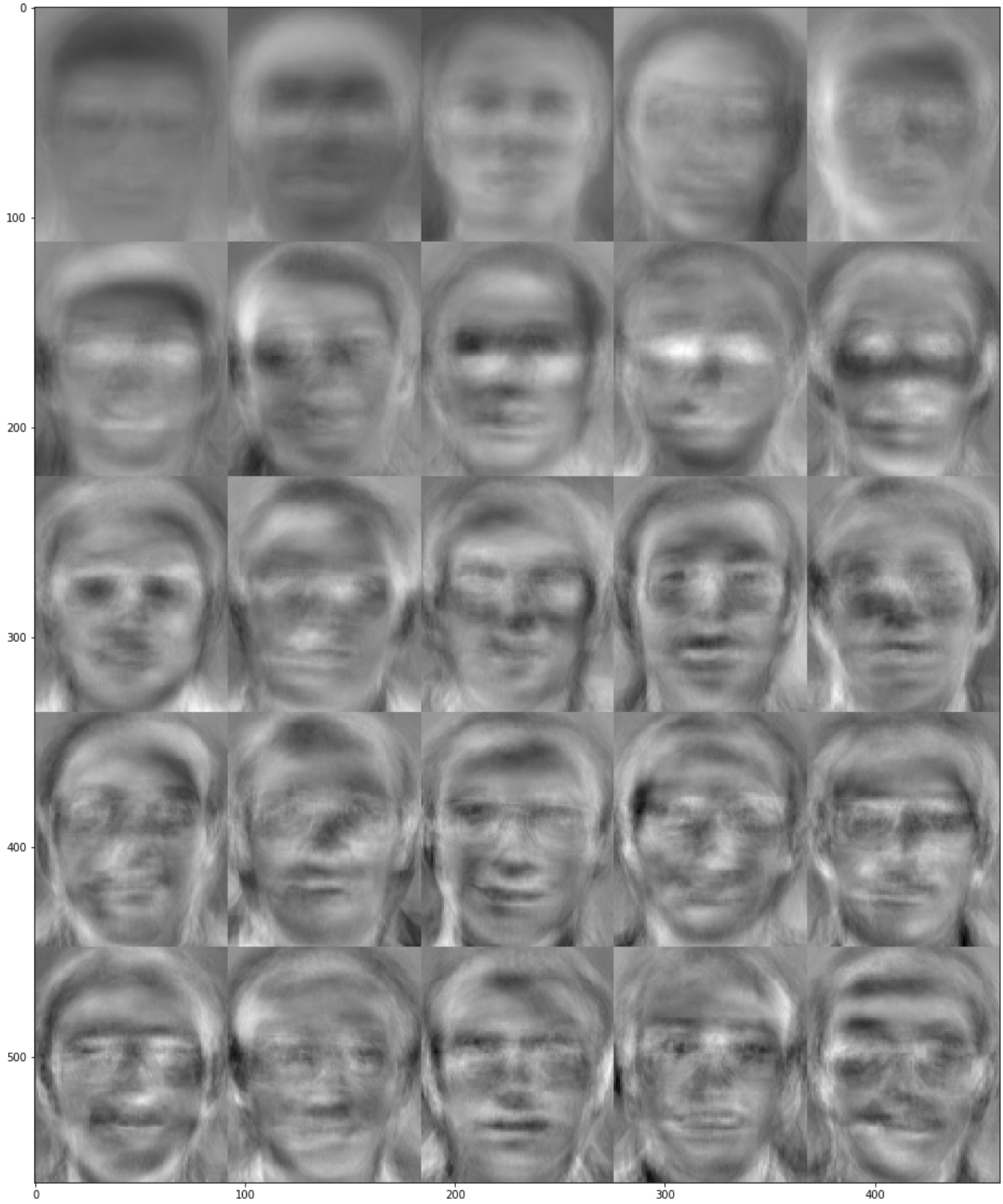
eigenfaces from `att_faces`

先將人臉圖像讀進來存成 `nparray` 方便後續處理

```
import PIL.Image
att_faces_data = []
for i in range(1,41):
    for j in range(1,11):
        with PIL.Image.open(
        './att_faces/s{}/{}.pgm'.format(i, j)
        ) as im:
            att_faces_data += [np.array(im).reshape(1,-1)]

att_faces_data = np.concatenate(att_faces_data, axis = 0)
```

`att_faces_data` 為 400x10304 的矩陣 (400 張人臉，一張 112x92)
使用 PCA 找到 25 個主成份也就 eigenface (特徵臉)

```
face_pcaspace, face_eigen, face_eva, face_evc = PCA(att_faces_data, 25)
showFaces(face_eigen.T, col=5)
```



並且將原本的人臉投影到特徵臉在投影回來 $\text{face} * \text{eigenfaces} * \text{eigenfaces}^T$
只隨機挑選 10 位人臉的不同樣子 (表情，眼鏡穿戴與否，臉轉動等)

```
# random pick subject
# random pick subject
random_faces = np.concatenate([
    att_faces_data[sj*10:(sj+1)*10]
    for sj in np.random.randint(0,40, size=10)]
, axis=0)

showFaces(random_faces)
showFaces(np.matmul(
    np.matmul(random_faces, face_eigen),
    face_eigen.T)
)
```

左邊為原本的人臉，右邊投影到 eigenfaces 上再重建回來

my observation

結果還蠻有趣的，如果觀察 eigenfaces 會發現有人臉的輪廓
同時也感覺得到有不同五官出現，只是較為模糊
找到這樣特徵臉可以幫助我們做人臉辨識，可以將 112x92 共 10340 維度的資料
弄到只有 25 維，所代表含意是在 25 張特徵臉成份多寡

但把 400 張人臉 (40 個人) 投影到 25 張特徵臉是否會導致每個人的獨特特徵完全消失
導致只能辨識這是人臉但無法辨識這是誰的臉?

這邊可以觀察重構後的人臉(投影到特徵臉空間在投影回原本的資料空間)
可以發現右邊重構的人臉還是能夠看出是不同人，只是清晰的五官都模糊了
代表在特徵臉空間的 25 維向量是可以拿來做分類並達到 身份辨識

而且除此之外，也有一定的容錯空間，每個人有十張不同姿態的人臉
有沒表情，有表情，有眼鏡沒眼鏡，傾斜等等
但可以看到重構後同一個人的十張臉幾乎都長的差不多
這也代表在特徵臉空間同一個人不同姿態的向量不會距離太遠

藉由這樣的例子也可以感受到 PCA 的厲害
找出該資料的主要成分，並藉由該成份去表達原本資料
同時去除無用資料與雜訊只關注我們想關注的主要特徵