ML Homework 6 Report

k-means clustering, kernel k-means, spectral clustering, DBSCAN

Implementation

1. k-means clustering

First, initialize n cluster centers using mean and standard deviation.

```python
def k_means(data, n_cluster, data_file):
    k = n_cluster      # number of clusters
    n = data.shape[0]  # number of data points

    # generate random centers
    mean = np.mean(data, axis=0)
    std = np.std(data, axis=0)
    centers = np.random.randn(k,data.shape[1]) * std + mean
```

Then, compute euclidean distances from each data point to all cluster centers, and store the results in numpy array `distances`. Use `distances` to assign each data point to the closest center (cluster). With the new cluster result, update the cluster centers. Do the above two steps until the centers no longer change.

```python
    centers_old = np.zeros(centers.shape)
    centers_new = np.copy(centers)
    # distances[i, k]: distance of data i to cluster k
    distances = np.zeros((n,k))
    # data i belongs to clusters[i]
    clusters = np.zeros(n)
    iteration = 0
    while (LA.norm(centers_new - centers_old) != 0):
        iteration += 1
        # compute distances from each data to every cluster center
        for i in range(k):
            distances[:,i] = LA.norm(data - centers[i], axis=1)
        # assign each data to the closest center (cluster)
        clusters = np.argmin(distances, axis=1)
        centers_old = np.copy(centers_new)
        # update centers
        for i in range(k):
            centers_new[i,:] = np.mean(data[clusters == i], axis=0)
    return clusters
```

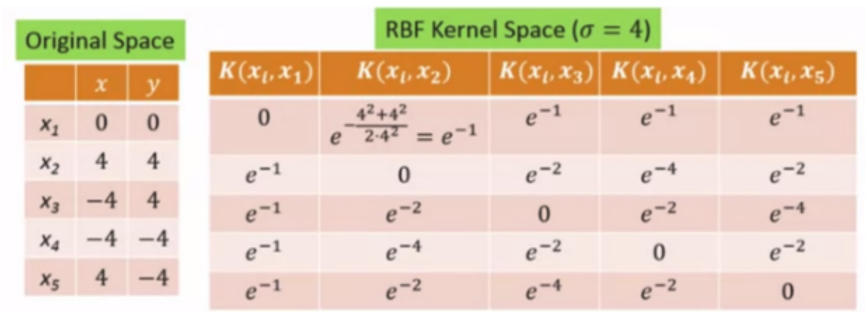
2. kernel k-means

First, compute the gram matrix using RBF kernel.

$$k(x,y) = e^{-\sigma\|x-y\|^2}$$

RBF kernel

Gram matrix is symmetric.



Original Space		
	x	y
x_1	0	0
x_2	4	4
x_3	−4	4
x_4	−4	−4
x_5	4	−4

RBF Kernel Space ($\sigma = 4$)				
$K(x_i,x_1)$	$K(x_i,x_2)$	$K(x_i,x_3)$	$K(x_i,x_4)$	$K(x_i,x_5)$
0	$e^{-\frac{4^2+4^2}{2\cdot 4^2}} = e^{-1}$	e^{-1}	e^{-1}	e^{-1}
e^{-1}	0	e^{-2}	e^{-4}	e^{-2}
e^{-1}	e^{-2}	0	e^{-2}	e^{-4}
e^{-1}	e^{-4}	e^{-2}	0	e^{-2}
e^{-1}	e^{-2}	e^{-4}	e^{-2}	0

Example of gram matrix

```python
def kernel_kmeans(data, n_class, gamma):
    k = n_class            # number of clusters
    n = data.shape[0]      # number of data
    # compute Gram matrix
    gram = np.zeros((n, n))
    for i in range(n):
        for j in range(i+1, n):
            # RBF kernel
            gram[i, j] = np.exp(-gamma * (LA.norm(data[i] - data[j]) ** 2))
            gram[j, i] = gram[i, j]
```

`alpha` is an indicator matrix of cluster assignment. If `alpha[n,k]` equals 1, `data[n]` belongs to cluster k. If `alpha[n,k]` equals 0, `data[n]` dose NOT belong to cluster k. Randomly initialize `alpha`.

```python
    # initialize alpha
    alpha = np.zeros((n, k))
    alpha[:, 0] = np.random.randint(2, size=n)
    alpha[:, 1] = 1 - alpha[:, 0]
```

Compute distances from each data point to all cluster centers using the following formula. Assign each data to the closest center and update clusters. If the `alpha` no longer changes, we are done with the clustering.

$$\left\|\phi(x_j) - \mu_k^{\phi}\right\| = \left\|\phi(x_j) - \sum_{n=1}^{N} \alpha_{kn}\phi(x_n)\right\|$$
$$= \mathbf{k}(x_j, x_j) - \frac{2}{|C_k|}\sum_{n} \alpha_{kn}\mathbf{k}(x_j, x_n) + \frac{1}{|C_k|^2}\sum_{p}\sum_{q} \alpha_{kp}\underbrace{\alpha_{kq}\mathbf{k}(x_p, x_q)}_{\text{Gram matrix!}}$$

distance function of kernel k-means

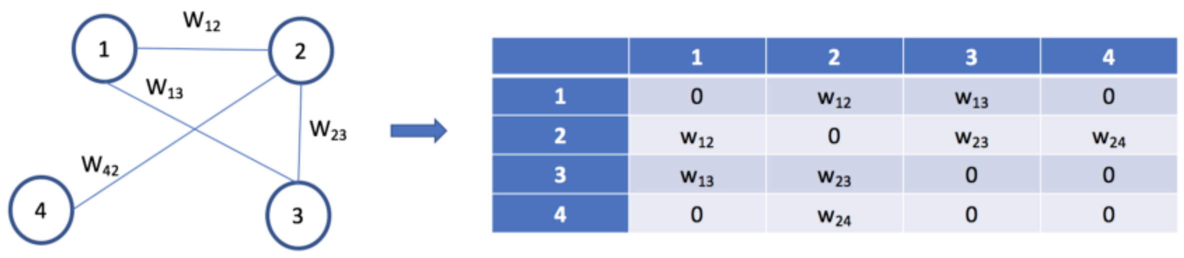
```python
    distances = np.zeros((n, k))
    converge = False
    iteration = 0
    while not converge:
        n_k = sum(alpha)    # number of data points for each class
        iteration += 1
        for i in range(k):
            tmp1 = np.ones(n)
            tmp2 = (2 / n_k[i]) * np.sum((np.tile(alpha[:,i].T, (n,1)) * gram), axis=1)
            tmp3 = (n_k[i] ** (-2)) *  np.sum((np.array([alpha[:,i].T,]*n).T * np.tile(alpha[:,i], (n,1))) * gram)
            distances[:,i] = tmp1 - tmp2 +  tmp3

        old_alpha = alpha
        for i in range(k):
            alpha[:,i] = 1 * (i == np.argmin(distances, axis=1))
        if np.sum(alpha - old_alpha) == 0:
            converge = True
    return alpha
```

3. spectral clustering

First construct the Gaussian kernel similarity matrix (RBF kernel).

```
def spectral_clustering(data, n_class):
    k = n_class             # number of class
    n = data.shape[0]       # number of data points
    # compute similarity
    # similarity matrix
    w = np.zeros((n,n))
    # Gaussian kernel similarity function
    for i in range(n):
        for j in range(n):
            gamma = 50
            w[i, j] = np.exp(-gamma * (LA.norm(data[i] - data[j]) ** 2))
            if i == j:
                w[i, j] = 0
```

Then, we compute Graph Laplacian and the normalized one. Use the normalized Graph Laplacian to get the second and third smallest eigen vectors as our new feature values. Finally, do k-means with the new features of eigen vectors as input data.



	1	2	3	4
1	0	w_{12}	w_{13}	0
2	w_{12}	0	w_{23}	w_{24}
3	w_{13}	w_{23}	0	0
4	0	w_{24}	0	0

example data

$$L = D - A,$$

where A is the adjacency matrix and D is the degree matrix such that

$$d_i = \sum_{\{j|(i,j)\epsilon E\}} w_{ij} \qquad Thus, L_{ij} = \begin{cases} d_i & if\ i = j \\ -w_{ij} & if\ (i,j)\ \epsilon\ E \\ 0 & if\ (i,j) \notin E \end{cases}$$

Graph Laplacian

$$\begin{aligned} d_1 &= w_{12} + w_{13} \\ d_2 &= w_{12} + w_{23} + w_{24} \\ d_3 &= w_{12} + w_{23} \\ d_4 &= w_{24} \end{aligned} \Rightarrow L = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & d_1 & -w_{12} & -w_{13} & 0 \\ 2 & -w_{12} & d_2 & -w_{23} & -w_{24} \\ 3 & -w_{13} & -w_{23} & d_3 & 0 \\ 4 & 0 & -w_{24} & 0 & d_4 \end{array}$$

example of Graph Laplacian

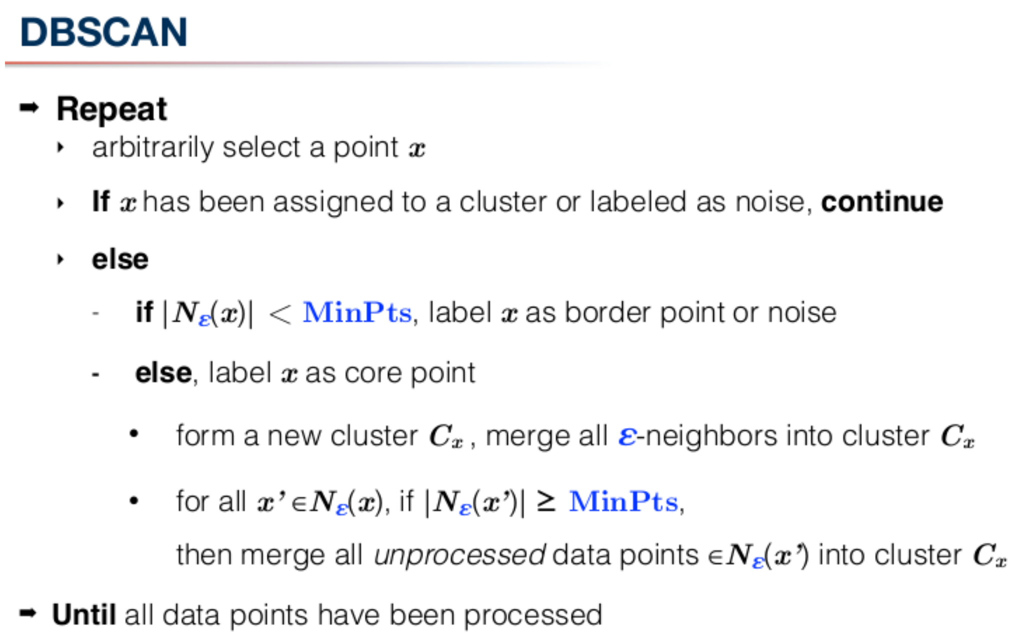
$$L_{norm} = D^{-1/2} L D^{-1/2}$$

normalized Graph Laplacian

```
    # compute normalized laplacian
    # L = D - W
    # L = D^{-1/2} L D{-1/2}
    D = np.zeros(w.shape)
    tmp = np.sum(w, axis=1)
    D = np.diag(tmp ** (-0.5))
    L_normalized = D.dot(np.diag(tmp) - w).dot(D)

    eig_val, eig_vec = LA.eig(L_normalized)
    dim = len(eig_val)
    dict_eval = dict(zip(eig_val,range(0,dim)))
    k_e_val = np.sort(eig_val)[1:1+k]
    idx = [dict_eval[k] for k in k_e_val]
    e_val = eig_val[idx]
    e_vec = eig_vec[:,idx]
    X = e_vec / np.sqrt(np.sum(e_vec ** 2, axis = 1)).reshape(n,1)
    return X
```

4. DBSCAN

According the algorithm:



DBSCAN

- **Repeat**
  - arbitrarily select a point x
  - **If** x has been assigned to a cluster or labeled as noise, **continue**
  - **else**
    - **if** $|N_\varepsilon(x)| < \mathrm{MinPts}$, label x as border point or noise
    - **else**, label x as core point
      - form a new cluster C_x , merge all ε-neighbors into cluster C_x
      - for all $x' \in N_\varepsilon(x)$, if $|N_\varepsilon(x')| \geq \mathrm{MinPts}$,
        then merge all *unprocessed* data points $\in N_\varepsilon(x')$ into cluster C_x
- **Until** all data points have been processed

DBSCAN algorithm

`classifications` of all data points are initialized as `UNCLASSIFIED`. Then, we iterate through all data points. If the current data point is `UNCLASSIFIED`, we check whether we should label it as `NOISE` or form a new cluster.

```python
def dbscan(data, eps, min_pts):
    cluster_id = 1
    # number of data points
    n = data.shape[0]
    classifications = [UNCLASSIFIED] * n
    for pt_id in range(n):
        if classifications[pt_id] == UNCLASSIFIED:
            if merge_to_cluster(data, eps, min_pts, classifications, pt_id, cluster_id):
                cluster_id = cluster_id + 1
   return classifications
```

Get the `neighbors` of the current point. If the number of `neighbors` is smaller than a given threshold, i.e. `imin_pts`, we label the current point as `NOISE`. If not, it is as core point, and we have to expand a new cluster.

```python
def merge_to_cluster(data, eps, min_pts, classifications, pt_id, cluster_id):
    neighbors = eps_neighbors(data, pt_id, eps)
    if len(neighbors) < min_pts:
        classifications[pt_id] = NOISE
        return False
    else:
        classifications[pt_id] = cluster_id
        for i in neighbors:
            classifications[i] = cluster_id

        while len(neighbors) > 0:
            current_point = neighbors[0]
            current_neighbors = eps_neighbors(data, current_point, eps)
            if len(current_neighbors) >= min_pts:
                for i in range(len(current_neighbors)):
                    target_point = current_neighbors[i]
                    if classifications[target_point] == UNCLASSIFIED:
                        neighbors.append(target_point)
                        classifications[target_point] = cluster_id
                    if classifications[target_point] == NOISE:
                        classifications[target_point] = cluster_id
            neighbors = neighbors[1:]
        return True
```

Initialization of k-means with different method

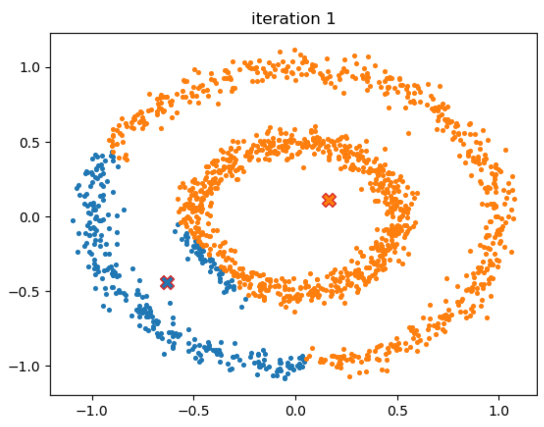
Use mean and standard deviation to initialize the centers.

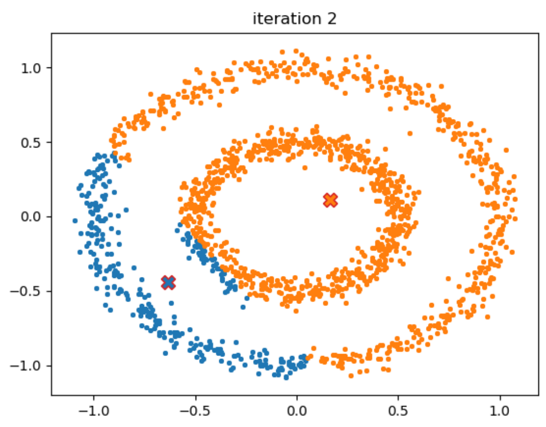
```
mean = np.mean(data, axis=0)
std = np.std(data, axis=0)
centers = np.random.randn(k,data.shape[1]) * std + mean
```

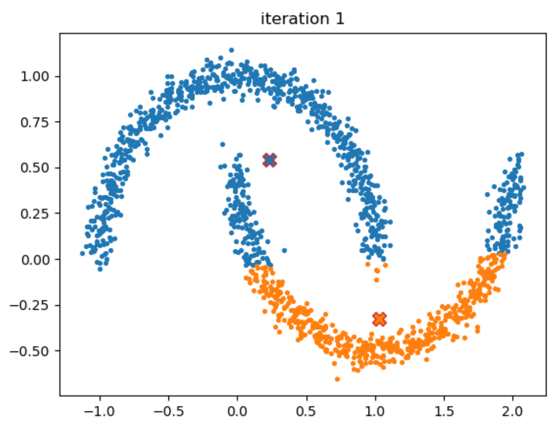
Results with n clusters

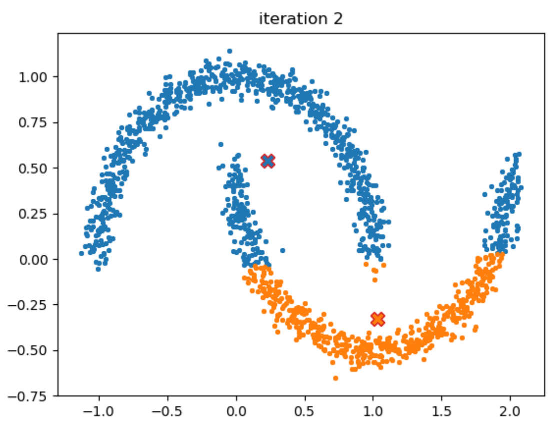
1. k-means

- 2 clusters

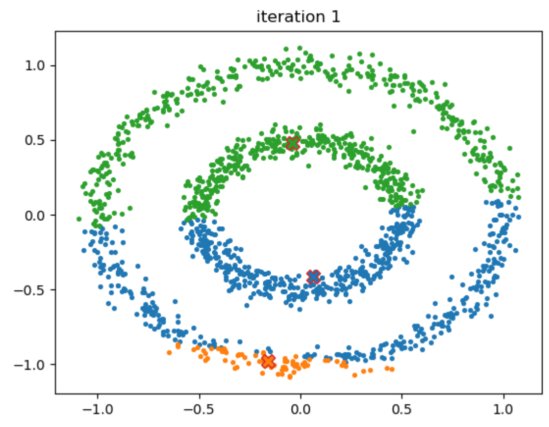


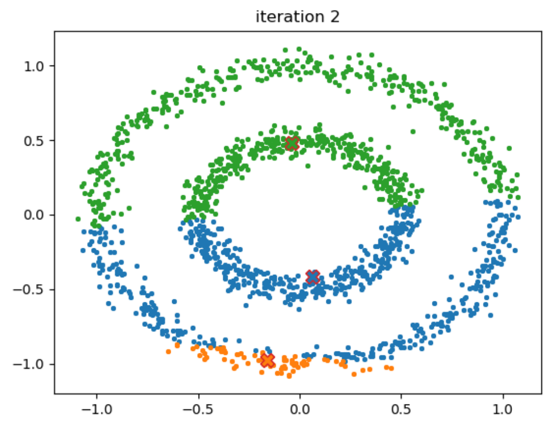


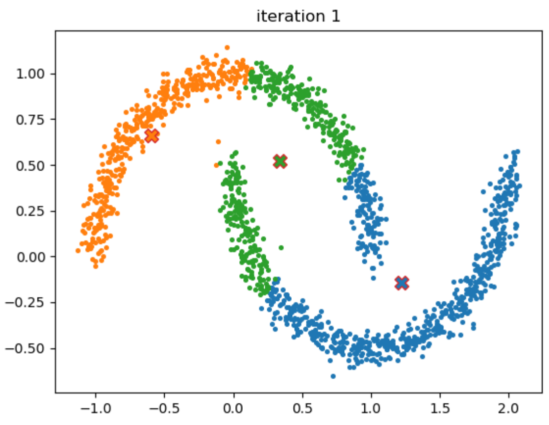


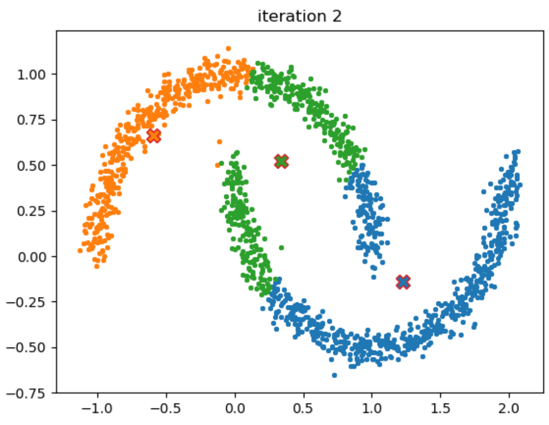


- 3 clusters



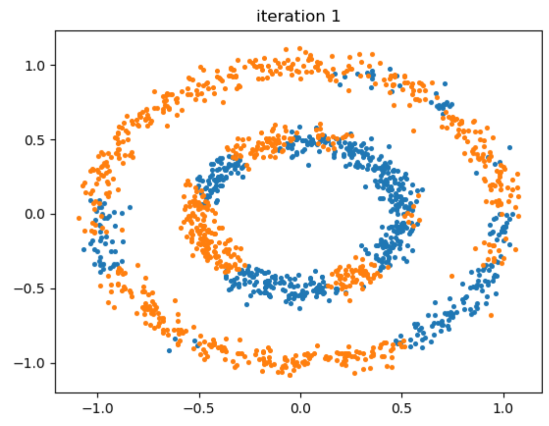


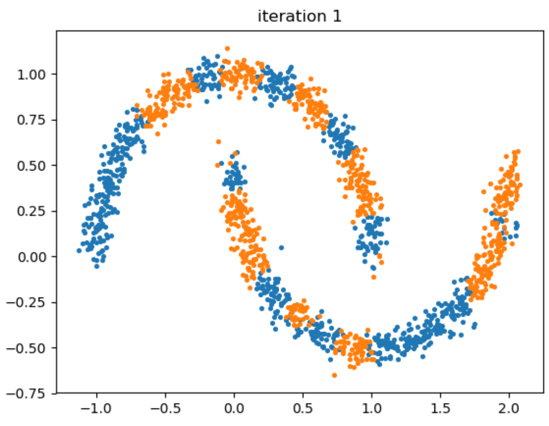




2. kernel k-means

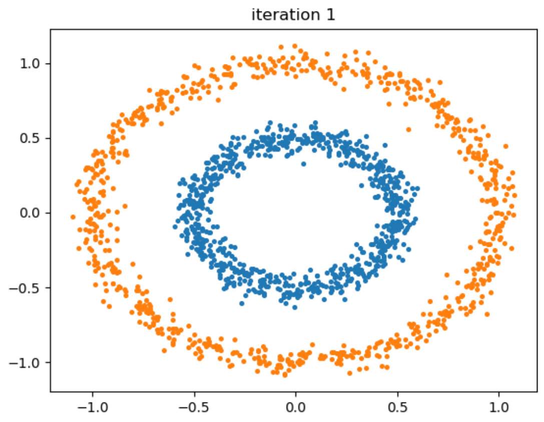
- 2 clusters

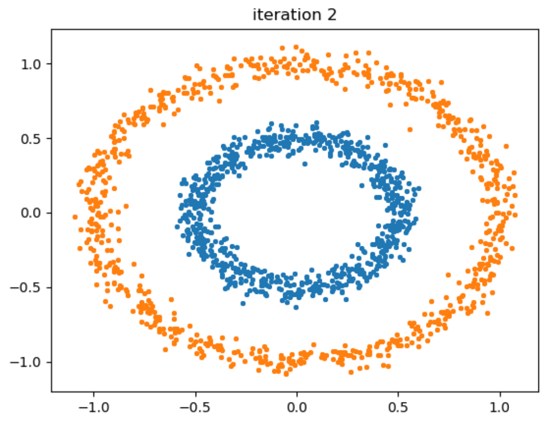


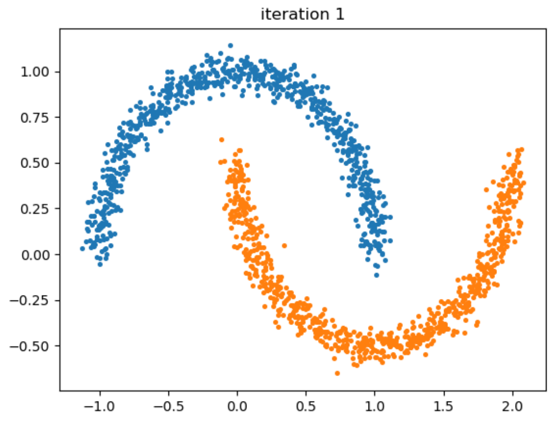


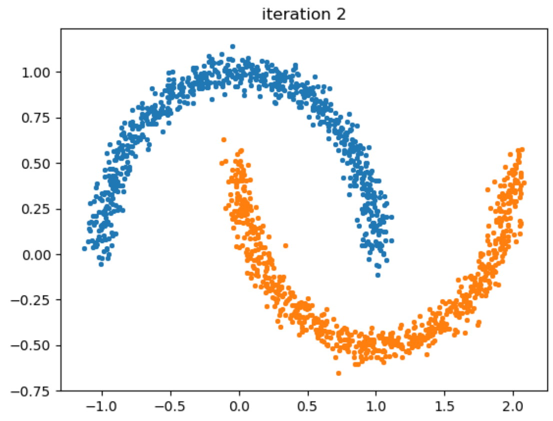
3. spectral clustering

- 2 clusters

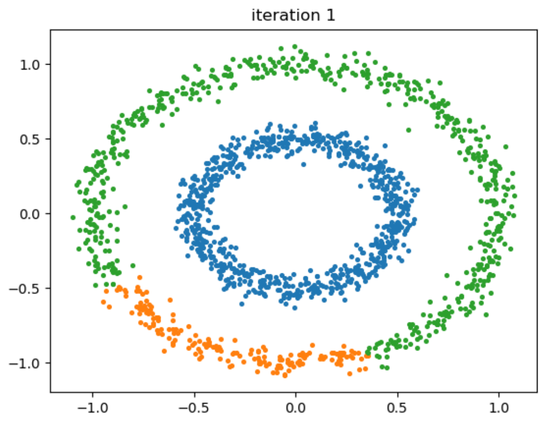

iteration 1

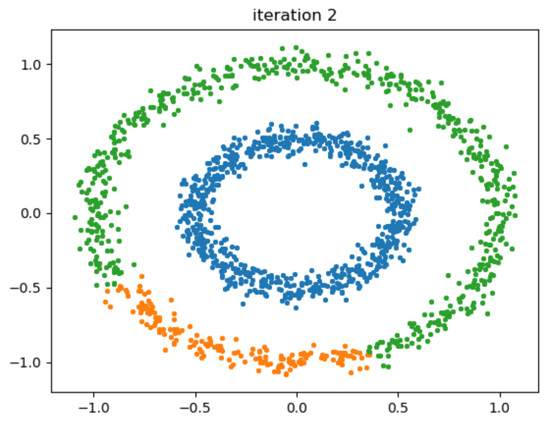

iteration 2

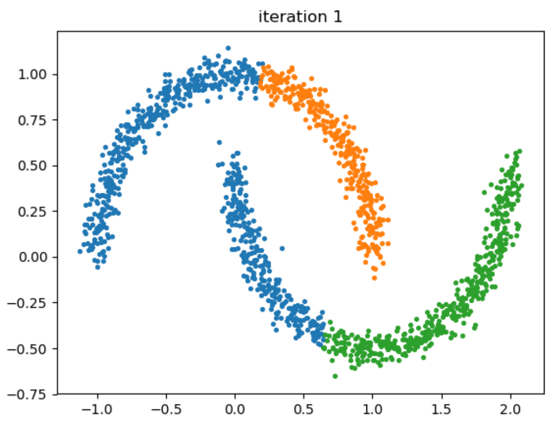

iteration 1

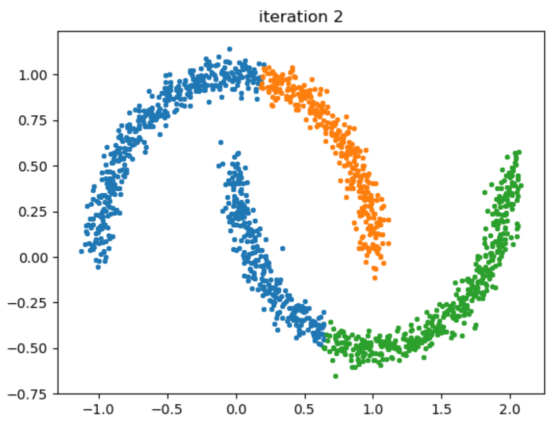

iteration 2

- 3 clusters

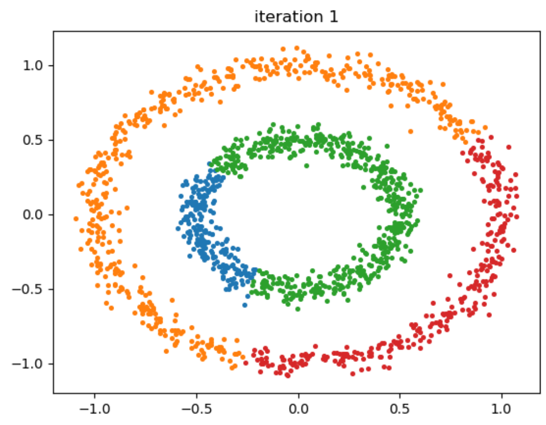

iteration 1

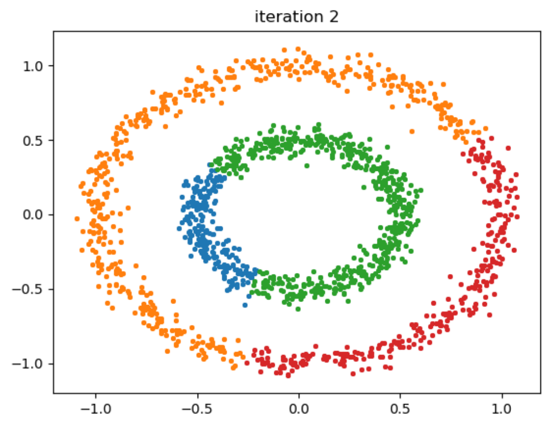

iteration 2

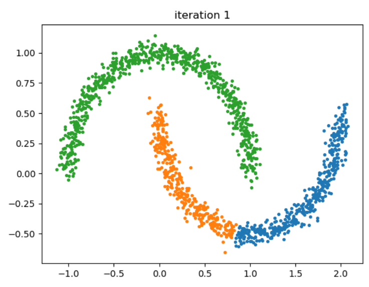


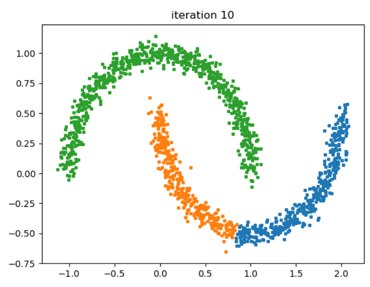


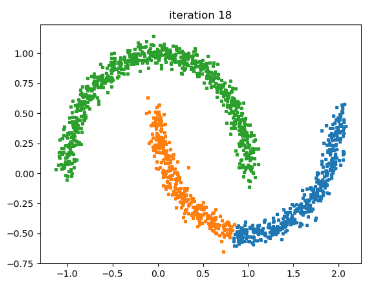
- 4 clusters





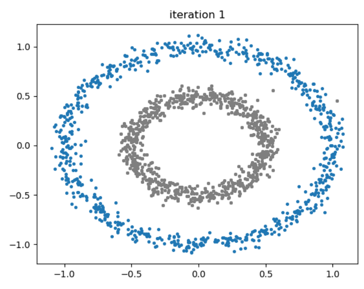


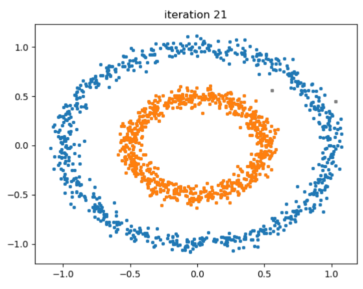


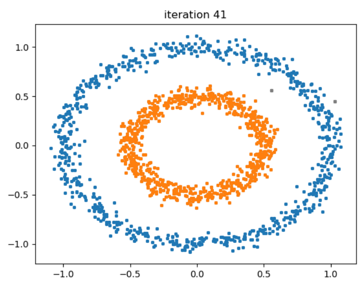


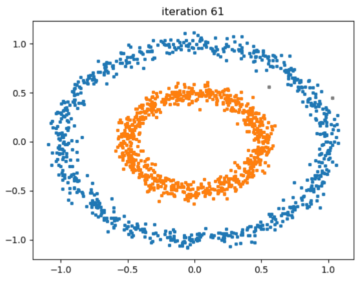
4. DBSCAN

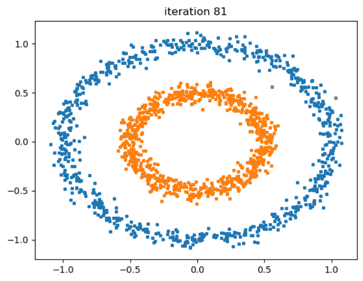
gray points are unprocessed data

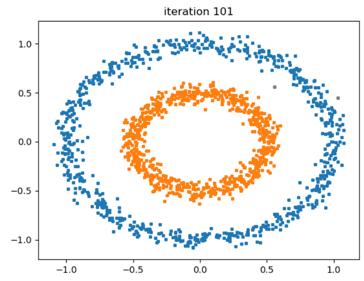


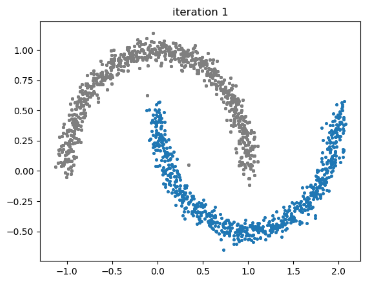


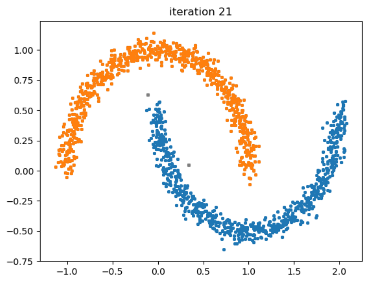


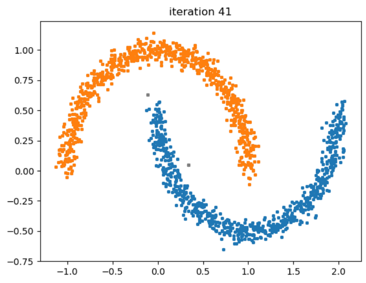


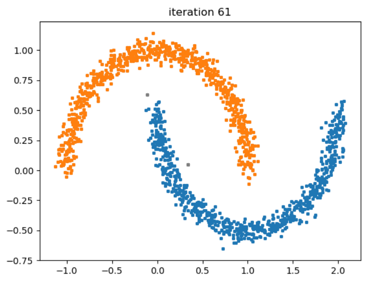


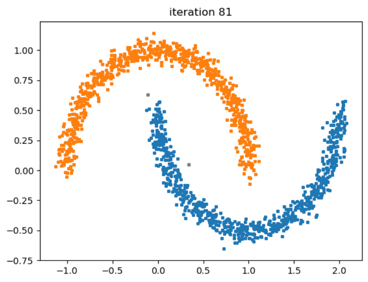


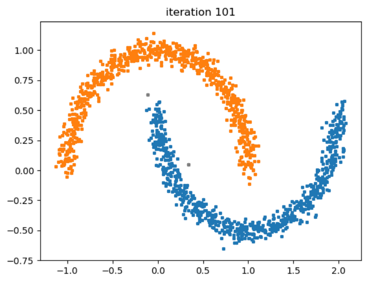




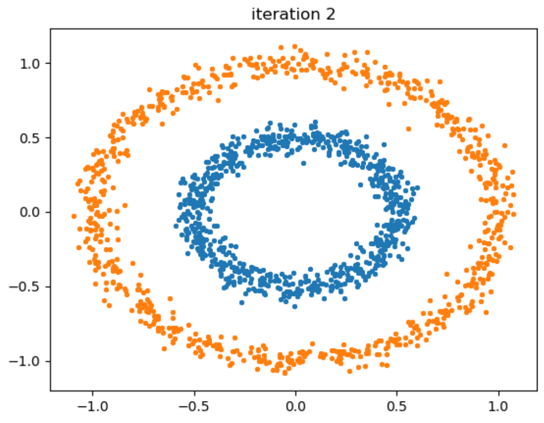




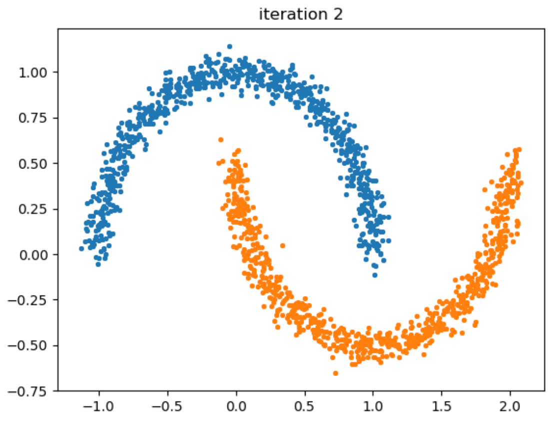




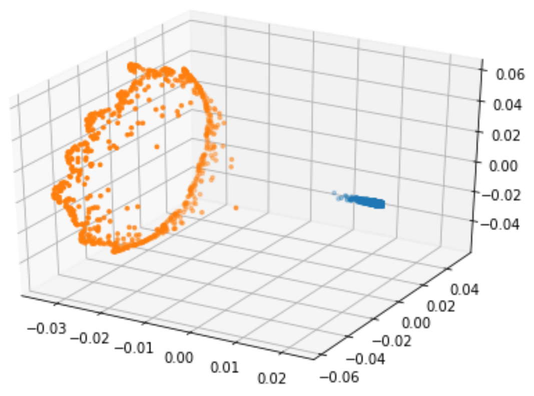
Eigenspace of Graph Laplacian in spectral clustering



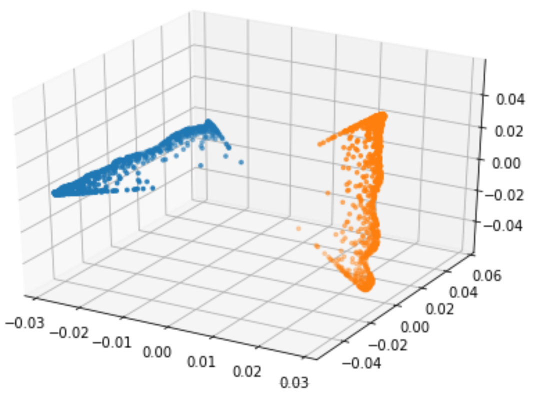
circle data in original space



moon data in original space



circle data in Eigenspace



moon data in Eigenspace

When the data are projected to the Eigenspace, they become linearly separable. The data within the same cluster have the same coordinates in the Eigenspace. For instance, the outer ring of the circle data are of the same class in the original space. They correspond to the orange ring in the Eigenspace.

Reference

- Spectral clustering