2018 ML HW6 Clustering

0756110 李東霖

This document made by HackMD, you can view here https://hackmd.io/s/rJ-O49ikE (https://hackmd.io/s/rJ-O49ikE)

- 2018 ML HW6 Clustering
  - k-means / kernel k-means
    - objective
    - steps
    - my implementation
    - run k-means / kernel k-means
    - clustering procedure
    - my observation
  - spectral clustering
    - steps & my implementation
    - spectral clustering result
    - my observation
    - discuss eigen space of graph laplacian
  - different initialization method
    - diff gamma for kernel k-means
    - diff gamma for spectral clustering

k-means / kernel k-means

因為 `k-means` 與 `kernel k-means` 整個行為非常相似，所以這邊放在一起去解釋與說明

objective

目標都是最小化屬於該群的資料點與群中心的距離

steps

- 初始化 initialization

  決定要分成 k 群，並假設 k 群中心

k-means
可以直接假設群中心 u_k，預設使用隨機去取得

kernel k-means
因為群中心 u_k 牽涉到 feature space 的轉換
因此轉而假設每個資料點的群 $c_{ik} = \begin{cases} 1, & x_i \in C_k \\ 0, & x_i \notin C_k \end{cases}$

- 估計 E-step

使用當前的群假設去算所有資料點與群的距離
得到 w (`number of data points x number of cluster`)

k-means
使用歐基里德距離 $w_{ik} = ||x_i - u_k||^2$ 去計算第 i 個資料到第 k 群的距離

kernel k-means

先投影資料到 feature space 再使用歐基里德 $w_{ik} = ||\phi(x_j) - u_i^\phi||^2, u_i^\phi = \frac{\sum_{x_i \in C_k} \phi(x_i)}{|C_k|}$
因為 $\phi(x)$ 難以推導與得知，因此使用 *kernel trick* 來解決這個問題

$$||\phi(x_i) - u_k^\phi||^2 = ||k(x_i, x_i) - \frac{2}{|C_k|} \sum_{x_j \in C_k} k(x_i, x_j) + \frac{1}{|C_k|^2} \sum_{x_p \in C_k} \sum_{x_q \in C_k} k(x_p, x_q)||$$

如此一來就只需要計算 $k(x, x)$ 就能得到在 feature space 中資料點與群中心的距離

- 優化 M-step

藉由 $w \in R^{n \times k}$ 與目標函數 $\min_{C_1 \dots C_k} \sum_{q=1}^{k} \sum_{x_p \in C_q} \text{distance}(x_p, u_q)$
我們知道要盡量把第 i 個資料點分進距離最小的群 $argmin_j\ (w_{ij})$
最終得到新的分群 $c'_{ik} = \begin{cases} 1, & x_i \in C_k \\ 0, & x_i \notin C_k \end{cases}$

k-means
藉由 c' 更新 $u, u_k = \frac{\sum_i^n c'_{ik} x_i}{\sum_i^n c'_{ik}}$

kernel k-means
只需要更新 $c, c = c'$

- 收斂 converge

當每次迭代得到的新分群 c' 與之前分群沒有改變 $c' == c$ 即是達到收斂

my implementation

```
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 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:np.exp(-1*gamma*Euclidean(u,v));

def KernelTrick(gram_m, c):
    """
    calculate Euclidean distance in feature space by kernel trick
    parameters:
        gram_m : gram matrix K(x, x)
        c : cluster vector c(i,k) = 1 if x(i) belong k clustering
    return:
        w : n*k
    """
    return (
        np.matmul(
        gram_m * np.eye(gram_m.shape[0]),
        np.ones((gram_m.shape[0], c.shape[1]))
    ) \
    - 2*( np.matmul(gram_m, c) / np.sum(c, axis=0) ) \
    + (np.matmul(
        np.ones((gram_m.shape[0], c.shape[1])),
        np.matmul(np.matmul(c.T, gram_m), c)*np.eye(c.shape[1])
    ) / (np.sum(c,axis=0)**2) )
    )

def RandomCluster(n,k):
    """
    get random cluster c
    """
```

```
    """
    c = np.zeros((n,k))
    c[np.arange(n),np.random.randint(k,size=n)] = 1
    return c

def RandomMean(k,dim):
    """
    get random mean
    """
    return -1 + 2*np.random.random((k,dim))

def GetMeanFromCluster(datas, c):
    """
    get mean from cluster c
    """
    if np.count_nonzero(np.sum(c, axis=0) == 0):
        raise ArithmeticError
    return np.matmul(c.T, datas) / np.sum(c, axis=0)[:,None]

def GetCluster(w):
    """
    get cluster from w (distance between x and u)
    """
    new_c = np.zeros(w.shape)
    new_c[np.arange(w.shape[0]),np.argmin(w, axis=1)] = 1
    return new_c

def kmeans(datas, k,
           initial_u=None,
           initial_c=None,
           isKernel=False,
           converge_count = 1
          ):
    """
    use generator to iter steps
    parameters:
        datas : data points
        k : how many cluster
        initial_u : assign mean u
        initial_c : assign cluster c
        isKernel : use kernel k-means (default is False)
    return:
        python generator

    next() will get cluster c and mean u
    e.g.
        c, u = next(g)
    if converge, next() will raise Error
    """

    # initialization
    if isKernel:
        gram_matrix = datas
        c = initial_c if type(initial_c)!=type(None) \
            else RandomCluster(datas.shape[0], k)
```

```
    else:
        u = initial_u if type(initial_u)!=type(None) \
            else RandomMean(k,datas.shape[1])
        c = np.zeros((datas.shape[0],k))

    while(1):
        # E-step
        if not isKernel:
            w = Euclidean(datas, u)
        else:
            w = KernelTrick(gram_matrix, c)

        # M-step
        update_c = np.zeros(w.shape)
        update_c[np.arange(w.shape[0]),np.argmin(w, axis=1)] = 1

        delta_c = np.count_nonzero(np.abs(update_c - c))
        if not isKernel:
            u = GetMeanFromCluster(datas, update_c)
        else:
            u = None

        yield update_c, u

        if delta_c == 0:
            converge_count-=1
            # find converge
            if converge_count == 0:
                break

        c = update_c

    return
```

幾個關鍵地方解釋

- kernel trick

  不使用 for 迴圈，將 kernel trick 改寫成 numpy 運算
  $matmul$ 為 矩陣乘法，$*$ 為 分量乘法(一個元素對一個元素乘)

$$matmul(k(D,D) * eye(n), ones((n,k)))$$
$$-2 * \frac{matmul(k(D,D),C)}{sum(C)} + \frac{matmul(ones((n,k)),(C^T k(D,D) C) * eye(k))}{sum(C)^2}$$

  D 為 all data points ，$k(D,D)$ 為 gram matrix ，C 為 cluster matrix
  程式碼實作詳見上方

run k-means / kernel k-means

```
def runKmeans(launcher):
    iter_count = 0
    all_c = []
    all_u = []
    for new_c, new_u in launcher:
        IDisplay.clear_output(wait=True)
        iter_count += 1
        showClustering(data_source,
                       new_c, new_u,
                       title='iter [{}]'.format(iter_count)
                      )
        all_c.append(new_c)
        all_u.append(new_u)

    print('use {} counts to converge'.format(iter_count))

    return all_c, all_u
```

因為我的迭代方式是使用 python 的 generator 來實作
可以直接使用 for 去得到每一次迭代新計算出來的 分群 c 與 群中心 u
並在取得之後使用 `matplotlib` 去顯示出來
最後會將每一步迭代的 c 和 u 保存起來，以方便之後的再次使用

k-means

```
# initial setting
data_source = circle
k = 2

launcher = kmeans(data_source, k)
all_kmean_c, all_kmean_u = runKmeans(launcher)
```

kernel k-means

```
# initial setting
data_source = moon
k = 2
kernel = RBFkernel(80)
gram_matrix = kernel(data_source, data_source)

launcher = kmeans(gram_matrix, k, isKernel=True)
all_kkmean_c, all_kkmean_u = runKmeans(launcher)
```

給予好初始設定之後，就可以開始分群演算法
跑到收斂即是該演算法在該初始設定下所能找到的最優解

clustering procedure

這邊使用 `matplotlib` 的 `animation` 去呈現動畫並將之存成 `mp4`
因為 kernel k-means 不好找到群中心，因此只有分群情況

- k-means
  - moon
    - k=2
      kmeans data=moon k=2 init=random (https://youtu.be/oUiwxO8XxYU)
    - k=3
      kmeans data=moon k=3 init=random (https://youtu.be/cINW-ZhWvMA)
    - k=4
      kmeans data=moon k=4 init=random (https://youtu.be/d4WaNa9hDGY)
  - circle
    - k=2
      kmeans data=circle k=2 init=random (https://youtu.be/1_Sxi6CN0MQ)
    - k=3
      kmeans data=circle k=3 init=random (https://youtu.be/HH8vafgio5Q)
    - k=4
      kmeans data=circle k=4 init=random (https://youtu.be/w62gltqA1bI)
- kernel k-means
  - moon
    - k=2
      kernel kmeans data=moon k=2 init=random (https://youtu.be/dytmn38TdLo)
    - k=3
      kernel kmeans data=moon k=3 init=random (https://youtu.be/S6YbzGmA9dU)
    - k=4
      kernel kmeans data=moon k=4 init=random (https://youtu.be/gCyxvYVaGq8)
  - circle
    - k=2
      kernel kmeans data=circle k=2 init=random (https://youtu.be/0lkshGj7Epc)
    - k=3
      kernel kmeans data=circle k=3 init=random (https://youtu.be/ylxPcx9pZvI)
    - k=4
      kernel kmeans data=circle k=4 init=random (https://youtu.be/kDccw6dMAsQ)

my observation

k-means 在處理 `moon` 與 `circle` 這兩個資料集都無法得到好結果
但是視覺化可以很明顯呈現找群中心的感覺，也因為是直接使用這二維空間去計算距離
我直接看可以完全理解為什麼

在 kernel k-means 比起 k-means 有好一點，至少可以跳脫當前的空間
嘗試從不同空間去計算資料點之間的距離
也因此我直接看並無法知道為什麼會這樣分群

spectral clustering

這邊實作的版本是 unnormalized 的
下面同時說明步驟與程式碼實作

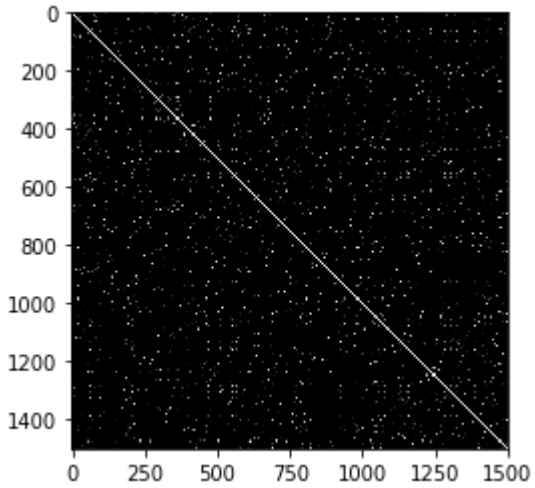
steps & my implementation

- 初始化

```
# initial setting
data_source = circle
kernel = RBFkernel(80)
k = 2
```

- 找到相似度矩陣，當作 $W, W_{v,u} = kernel(v, u)$

```
similarity_matrix = kernel(data_source, data_source)
showGram(similarity_matrix)
W = similarity_matrix
```



- 計算 degree $d_v = \sum_{u \in V} W_{vu}$
  產生特殊矩陣 $D, D_{v,u} = d_v \delta_{vu}, \delta_{vu} = \begin{cases} 1, & u = v \\ 0, & u \neq v \end{cases}$
  也就是只有 軸元 上才有數值，其他地方皆為 0

```
8  D = np.sum(W, axis=1)*np.eye(W.shape[0])
```

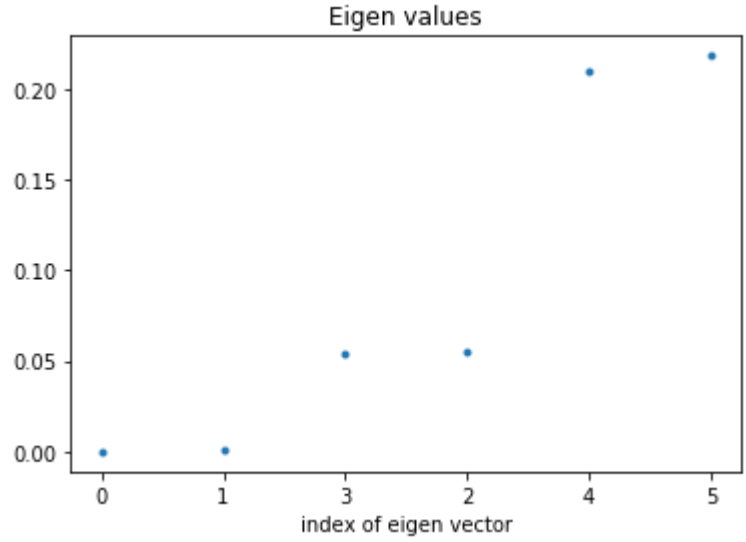
- 取得 graph laplacian $L = D - W$
  計算 graph laplacian 的特徵分解，取得 n 個 eigen values 和 eigen vectors
  要取出第 i 個 eigen vector 是 `eigen_vectors[:,i]`

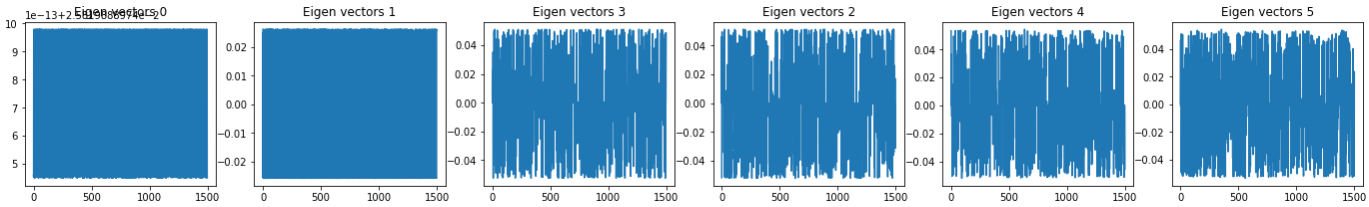
```
9  L = D - W
10 eigen_values, eigen_vectors = np.linalg.eig(L)
```

- 使用前 k 個 eigen value 最小的 eigen vectors
  組成新的 eigen space 去表示原本的資料

$$ev_i \in R^{n \times 1}$$

$$U = [ev_0\ ev_1\ \dots\ ev_k] \in R^{n \times k}$$





```
# generate new vector space to present origin data
# if eigen vector is fully connect ,
# we need to drop it and use next one

sorted_idx = np.argsort(eigen_values)

U = []
use_eigen_idx = []
current_i = 0
current_k = 0

while current_k < k:
    evc = eigen_vectors[:,sorted_idx[current_i]]
    if True:
    #if (np.var(evc) > 10**-20):
        U.append(evc[:,None])
        use_eigen_idx.append(sorted_idx[current_i])
        current_k += 1

    current_i += 1

U = np.concatenate(U, axis=1)
```

這邊本來打算把沒有鑑別度也就是 fully connect 的給 drop 掉
但很難拿捏那個 variance 的 threshold 最後就放棄了

- 使用 eigen space U 去進行 k-means，U_i 得到的分群即為第 i 筆原本資料的群

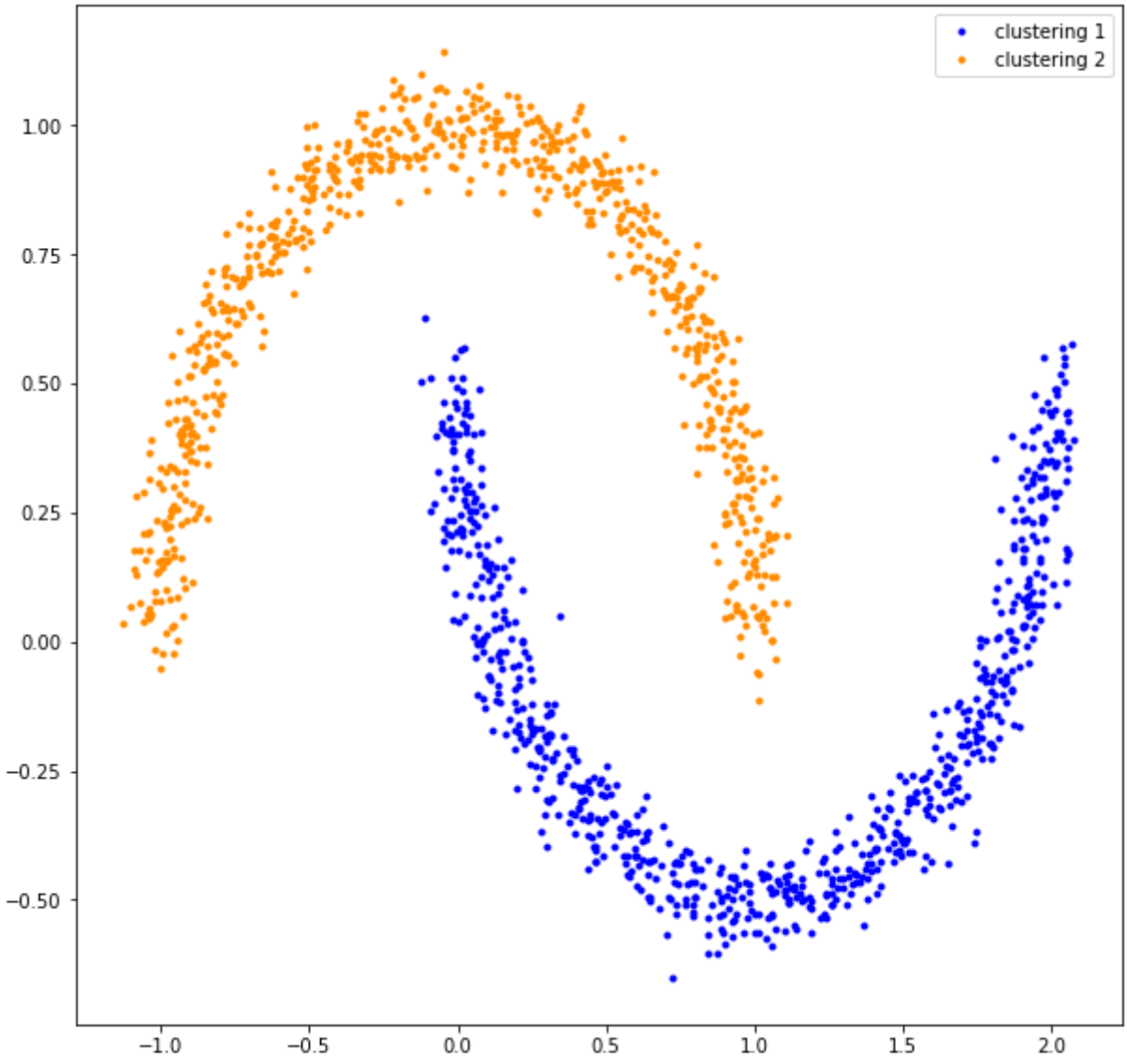
```
launcher = kmeans(U, k,
                  initial_u=GetMeanFromCluster(
                      U,
                      RandomCluster(U.shape[0], k)
                  )
                 )
all_spectral_c, all_spectral_u = runKmeans(launcher)
IDisplay.clear_output(wait=True)
showClustering(data_source, all_spectral_c[-1])
showGram(ReorderGram(W, all_spectral_c[-1]))
```

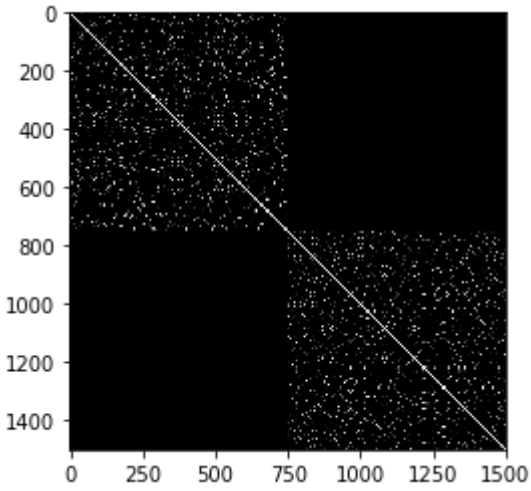
因為 eigen space 中點與點距離小，容易產生有一群沒有任何資料點
因此初始化採用先隨機分群再去找群中心，避免空群

spectral clustering result

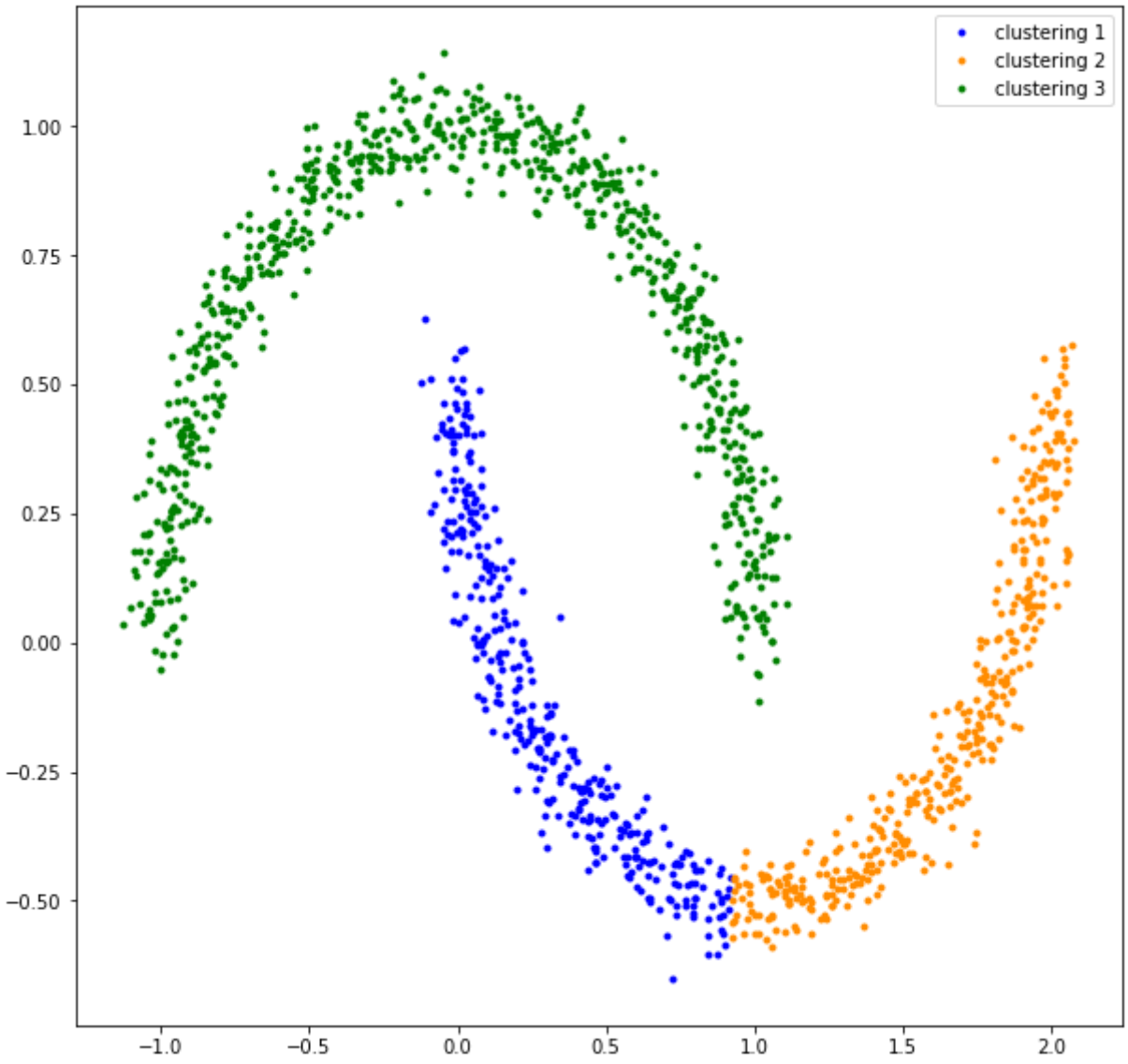
gamma 固定為 80
同時呈現分群結果與重新排列順序的 gram matrix

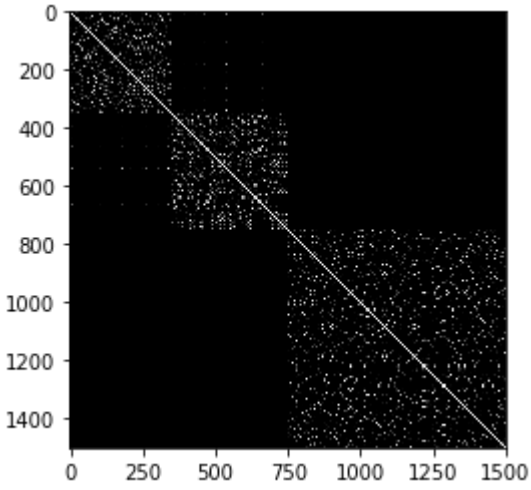
- moon
  - k=2



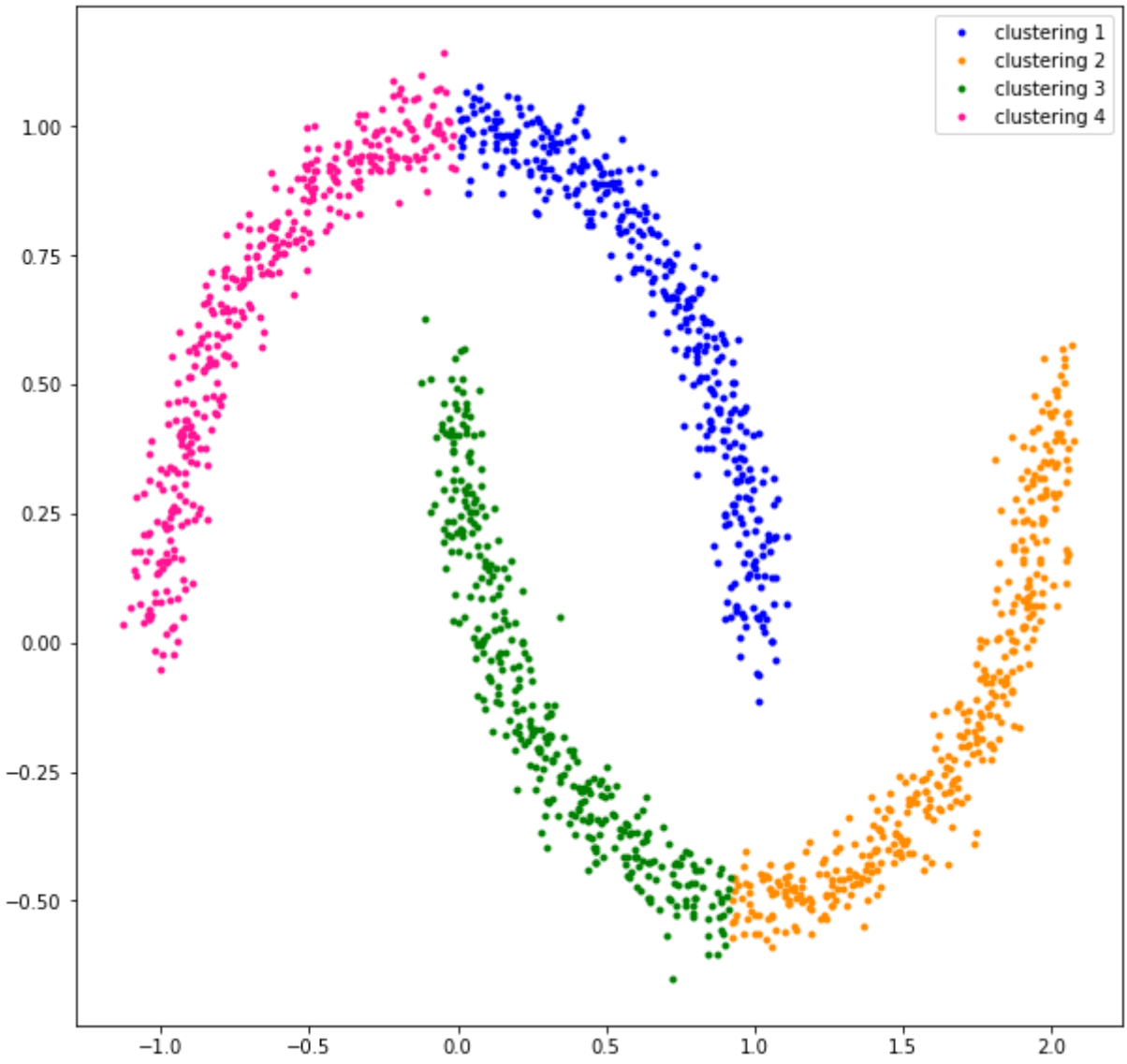


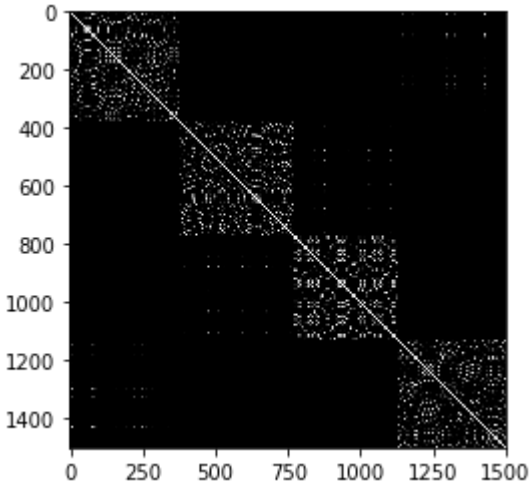
- k=3


clustering 1
clustering 2
clustering 3
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

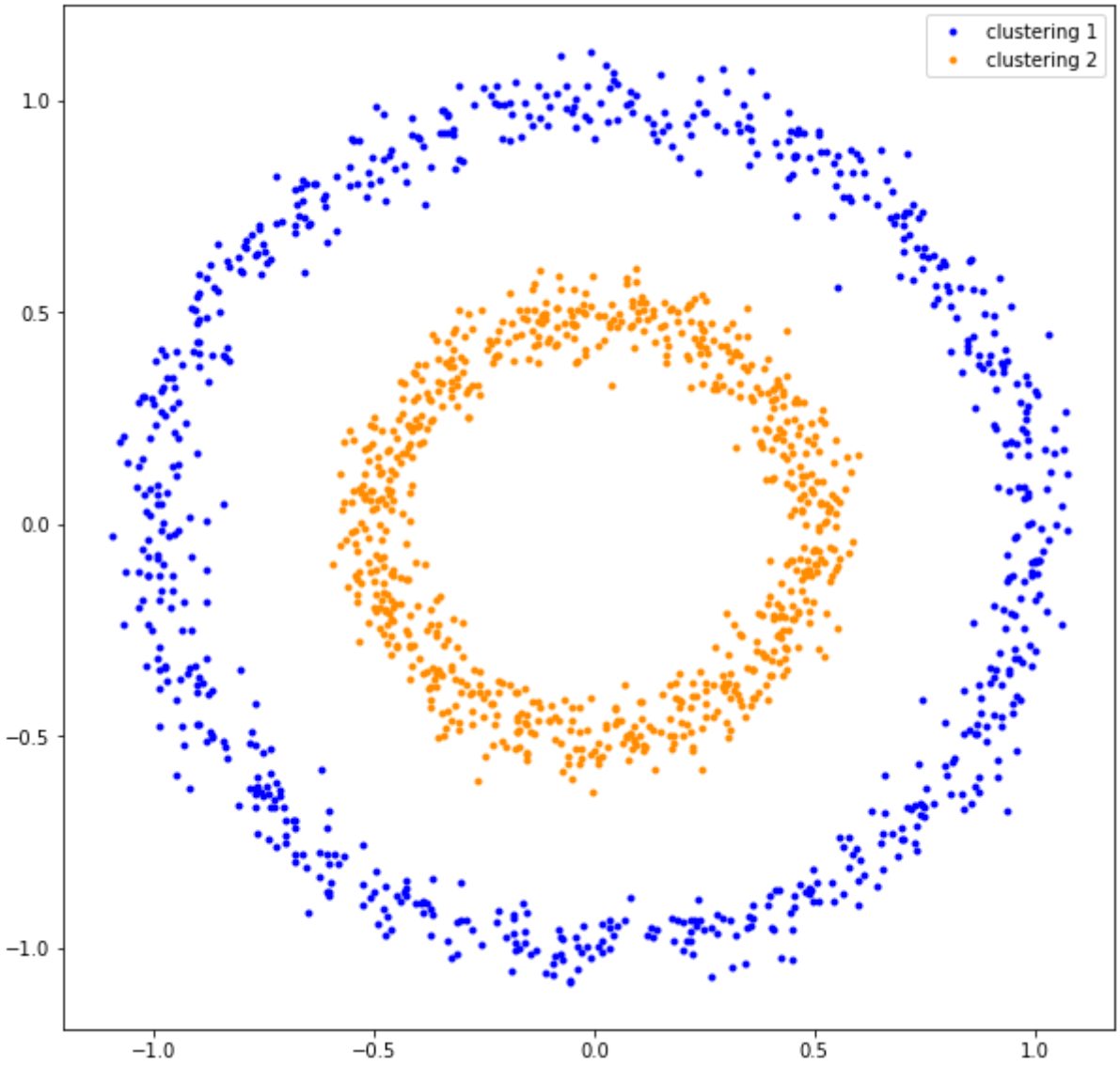

0
200
400
600
800
1000
1200
1400
0
250
500
750
1000
1250
1500

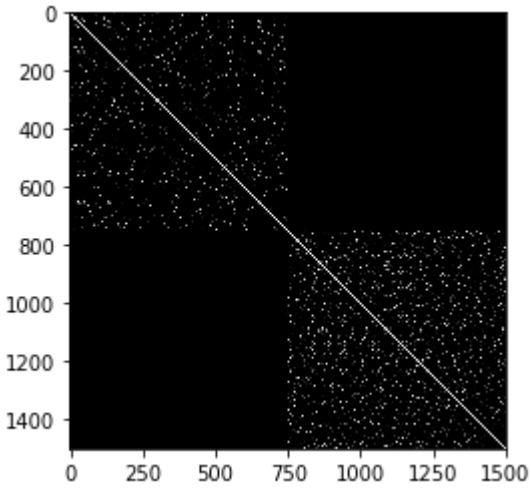
- k=4


clustering 1
clustering 2
clustering 3
clustering 4
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

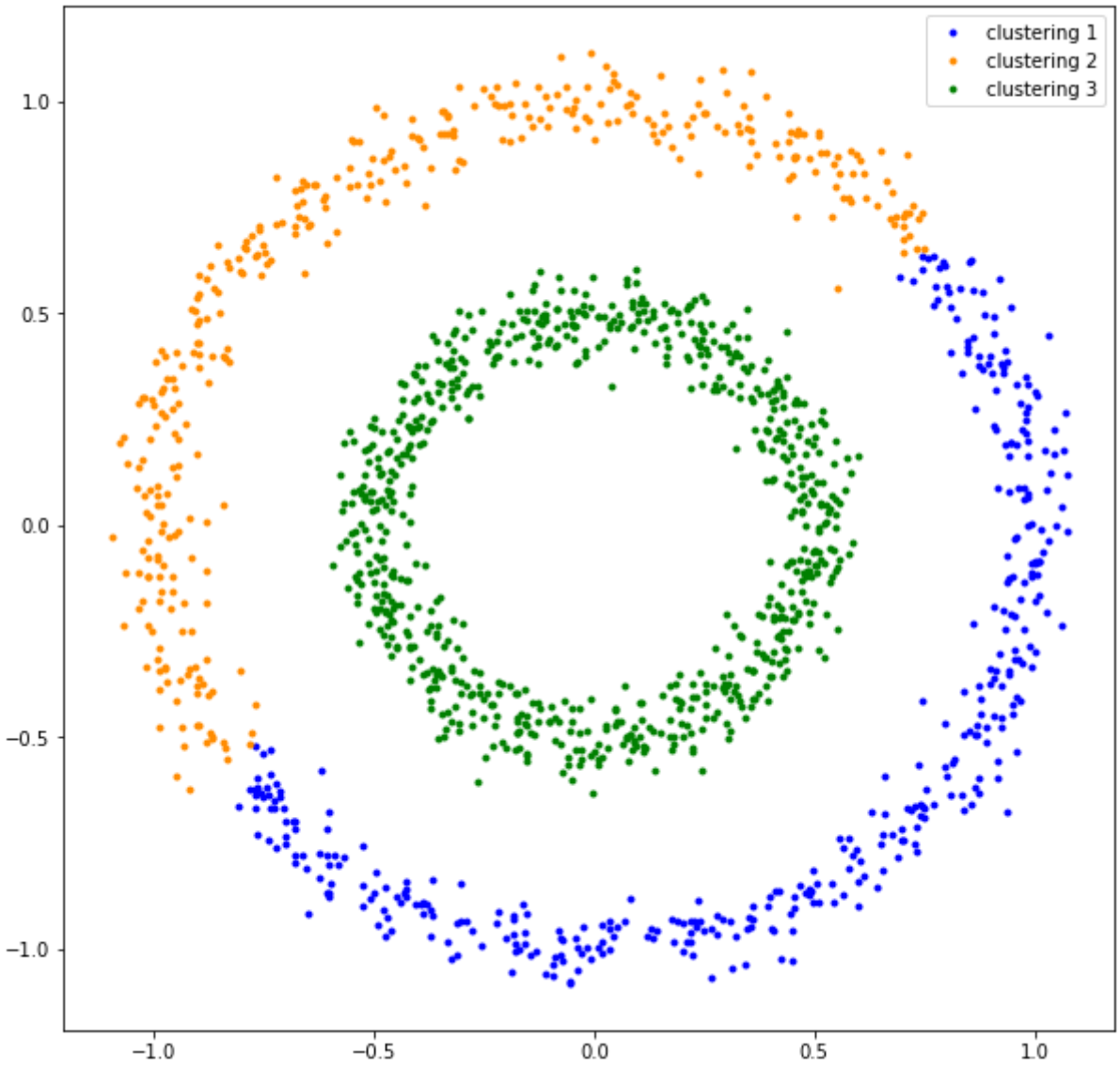

0
200
400
600
800
1000
1200
1400
0
250
500
750
1000
1250
1500

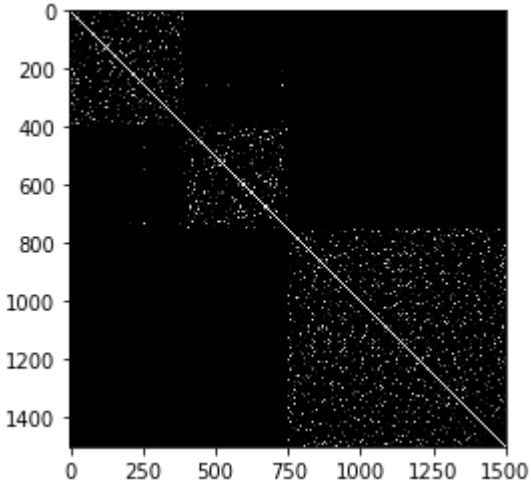
- circle
  - k=2


clustering 1
clustering 2
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

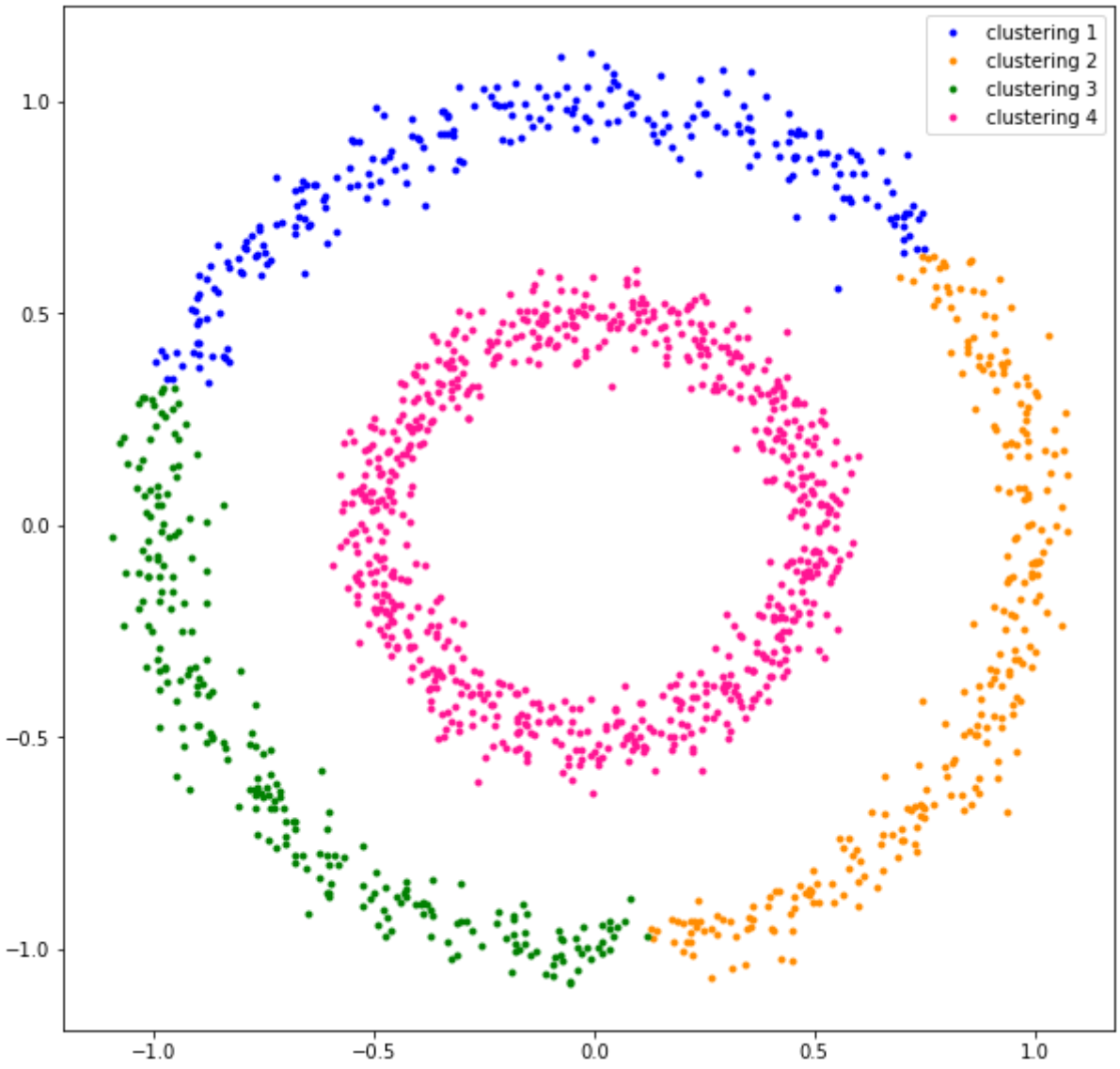

0
200
400
600
800
1000
1200
1400
0
250
500
750
1000
1250
1500

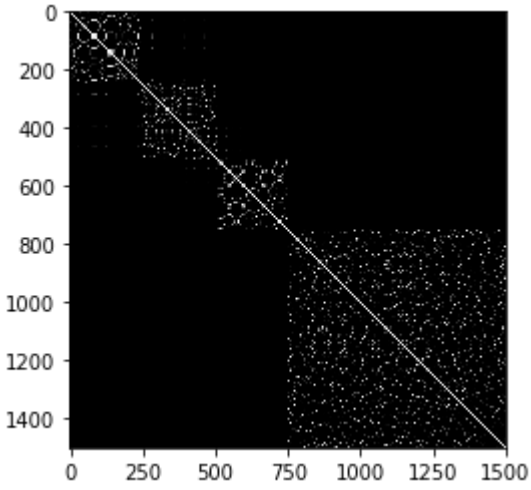
- k=3


clustering 1
clustering 2
clustering 3
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0


0
200
400
600
800
1000
1200
1400
0
250
500
750
1000
1250
1500

- k=4


clustering 1
clustering 2
clustering 3
clustering 4
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0


0
200
400
600
800
1000
1200
1400
0
250
500
750
1000
1250
1500

my observation

剛寫完的時候真的是被驚嘆到， spectral clustring 非常快就找到答案
而且分的非常好，把前面 kernel k-means 與 k-means 無法處理的都解決了

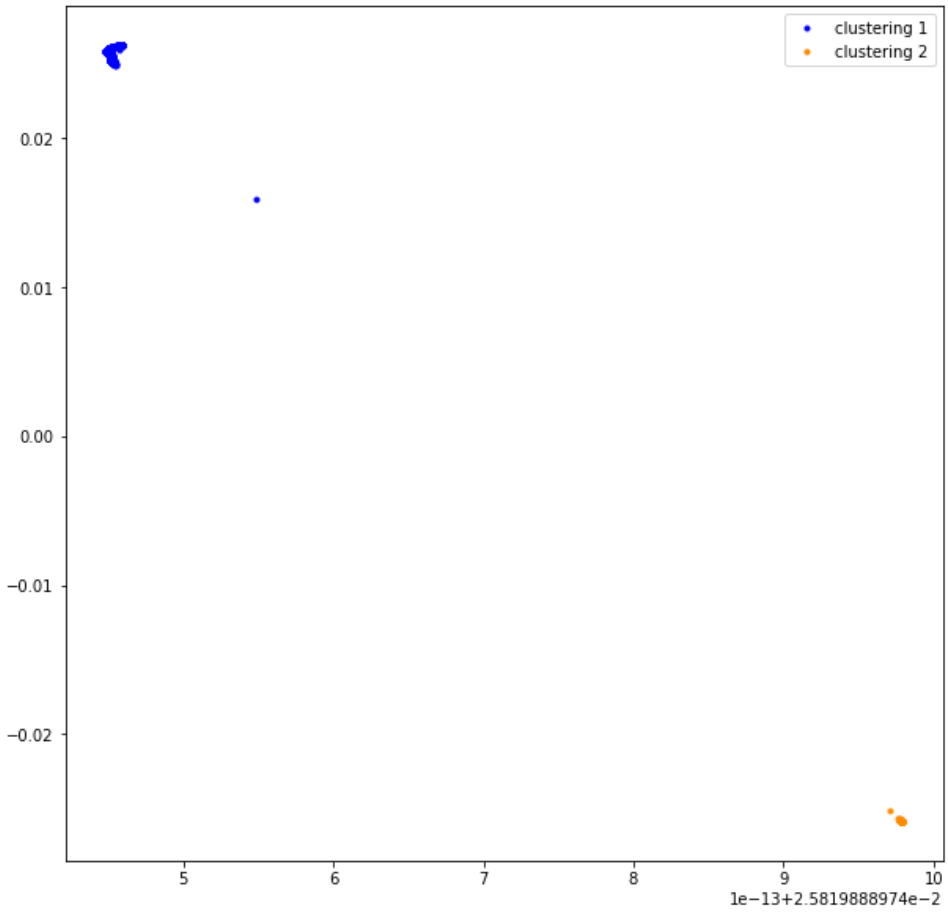
同時觀察 gram matrix 可以看到讓 cut(A,B) 最小化
幾乎是黑黑的一片，代表兩群間的差距大

這也代表 kernel k-means 也可以做到一樣效果
可是初始化太過影響它的結果，很難找到全局最優解

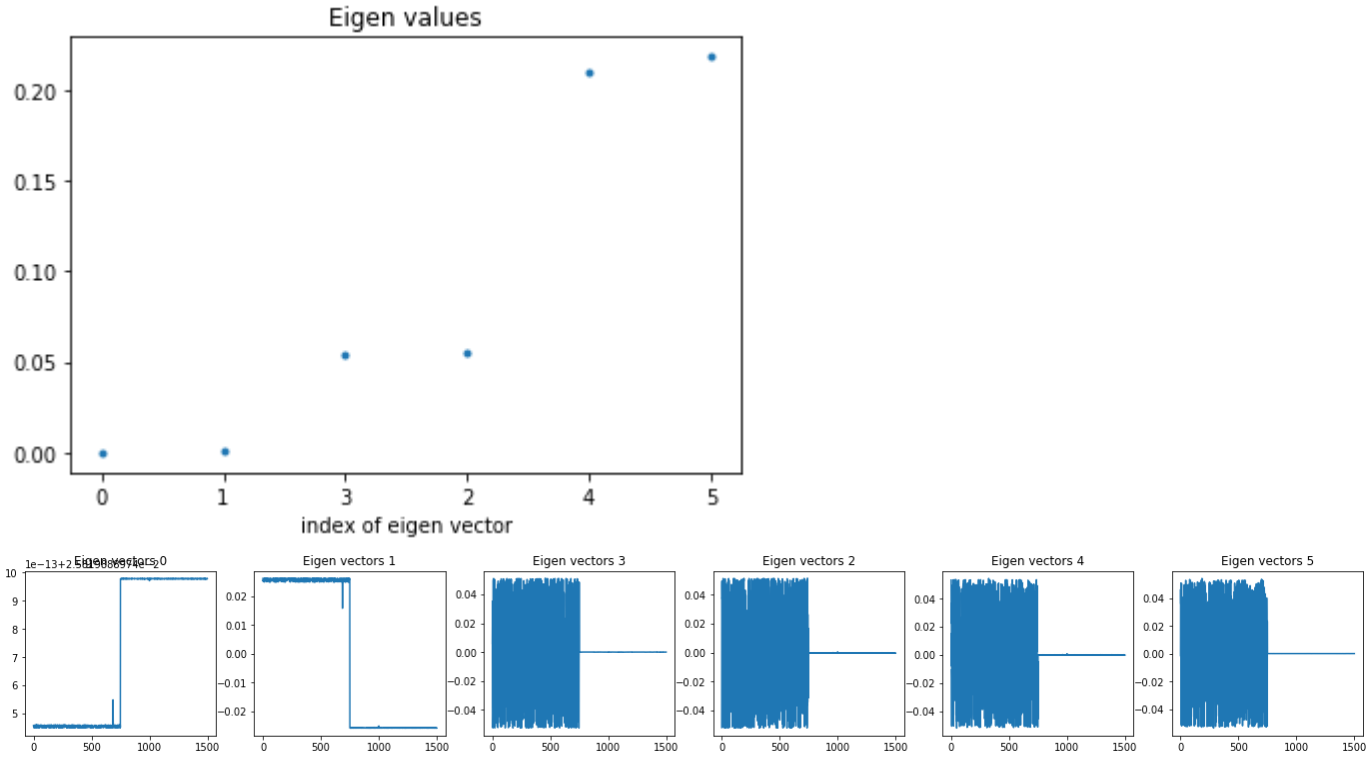
discuss eigen space of graph laplacian

我們先來觀察在 eigen space 是否有同一群都靠近類似座標
這邊使用 circle dataset 與 k=2, gamma=80 的時候

```
showClustering(U, all_spectral_c[-1])
```



可以看到同一群都會擁有相近的座標，這也讓後來的 k-means 可以分出正確的群
而這個 eigen space 是由 eigen vector 組成的，所以來看看 重新排序 後的



可以看到在前兩個 eigen vector 擁有很小的 eigen value
並且在 eigen vector 可以分好的分成兩群，其他較小的 eigen vector 卻沒有這個特性
似乎只能夠給予某一群更多的分群 (這也是為什麼 k=3,4 會只有再被分群)

至於 eigen vector 究竟在 spectral clustering 代表什麼
就需要回到為什麼解出 eigen problem 就可以得到最優解

$$\min cut(A,B) \Rightarrow \min_f f^T Lf, Lf = \lambda f, f^T f = 1 \Rightarrow \min_f f^T \lambda f = \lambda$$

可以發現擁有最小 eigen value 的 eigen vector 代表怎麼分出 A B 群可以使 cut(A,B) 最小
也就是一個 eigen vector 可以幫助分出兩個群

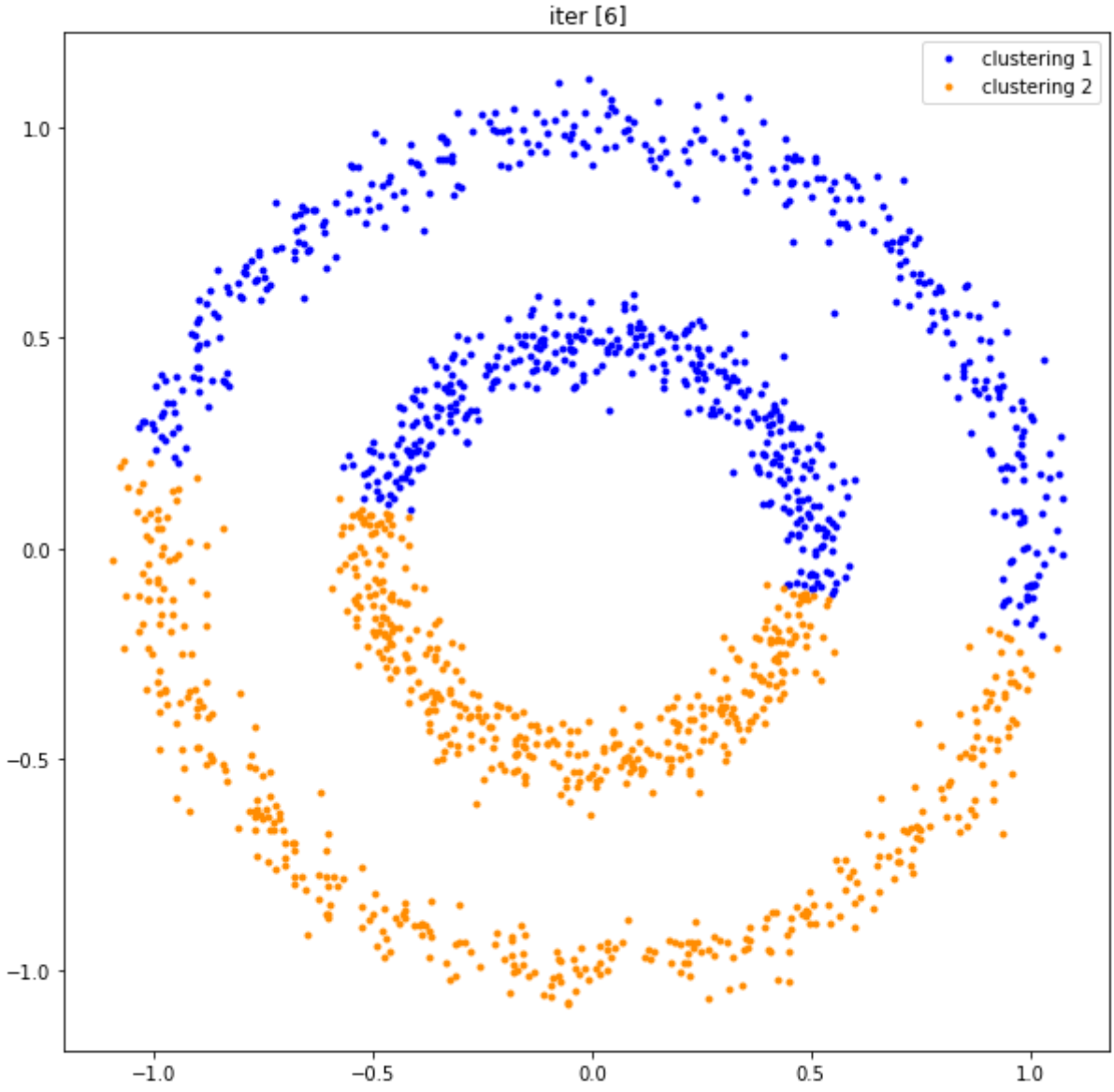
這也最終導致在 eigen space 會讓同一群擁有相似座標

different initialization method

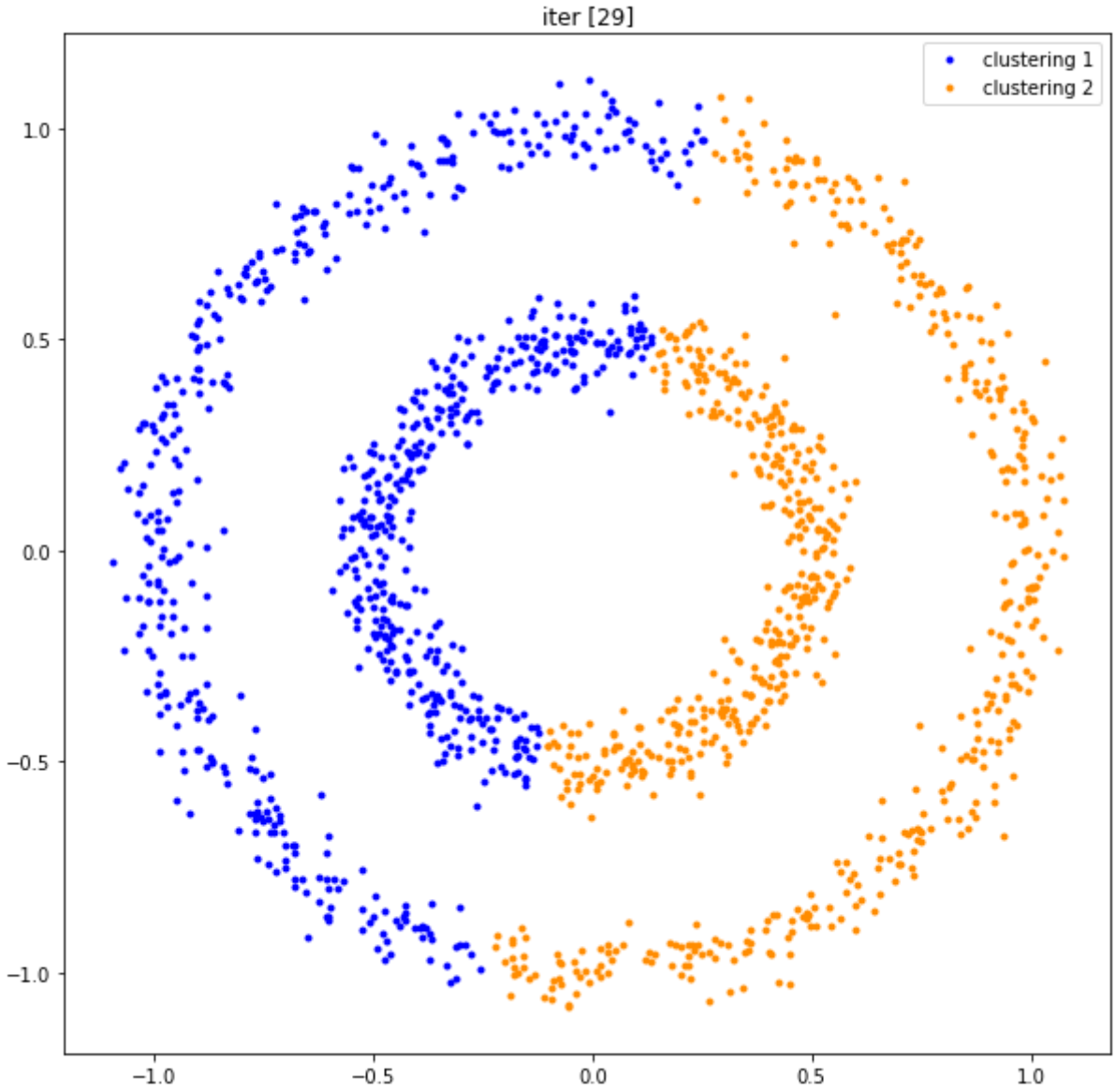
diff gamma for kernel k-means

初始化分群採用隨機，使用 circle dataset, k=2

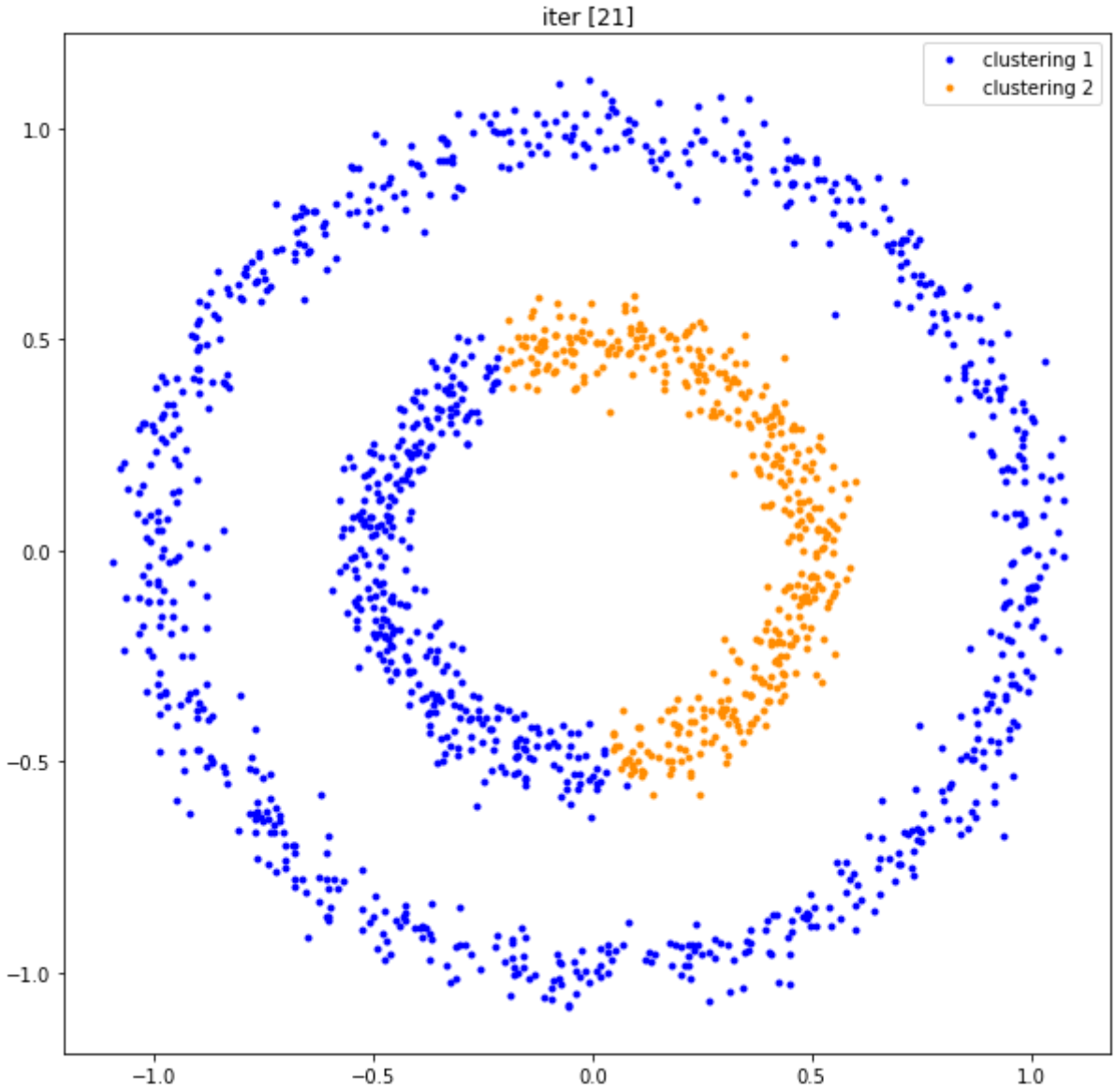
- gamma = 0.1



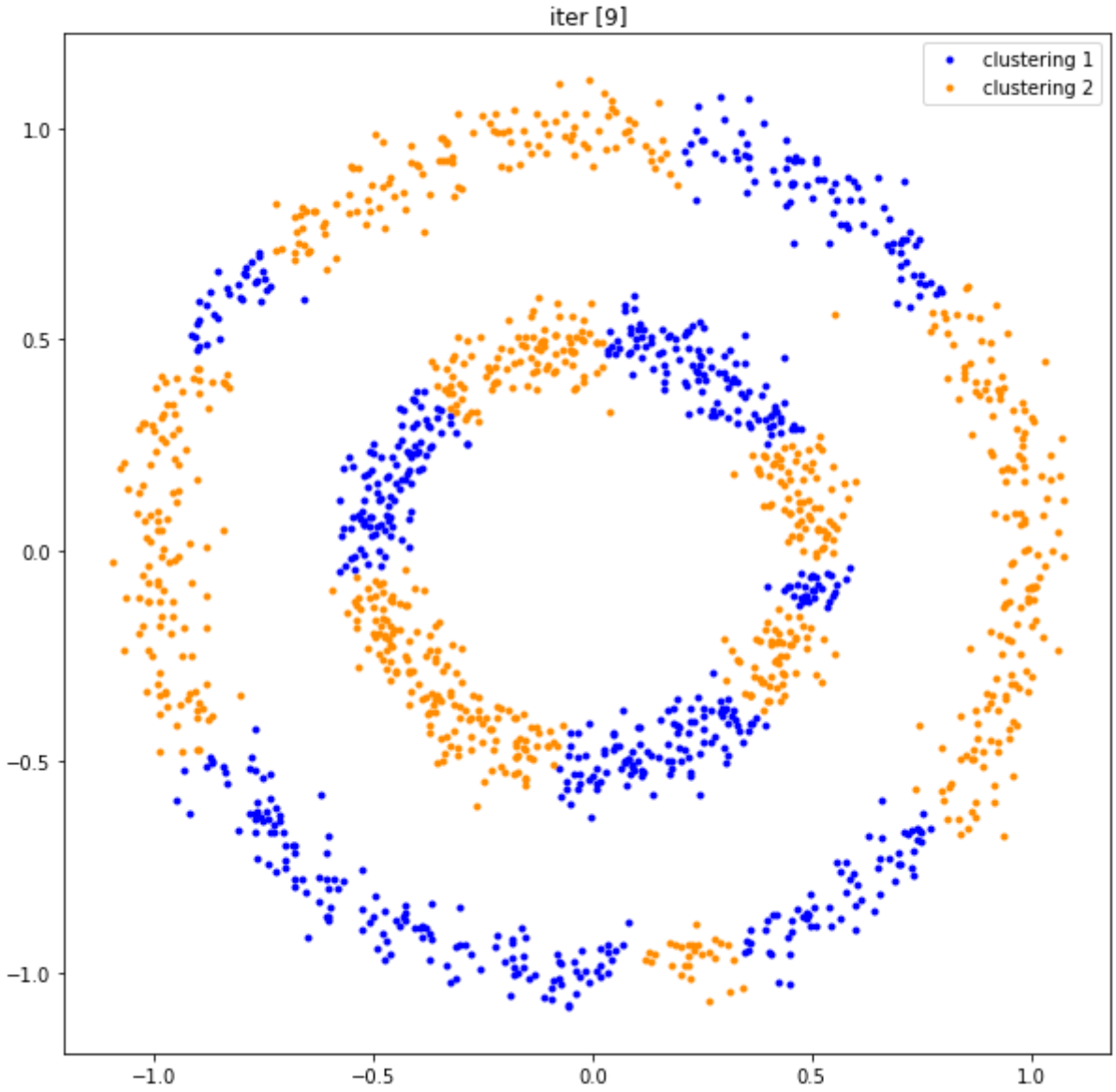
- gamma = 1



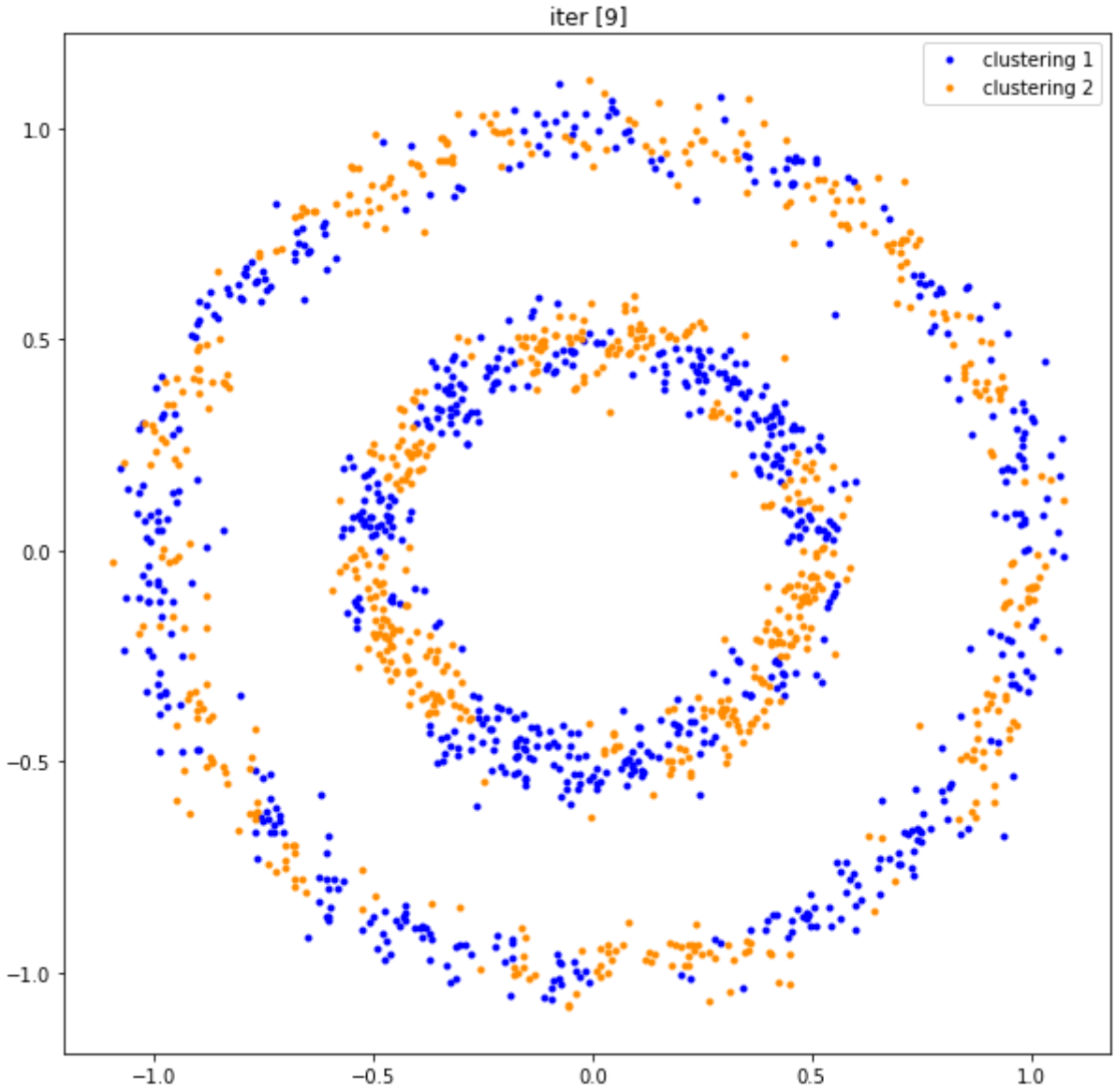
- gamma = 10



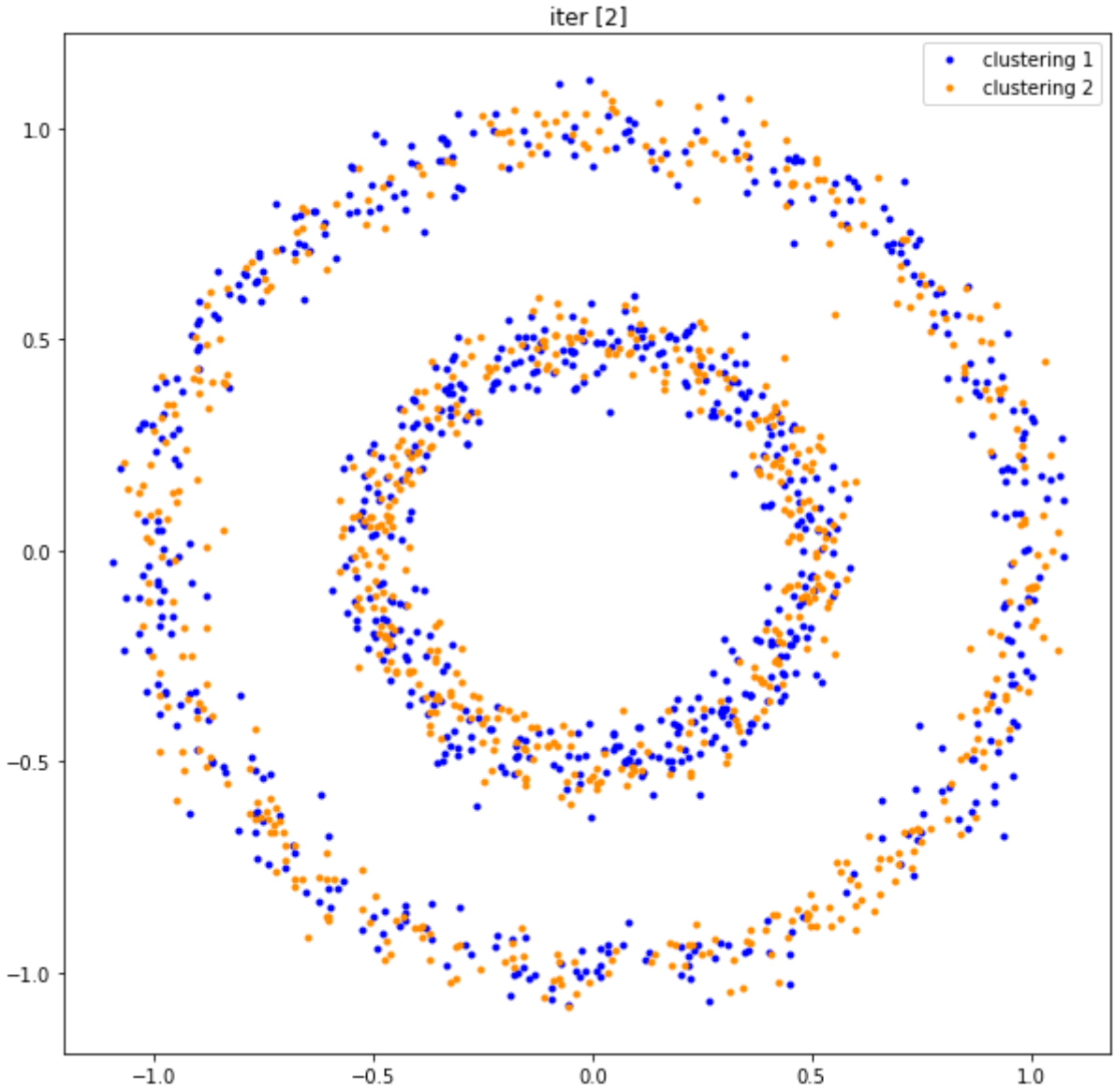
- gamma = 100



- gamma = 1000



- gamma = 10000



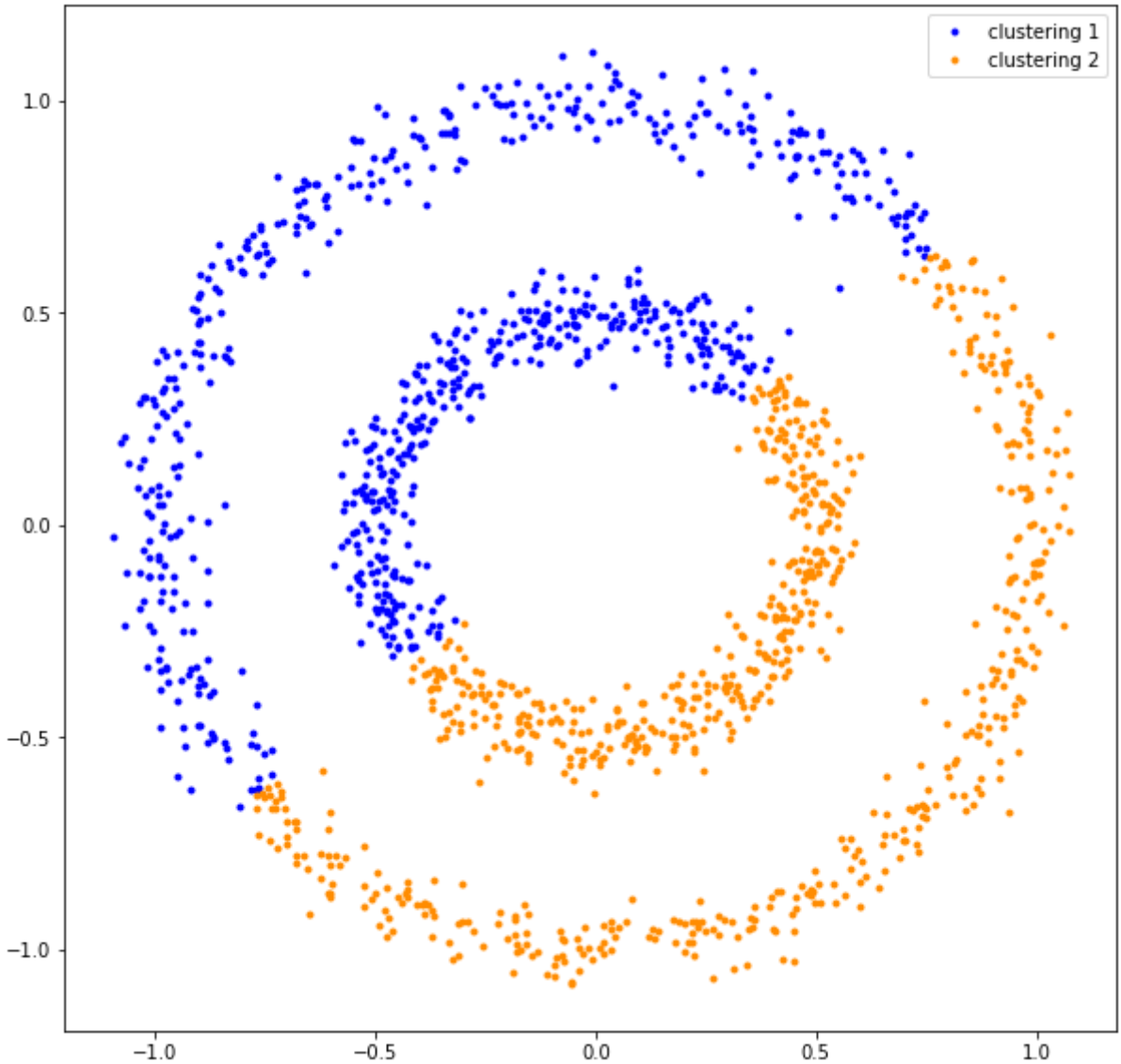
可以看到 gamma 越大 overfitting 越嚴重
加上分群初始化是隨機，所以 overfitting 一開始的結果就像平均分佈在分群

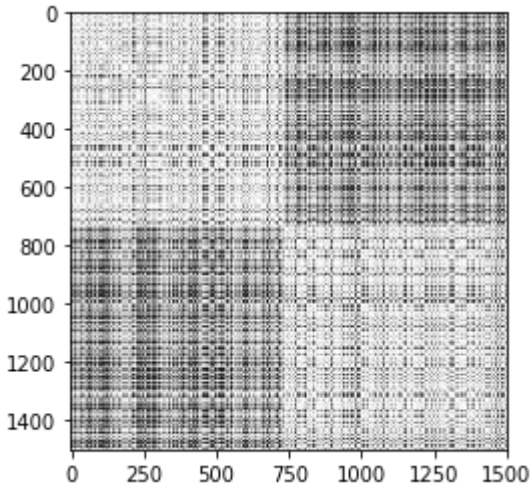
如果 gamma 太小就會有點變成 k-means

diff gamma for spectral clustering

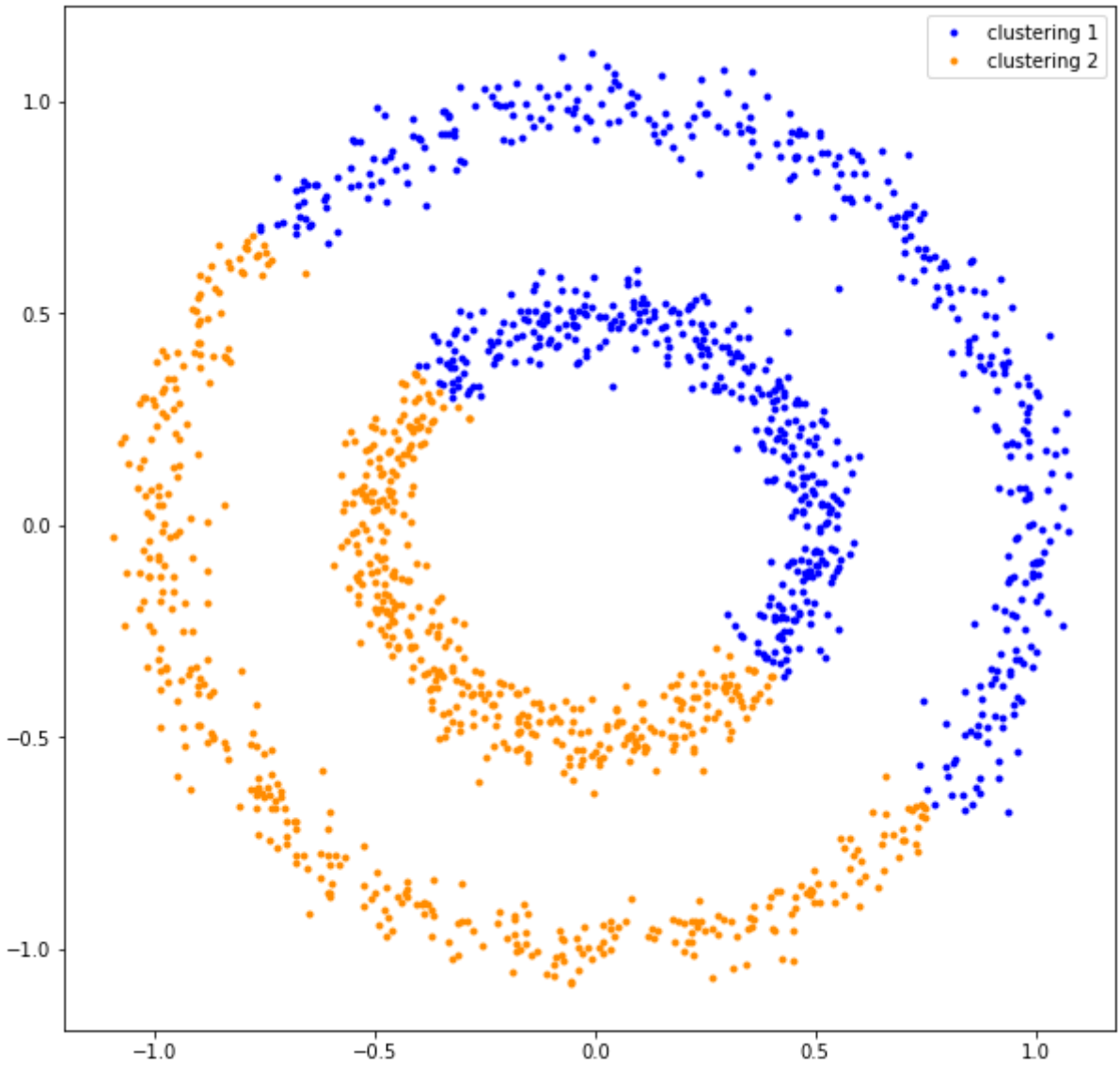
使用 circle dataset, k=2

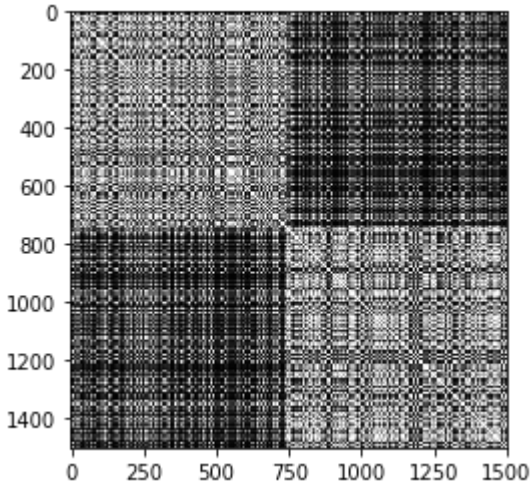
- gamma = 0.1



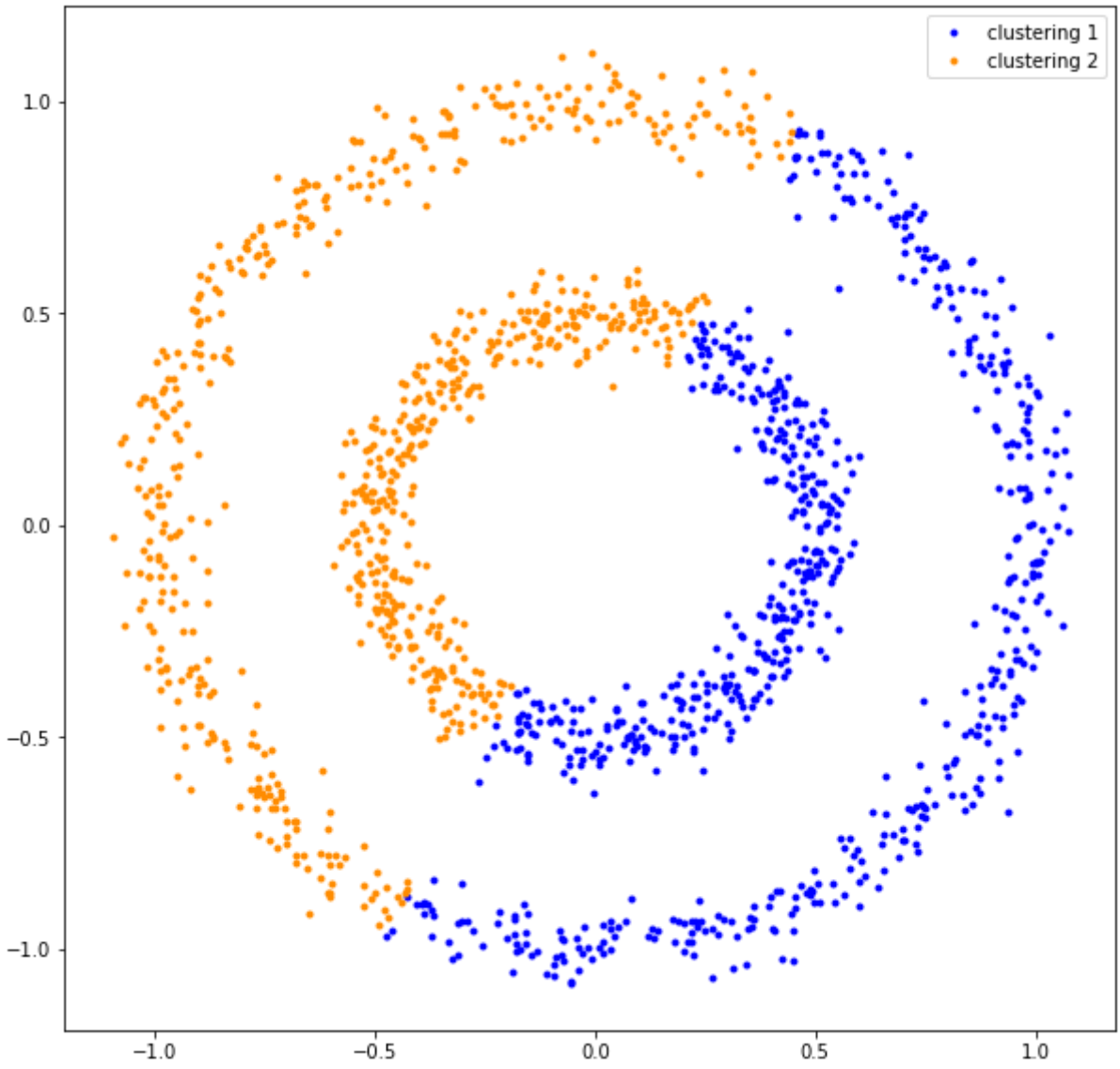


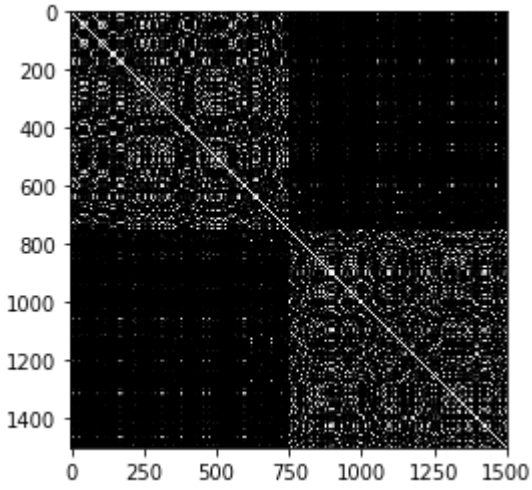
- gamma = 1


clustering 1
clustering 2
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

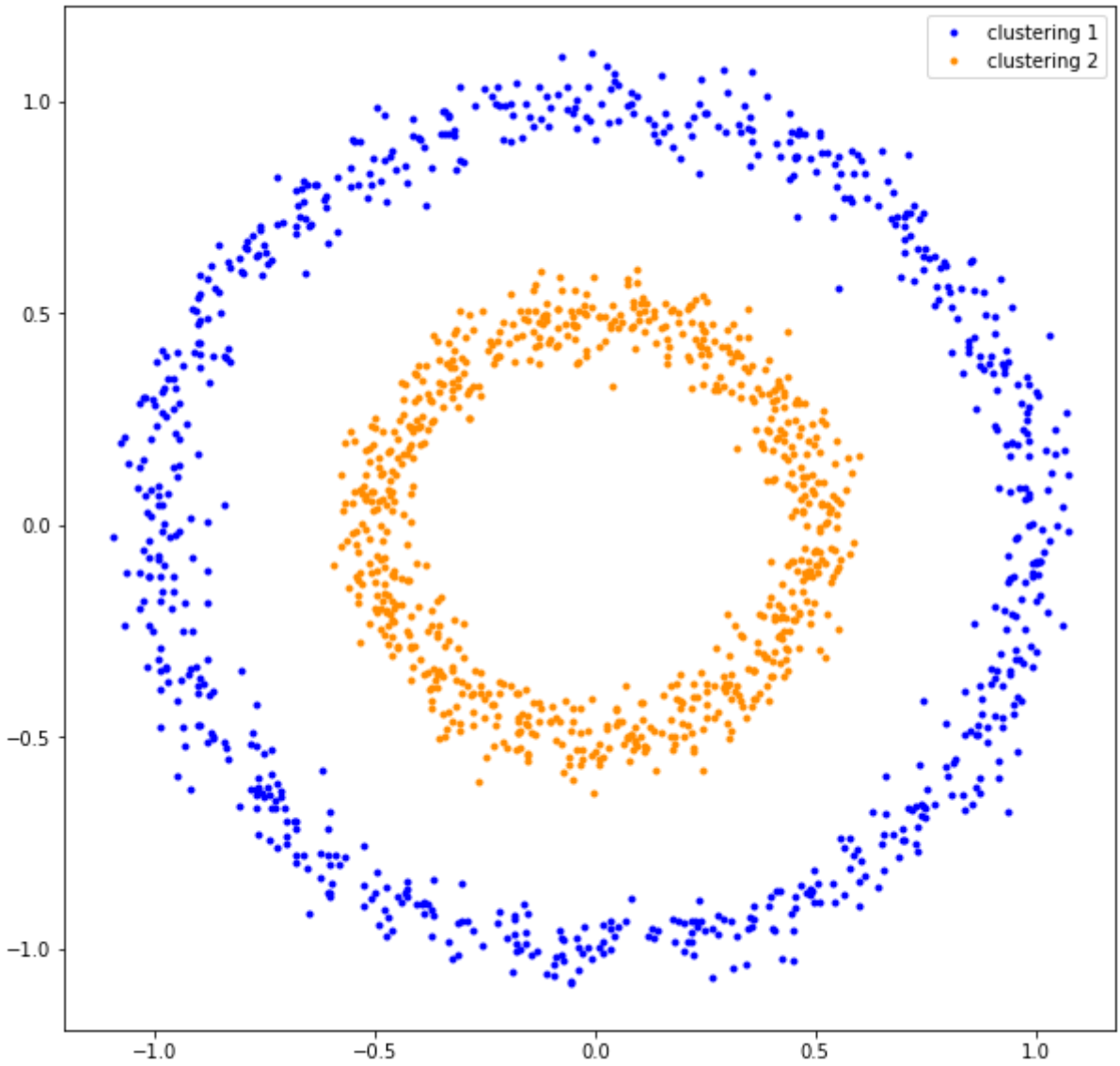

0
200
400
600
800
1000
1200
1400
0
250
500
750
1000
1250
1500

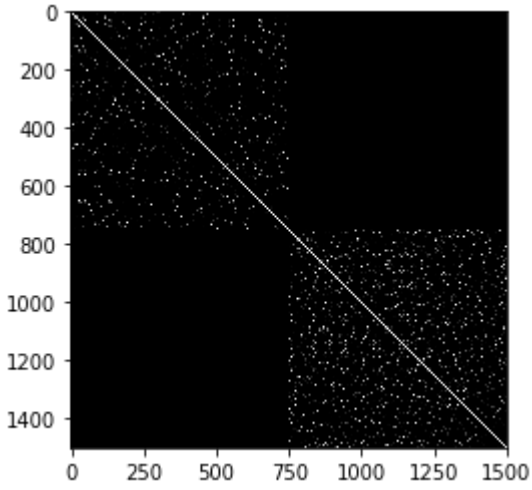
- gamma = 10


clustering 1
clustering 2
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

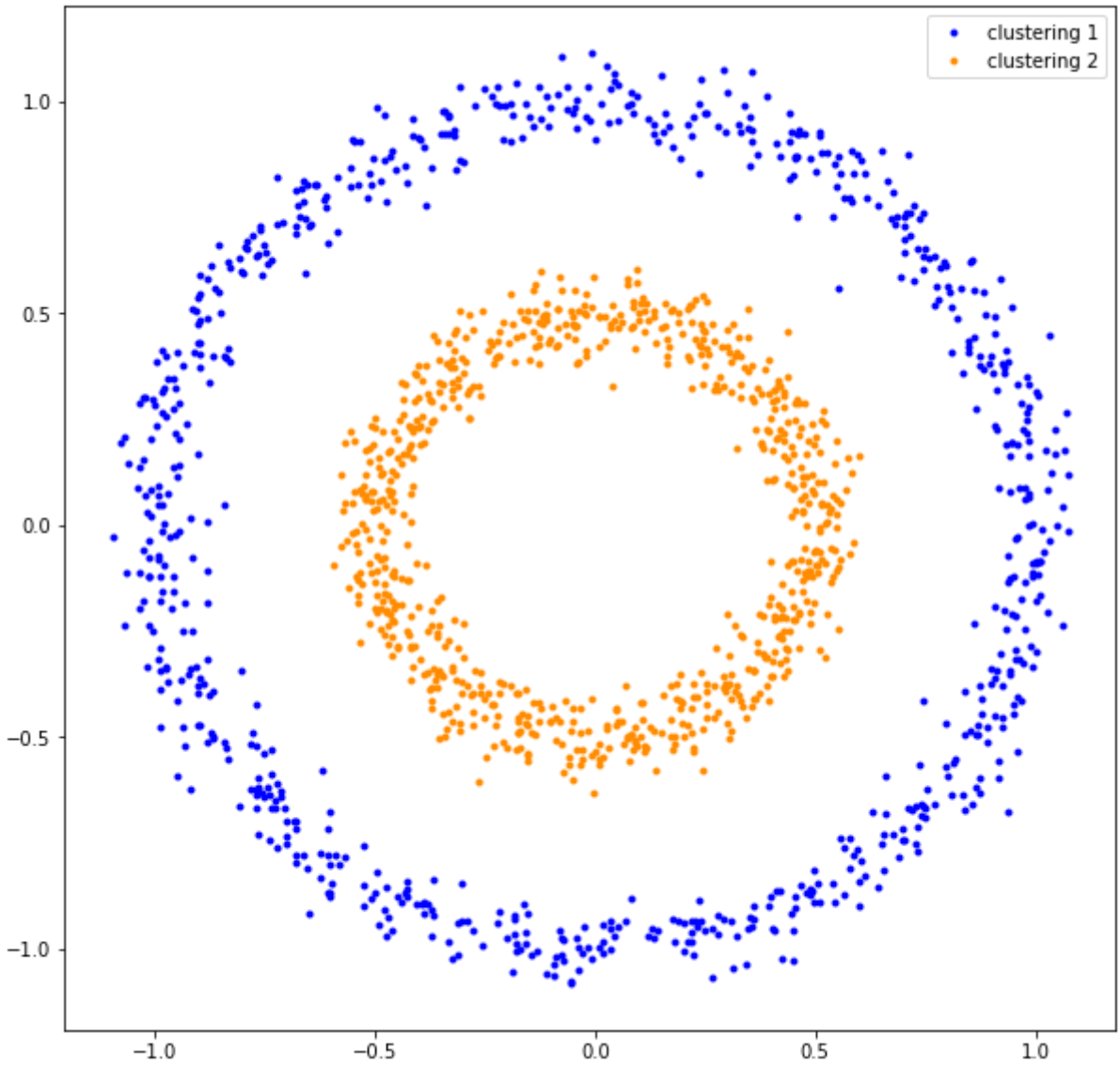

0
200
400
600
800
1000
1200
1400
0
250
500
750
1000
1250
1500

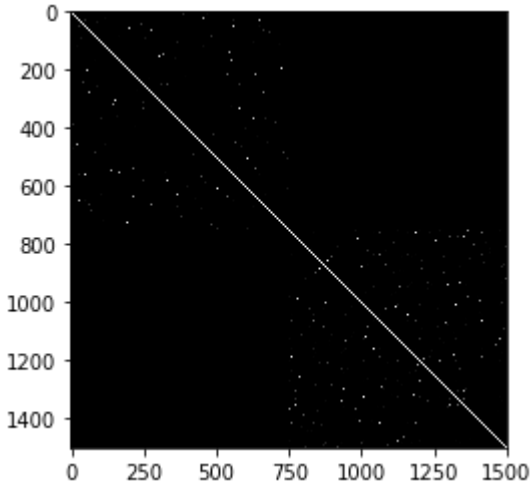
- gamma = 100


clustering 1
clustering 2
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

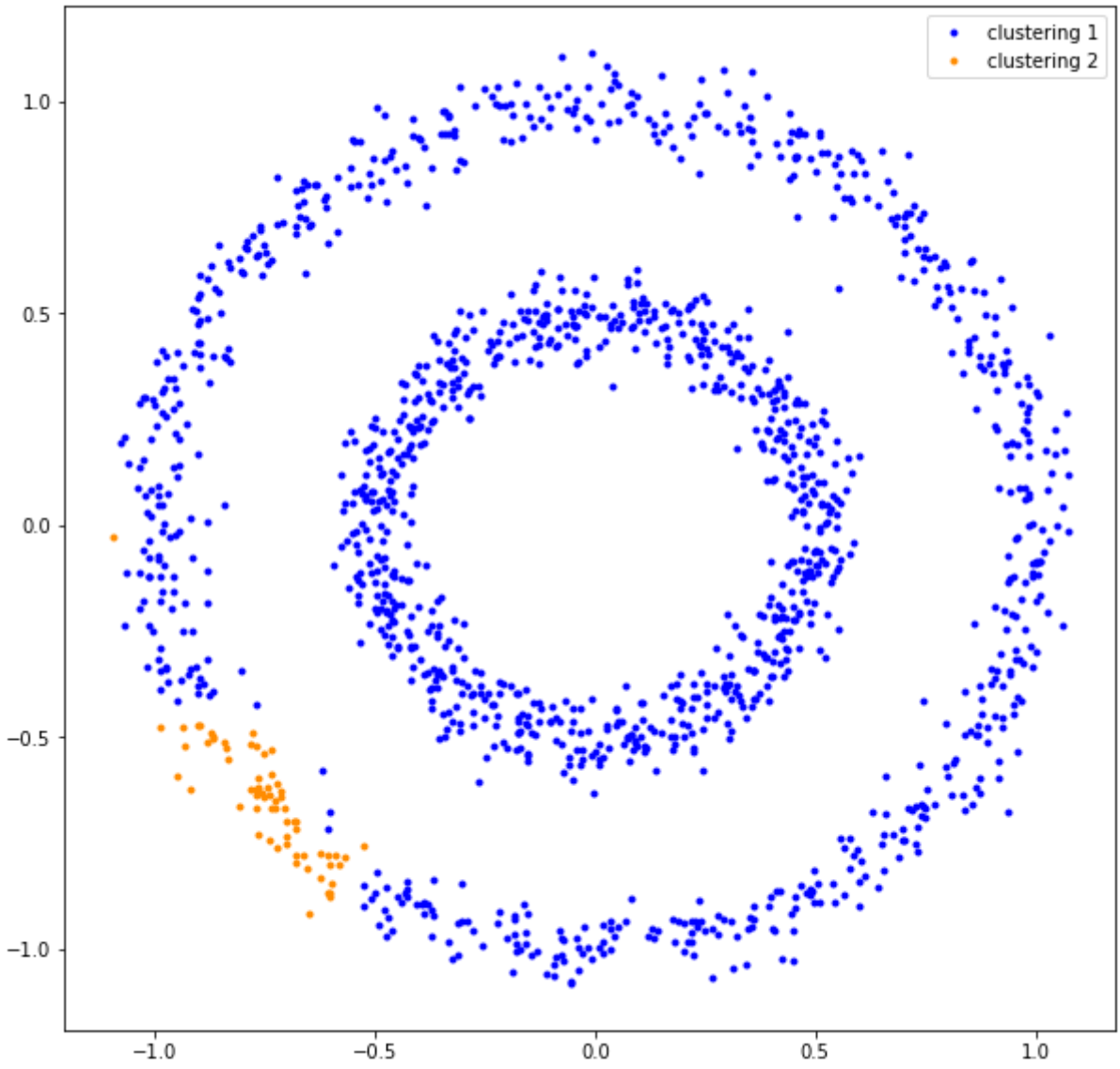

0
200
400
600
800
1000
1200
1400
0
250
500
750
1000
1250
1500

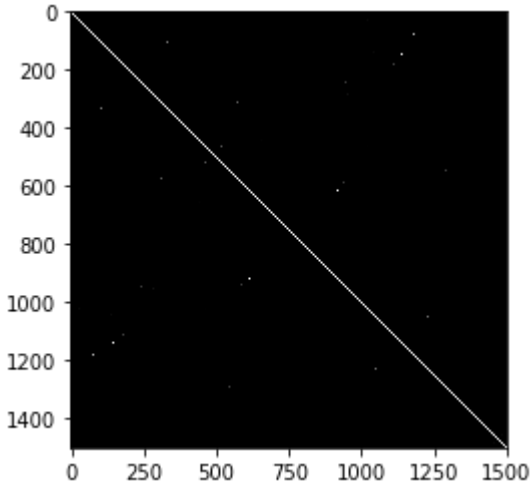
- gamma = 1000


clustering 1
clustering 2
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0


0
200
400
600
800
1000
1200
1400
0
250
500
750
1000
1250
1500

- gamma = 10000





也有 gamma 太大導致 overfitting，太小導致 underfitting
但是容忍範圍蠻廣泛 100 < gamma < 1000 都能夠分出好結果

而且 gram matrix 可以很明顯告訴你好不好分