ML Homework 06: Kernel K-Means and Spectral Clustering

- Student ID: 309553002
- Name: 林育愷

ML Homework 06: Kernel K-Means and Spectral Clustering
- Code with Detailed Explanations
  - Prerequisites
  - Input Data
  - Gram Matrix (Kernel Matrix)
  - [Part 1] Clustering Procedure
    - Kernel K-Means
    - Spectral Clustering
  - [Part 2] Change of k
  - [Part 3] Initialization of K-Means
    - [Part 4] Eigenspace of Graph Laplacian
- Experiments Settings, Results, and Discussion
  - Kernel K-Means
  - Spectral Clustering
- References

Code with Detailed Explanations

Prerequisites

I use Python 3.6 for this implementation based on platform Ubuntu 18.04, with the following packages

- NumPy,
- SciPy `scipy.spatial.distance`, and
- imageio.

Input Data

Input format is an image of size 100×100. We use `imageio` to read an image file.

```
import imageio


# filename = 'image1.png' or 'image2.png'
def load_image(filename):
    img = imageio.imread(filename)
    return img[..., :3]
```

Since we concern the spatial position of each pixel, we add x-axis value and y-axis as two addition channels to original data.

```
import numpy as np

```

```python

def add_spatial_channels(img):
    super_image = np.zeros(img.shape[:2] + (5, ))
    super_image[..., :3] = img[..., :3]

    indices = np.indices(img.shape[:2])
    super_image[..., 3] = indices[0]
    super_image[..., 4] = indices[1]
    return super_image


# main procedure for loading data
# filename = 'image1.png' or 'image2.png'
def load_data(filename):
    img = load_image(filename)
    data = add_spatial_channels(img)
    return data
```

Hence for any input data, the first three channels are related to RGB, and the later two channels are related to spatial position.

Gram Matrix (Kernel Matrix)

According to the equation

$$k(x_i, x_j) = \exp(-\gamma_s \|S(x_i) - S(x_j)\|) \times \exp(-\gamma_s \|C(x_i) - C(x_j)\|),$$

where $S(x)$ is the spatial information (the last 2 channels) and $C(x)$ is the color information (the first 3 channels), we build the gram matrix

$$K = \begin{pmatrix} k(x_1, x_1) & \cdots & k(x_1, x_n) \\ \vdots & & \vdots \\ k(x_n, x_1) & \cdots & k(x_n, x_n) \end{pmatrix},$$

where n is the number of pixels. In our cases $n = 10000$.

> My default configuration of (γ_c, γ_s) is $(10^{-4}, 10^{-3})$.

```python
from scipy.spatial.distance import cdist


def gram_matrix(data, gamma_c=1e-4, gamma_s=1e-3):
    size = data.shape[0]
    matrix = np.zeros((size, size))

    color_distance = cdist(data[:, :3], data[:, :3], metric='sqeuclidean')
    spatial_distance = cdist(data[:, 3:], data[:, 3:], metric='sqeuclidean')

    color_kernel = np.exp(-gamma_c * color_distance)
    spatial_distance = np.exp(-gamma_s * spatial_distance)

    matrix = color_kernel * spatial_distance
    return matrix
```

[Part 1] Clustering Procedure

Kernel K-Means

Suppose we obtain an initial label for each data point (will be discussed in the third part).

```
class KernelKMeans(object):
    def __init__(self, data, k, gram=None, init='random', tol=1e-3):
        self.data = data
        self.size = data.shape[0]
        self.k = k
        self.gram = gram_matrix(data) if gram is None else gram
        self.label = self._get_init_label(init)
        self.label_history = [self.label]
        self.tol = tol
```

Then for each iteration[1],

1. We calculate the distance of each data points x_i and each cluster mean in feature space using the equation

$$k(x_i, x_i) - \frac{2}{|C_k|}\sum_{j\in C_k} k(x_i, x_j) + \frac{1}{|C_k|^2}\sum_{p\in C_k}\sum_{q\in C_k} k(x_p, x_q),$$

   where $C_k = \{i | x_i \text{ belongs to cluster } k\}$.

```
1  class KernelKMeans(object):
2      def _calc_feature_space_distance(self, label_matrix=None):
3          if label_matrix is None:
4              k = self.k
5              label_matrix = self.label_matrix
6              label_count = self.label_count
7          else:
8              k = label_matrix.shape[1]
9              label_count = label_matrix.sum(axis=0)
10         assert np.all(label_count > 0)
11
12         # calculate distance in feature space using equation (2) in
13         # https://www.cs.utexas.edu/users/inderjit/public_papers/kdd_spectral_kernelkmeans.pdf
14         distance = np.zeros((self.size, k))
15         distance += np.diag(self.gram)[:, np.newaxis]
16
17         distance -= 2 * (self.gram @ label_matrix) / label_count
18
19         scale = 1 / label_count**2
20         distance += np.diag(label_matrix.T @ self.gram @ label_matrix) * scale
21         return distance
```

2. For each data point, we choose the minimum distance of them and assign such cluster to the data point `line 8`.
3. Check if the procedure converge, else we back to step 1 and perform the next iteration `line 11-16`.

We measure the convergence by the value

$$\frac{|\{i : x_i \text{ change another cluster though this iteration}\}|}{n},$$

and the convergence rate in my implementation is 10^{-3}.

```
class KernelKMeans(object):
    def train(self, max_iter=100):
        n_iter = 0
        while n_iter < max_iter:
            n_iter += 1
            logger.debug('train iter [%d/%d]' % (n_iter, max_iter))
            distance = self._calc_feature_space_distance()
            label = np.argmin(distance, axis=1)
            self.label_history.append(label)

            err = len(np.where(self.label != label)[0]) / self.size
            logger.debug('err: %f' % err)
            if err < self.tol:
                logger.debug('training terminated due to convergence')
                self.label = label
                break

            self.label = label
```

Note that we do not maintain mean of each cluster in feature space (especially for RBF kernel, the dimension of its feature space is infinity).

After the clustering algorithm, we calculate mean of each cluster and pick the color part for visualization.

```
class KernelKMeans(object):
    def visualize_as_gif(self, image_shape, filename):
        label_color = (self.label_matrix.T @ self.data)[:, :3]
        label_color /= self.label_count[:, np.newaxis]
        label_color = label_color.astype(np.uint8)
        with imageio.get_writer(filename, mode='I') as writer:
            for label in self.label_history:
                image = label_color[label.reshape(image_shape)]
                writer.append_data(image)

    def visualize_as_image(self, image_shape, filename):
        label_color = (self.label_matrix.T @ self.data)[:, :3]
        label_color /= self.label_count[:, np.newaxis]
        label_color = label_color.astype(np.uint8)

        image = label_color[self.label.reshape(image_shape)]
        imageio.imwrite(filename, image)
```

Finally the main function is shown below:

```
if __name__ == '__main__':
    # fixed random seed for debugging
    np.random.seed(0)

```

```python
    parser = argparse.ArgumentParser()
    parser.add_argument('filename', type=str, help='path to image file')
    parser.add_argument('k', type=int, help='number of clusters')
    parser.add_argument('init', type=str, help='methods of initialization')
    parser.add_argument('--debug',
                        action='store_true',
                        help='show debug message')

    args = parser.parse_args()
    if args.debug:
        logger.setLevel(logging.DEBUG)
    filename = args.filename
    k = args.k
    init = args.init

    raw_filename = osp.splitext(filename)[0]
    gif_filename = 'images/%s_%d_%s_kkm_result.gif' % (raw_filename, k, init)
    png_filename = 'images/%s_%d_%s_kkm_result.png' % (raw_filename, k, init)

    data = load_data(filename)
    logger.debug('calculating gram matrix')
    gram = gram_matrix(data.reshape(-1, 5))

    logger.debug('initializing kernel k-means')
    kkm = KernelKMeans(data.reshape(-1, 5), k, gram, init)

    kkm.train()

    logger.debug('visualizing result')
    kkm.visualize_as_gif(data.shape[:2], gif_filename)
    kkm.visualize_as_image(data.shape[:2], png_filename)
```

Spectral Clustering

According to [2], first we calculate Graph Laplacian L with the equation

$$L = D - K,$$

where

$$D = \begin{pmatrix} \sum_j k(x_j, x_1) & & & \\ & \sum_j k(x_j, x_2) & & \\ & & \ddots & \\ & & & \sum_j k(x_j, x_n) \end{pmatrix}.$$

Then we solve the eigenvalue problem on different matrices:

- $L_{\text{ratio}} = L$ for unnormalized (ratio-cut) case,
- $L_{\text{normalized}} = D^{-1/2} L D^{-1/2}$ for normalized case.

```python
class SpectralClustering(object):
    def _calc_graph_laplacian(self, mode='unnormalized'):
        graph_laplacian = self.degree_matrix - self.gram
        if mode == 'unnormalized':
            pass
        elif mode == 'normalized':
            d = np.diag(1 / np.sqrt(np.diag(self.degree_matrix)))
            graph_laplacian = d @ graph_laplacian @ d
        else:
            raise NotImplementedError("Unknown clustering mode")

        return graph_laplacian
```

We will obtain a set of ordered nonzero eigenvalues $\Lambda = \{\lambda_1, \dots \lambda_m\}$ and corresponding eigenvectors $V = \{v_1, \dots, v_m\}$ with $m \leq n$. Later we define the feature matrix

$$U = \begin{pmatrix} v_1 & v_2 & \dots & v_k \end{pmatrix} \in \mathbb{R}^{n \times k},$$

and the feature mapping $\phi(x_i) = Ue_i$.

> e_i is the i-th vector of the standard basis of $\mathbb{R}^n$.

Afterward we perform the standard K-Means algorithm to figure out the clustering result.

```python
class SpectralClustering(object):
    def __init__(self,
                 gram,
                 k,
                 kmeans_init='random',
                 mode='unnormalized',
                 tol=1e-3):
        self.gram = gram
        self.size = gram.shape[0]
        self.k = k
        self.kmeans_init = kmeans_init
        self.tol = tol
        self.degree_matrix = np.diag(gram.sum(axis=1))
        self.graph_laplacian = self._calc_graph_laplacian(mode)
        self.features = np.zeros((self.size, self.k))

        self.eigvals = None
        self.eigvecs = None

        self.km = None

    @property
    def label(self):
        return self.km.label

    @property
    def label_history(self):
        return self.km.label_history

    def train(self):
        # solve the eigenvalue problem
        if self.eigvals is None or self.eigvecs is None:
            eigvals, eigvecs = np.linalg.eig(self.graph_laplacian)
```

```
            self.eigvals = eigvals
            self.eigvecs = eigvecs

        # pick up the first k nonzero eigenvalues with ones corresponding
        # eigenvectors, and generate the feature mapping
        indices = np.argsort(self.eigvals)
        self.eigvals = self.eigvals[indices]
        self.eigvecs = self.eigvecs[:, indices].real
        base_idx = np.where(self.eigvals > 1e-8)[0][0]
        self.features = self.eigvecs[:, base_idx:base_idx + self.k]

        # trigger the K-Means algorithm
        self.km = KMeans(self.features, self.k, self.kmeans_init, self.tol)
        self.km.train()
```

The K-Means algorithm is similar to the Kernel K-Means while the Euclidean distance is used and we have to maintain the mean of each cluster (hence we have the maximiation step).

```
class KMeans(object):
    def __init__(self, data, k, init='random', tol=1e-3):
        self.data = data
        self.size = data.shape[0]
        self.degree = data.shape[1]
        self.k = k
        self.cluster_means = self._get_init_cluster_means(init)
        self.label = None
        self.label_history = []
        self.tol = tol

    @property
    def label_matrix(self):
        matrix = np.zeros((self.size, self.k), dtype=int)
        matrix[np.arange(self.size), self.label] = 1
        return matrix

    @property
    def label_count(self):
        return self.label_matrix.sum(axis=0)

    def _expectation_step(self):
        distance = np.zeros((self.size, self.k))
        for i in range(self.k):
            diff = self.data - self.cluster_means[i]
            distance[:, i] = np.linalg.norm(diff, axis=1)
        label = np.argmin(distance, axis=1)
        return label

    def _maximiation_step(self, label):
        cluster_means = np.zeros((self.k, self.degree))
        for i in range(self.k):
            indices = np.argwhere(label == i).flatten()
            cluster_means[i] = np.mean(self.data[indices], axis=0)
        return cluster_means

    def train(self, max_iter=100):
        n_iter = 0
        while n_iter < max_iter:
```

```
            n_iter += 1
            logger.debug('train iter [%d/%d]' % (n_iter, max_iter))
            label = self._expectation_step()
            cluster_means = self._maximiation_step(label)

            # handling first iteration
            if self.label is None:
                self.label = label
                self.cluster_means = cluster_means
                self.label_history.append(label)
                continue

            # check if converge
            err = len(np.where(self.label != label)[0]) / self.size
            logger.debug('err: %f' % err)
            if err < self.tol:
                logger.debug('training terminated due to convergence')
                self.label = label
                self.cluster_means = cluster_means
                self.label_history.append(label)
                break

            self.label = label
            self.cluster_means = cluster_means
            self.label_history.append(label)
```

The visualization method of Spectral Clustering is same as the method used in Kernel K-Means.

Finally the main function is shown below:

```
if __name__ == '__main__':
    # fixed random seed for debugging
    np.random.seed(0)

    parser = argparse.ArgumentParser()
    parser.add_argument('filename', type=str, help='path to image file')
    parser.add_argument('k', type=int, help='number of clusters')
    parser.add_argument('mode', type=str, help='clustering mode')
    parser.add_argument('init', type=str, help='methods of initialization')
    parser.add_argument('--debug',
                        action='store_true',
                        help='show debug message')
    parser.add_argument('--cache',
                        action='store_true',
                        help='use cached eigenvalues and eigenvectors')
    parser.add_argument('--save-cache',
                        action='store_true',
                        help='use cached eigenvalues and eigenvectors')

    args = parser.parse_args()
    if args.debug:
        logger.setLevel(logging.DEBUG)
    filename = args.filename
    k = args.k
    mode = args.mode
    init = args.init

```

```
28      raw_filename = osp.splitext(filename)[0]
29      gif_filename = 'images/%s_%d_%s_%s_sc_result.gif' % (raw_filename, k,
init,
30                                                          mode)
31      png_filename = 'images/%s_%d_%s_%s_sc_result.png' % (raw_filename, k,
init,
32                                                          mode)
33      eig_filename = 'images/%s_%d_%s_%s_sc_eig_result.png' % (raw_filename,
k,
34                                                              init, mode)
35      cache_eigvals_filename = '%s.%s.eigvals.npy' % (raw_filename, mode)
36      cache_eigvecs_filename = '%s.%s.eigvecs.npy' % (raw_filename, mode)
37
38      data = load_data(filename)
39      logger.debug('calculating gram matrix')
40      gram = gram_matrix(data.reshape(-1, 5))
41
42      logger.debug('initializing spectial clustering')
43      sc = SpectralClustering(gram, k, init, mode)
44
45      if args.cache:
46          sc.eigvals = np.load(cache_eigvals_filename)
47          sc.eigvecs = np.load(cache_eigvecs_filename)
48
49      logger.debug('trigger clustering')
50      sc.train()
51
52      sc.visualize_as_gif(data[..., :3], gif_filename)
53      sc.visualize_as_image(data[..., :3], png_filename)
54      sc.visualize_eigenspace_as_image(eig_filename)
55
56      if args.save_cache:
57          np.save(cache_eigvals_filename, sc.eigvals)
58          np.save(cache_eigvecs_filename, sc.eigvecs)
```

> - Since the computation cost of eigenvalue problem is heavy, we compute eigenvalues and eigenvectors in the first run and save as `.npy` files for caching. You can add `--save-cache` for the first computation and add `--cache` for later computation.
> - We will discuss the visualization of eigenspace (line 54) in part 4.

[Part 2] Change of k

In part one, the number of clusters is generalized, so we are able to change k when initializing a new instance of `KernelKMeans` or `SpectralClustering`.

The task trying different k is written outside the file.

```
import subprocess


# Kernel K-Means
for img in ['image1.png', 'image2.png']:
    for init in ['random', 'k-means++']:
        for k in map(str, [2, 3, 4]):
            cmd = ([
                'python3', 'HW06_KernelKMeans.py', img, k, init, '--debug'
            ])
```

```
            subprocess.check_output(cmd)

# Spectral Clustering
for img in ['image1.png', 'image2.png']:
    for init in ['random', 'k-means++']:
        for gl in ['unnormalized', 'normalized']:
            for k in map(str, [2, 3, 4]):
                cmd = ([
                    'python3', 'HW06_SpectralClustering.py', img, k, gl, init,
                    '--debug'
                ])
                subprocess.check_output(cmd)
```

[Part 3] Initialization of K-Means

In addition to random initialization, I implement the K-Means++ algorithm[3] in the two clustering methods.

The classical procedure is shown as follow:

1. Choose an initial center c_1 uniformly at random from data points.
2. Choose the next center $c_i = x'$ from data points with probability
$$\frac{D(x')^2}{\sum_x D(x)^2},$$
where $D(x)$ is the shortest distance from a data point x to the closest center we have already chosen.
3. Repeat step 2 until we have chosen a total of k centers.

```
class KMeans(object):
    def _get_init_cluster_means(self, init='random'):
        cluster_means = np.zeros((self.k, self.degree))
        if init == 'random':
            logger.debug('use random initialization for k-means')
            indices = np.random.randint(0, self.size, self.k)
            cluster_means[:] = self.data[indices]
        elif init == 'k-means++':
            logger.debug('use k-means++ initialization for k-means')
            idx = np.random.randint(0, self.size, 1)[0]
            cluster_means[0] = self.data[idx]
            for i in range(1, self.k):
                distance = np.zeros((self.size, i))
                for j in range(i):
                    diff = self.data - cluster_means[j]
                    distance[:, j] = np.sum(diff**2, axis=1)
                distance = np.min(distance, axis=1)
                distance = distance / np.sum(distance)
                print(distance)
                idx = np.random.choice(np.arange(self.size), p=distance)
                # idx = np.argmax(distance)
                cluster_means[i] = self.data[idx]
        else:
            raise NotImplementedError("Unknown init type")

        return cluster_means
```

As for Kernel K-Means, we also choose k centers. Note that

- the definition of distance is same as the one defined in Kernel K-Means;
- I will perform one step iteration so as to assign labels to all data points.

```
class KernelKMeans(object):
    def _get_init_label(self, init='random'):
        if init == 'random':
            return np.random.randint(0, self.k, self.size, dtype=int)
        elif init == 'k-means++':
            label = -1 * np.ones(self.size)
            label = label.astype(int)
            # randomly select a data point (as the center of the first cluster)
            label[np.random.randint(0, self.size, 1)] = 0

            # choose k-1 centers
            for i in range(1, self.k):
                label_matrix = np.zeros((self.size, i), dtype=int)
                label_matrix[np.arange(self.size), label] = 1
                label_matrix[np.argwhere(label == -1).flatten()] = 0

                distance = self._calc_feature_space_distance(label_matrix)
                distance = np.min(distance, axis=1)
                distance = distance / np.sum(distance)

                idx = np.random.choice(np.arange(self.size), p=distance)
                label[idx] = i

            # filling labels
            label_matrix = np.zeros((self.size, self.k), dtype=int)
            label_matrix[np.arange(self.size), label] = 1
            label_matrix[np.argwhere(label == -1).flatten()] = 0

            distance = self._calc_feature_space_distance(label_matrix)
            label = np.argmin(distance, axis=1)

            return label
```

[Part 4] Eigenspace of Graph Laplacian

We consider the relationship of eigenspace of graph Laplacian and clustering result. For my visualization, the coordinate in subspace of the eigenspace is computed by $U^T e_i$, where e_i is the i-th vector of the standard basis of $\mathbb{R}^n$.

```
class SpectralClustering(object):
    def visualize_eigenspace_as_image(self, filename):
        plt.figure()
        plt.xlabel('fisrt non-null eigenvector')
        plt.ylabel('second non-null eigenvector')
        coord = self.features[:, :2]
        for i in range(self.k):
            indices = np.argwhere(self.label == i).flatten()
            plt.scatter(coord[indices, 0], coord[indices, 1])
        plt.savefig(filename, dpi=500)
```

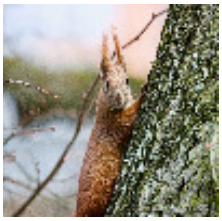
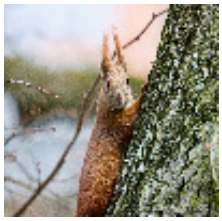
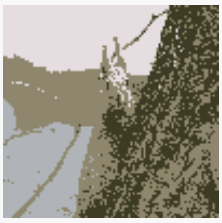
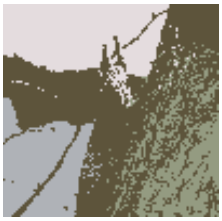
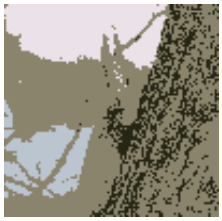
Experiments Settings, Results, and Discussion

I use two settings of (γ_c, γ_s) to see the differences between them. Overall the experiment results with different settings are shown below.

Kernel K-Means

Here are some observations:

1. The higher value of γ_c means that we have more concern of color information. In experiment results (especially for `image1.png`) the clustering result is more sparse than the other one (there are some "salt and pepper" in the case $\gamma_c = 10^{-3}$).
2. Different initialization methods cannot guarantee the convergence speed, but the results using K-Means++ are in general more reasonable than the results using random initialization.
3. For my visualization, colors are the means of clusters. You can see that especially for $(\gamma_c, \gamma_s) = (10^{-3}, 10^{-3})$, contrast of the results using K-Means++ are higher than the one of the results using random initialization.

	image1.png $(\gamma_c, \gamma_s) = (10^{-4}, 10^{-3})$	image1.png $(\gamma_c, \gamma_s) = (10^{-3}, 10^{-3})$	image2.png $(\gamma_c, \gamma_s) = (10^{-4}, 10^{-3})$	image2.png $(\gamma_c, \gamma_s) = (10^{-3}, 10^{-3})$
Original image				
2 Clusters (random)				
2 Clusters (K-Means++)				
3 Clusters (random)				
3 Clusters (K-Means++)				
4 Clusters (random)				
4 Clusters (K-Means++)				

> Click the result image to see the GIF file (make sure that folder `images33` and `images43` are preserved).

Spectral Clustering

Here are some observations:

1. Same as Kernel K-Means, the higher value of γ_c means that we have more concern of color information. In experiment results (especially for `image1.png`) the clustering result is more sparse than the other one (there are some "salt and pepper" in the case $\gamma_c = 10^{-3}$).

2. Without normalization of Graph Laplacian, one may occasionally trap into an unbalanced result (see Ratio-Cut in 3 and 4 clusters with K-Means++ algorithm).
3. Due to K-Means algorithm, we can see data points having same label cluster in the eigenspace of Graph Laplacian. Note that for ratio-cut cases we can see unbalanced numberical difference of coordinates of the first and the second non-null eigenvectors.

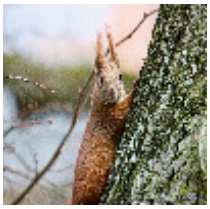
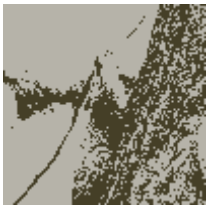
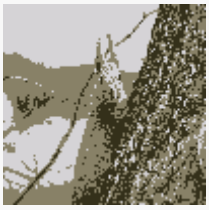
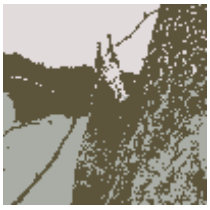
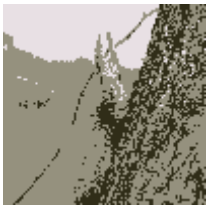
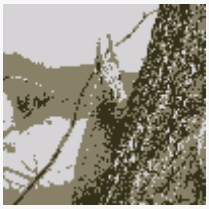
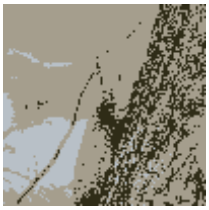
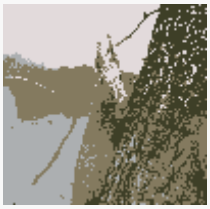
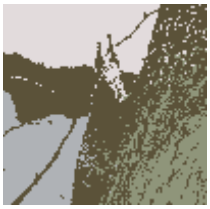
	image1.png $(\gamma_c, \gamma_s) = (10^{-4}, 10^{-3})$	image1.png $(\gamma_c, \gamma_s) = (10^{-3}, 10^{-3})$	image2.png $(\gamma_c, \gamma_s) = (10^{-4}, 10^{-3})$	image2.png $(\gamma_c, \gamma_s) = (10^{-3}, 10^{-3})$
Original image				
2 Clusters (Ratio-Cut, random)				
2 Clusters (N-Cut, random)				
2 Clusters (Ratio-Cut, K-Means++)				
2 Clusters (N-Cut, K-Means++)				
3 Clusters (Ratio-Cut, random)				
3 Clusters (N-Cut, random)				
3 Clusters (Ratio-Cut, K-Means++)				
3 Clusters (N-Cut, K-Means++)				

	image1.png $(\gamma_c, \gamma_s) = (10^{-4}, 10^{-3})$	**image1.png** $(\gamma_c, \gamma_s) = (10^{-3}, 10^{-3})$	**image2.png** $(\gamma_c, \gamma_s) = (10^{-4}, 10^{-3})$	**image2.png** $(\gamma_c, \gamma_s) = (10^{-3}, 10^{-3})$
4 Clusters (Ratio-Cut, random)				
4 Clusters (N-Cut, random)				
4 Clusters (Ratio-Cut, K-Means++)				
4 Clusters (N-Cut, K-Means++)				

> Click the result image to see the GIF file (make sure that folder `images33` and `images43` are preserved).

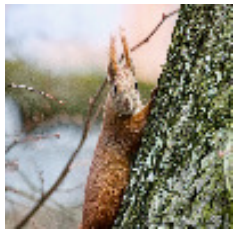
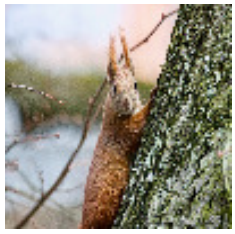
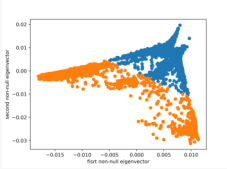
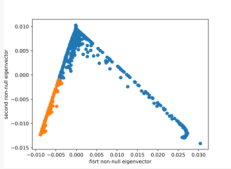
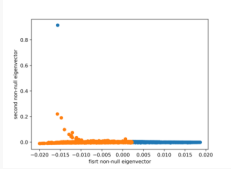
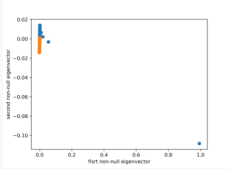
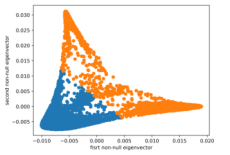
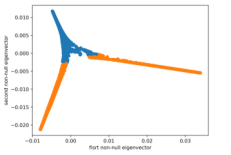
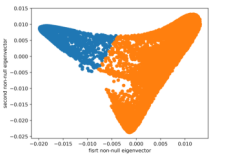
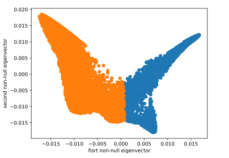
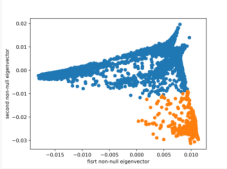
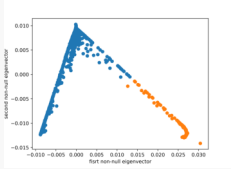
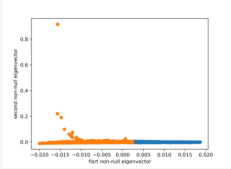
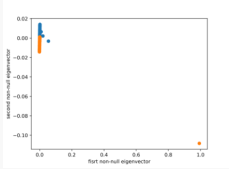
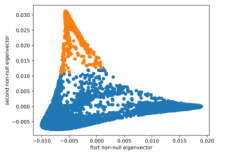
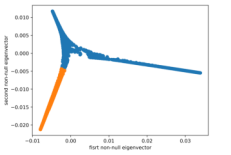
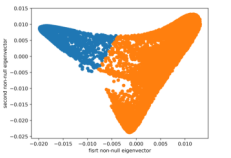
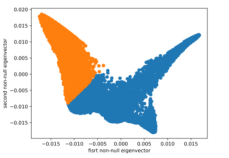
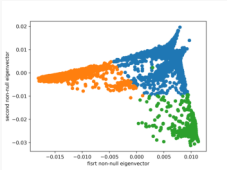
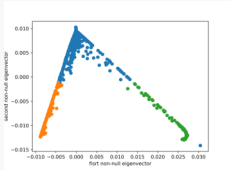
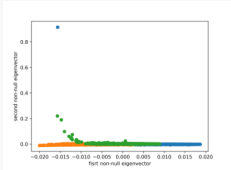
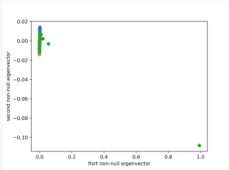
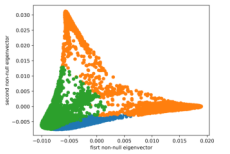
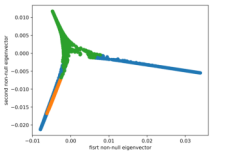
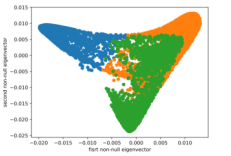
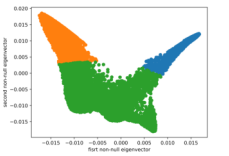
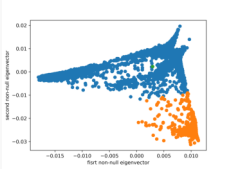
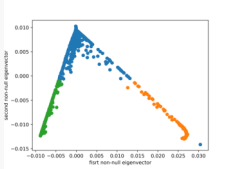
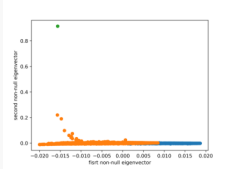
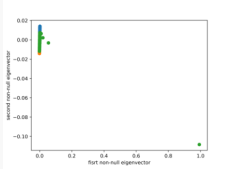
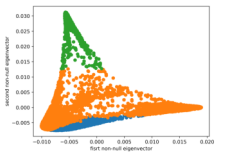
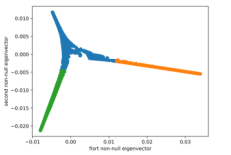
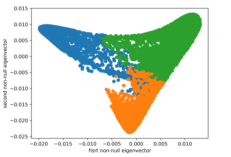
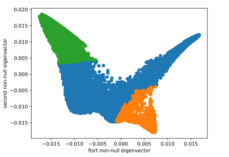
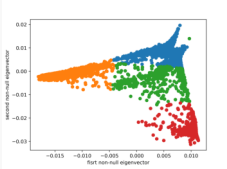
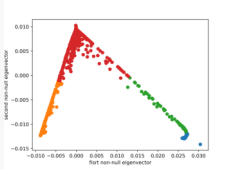
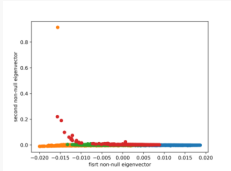
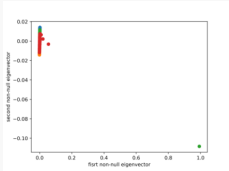
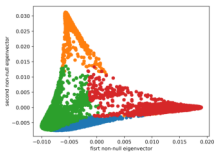
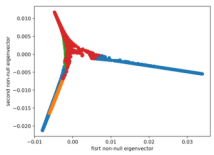
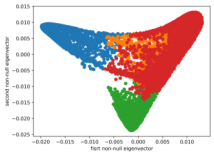
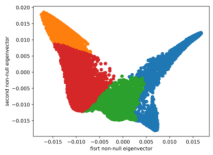
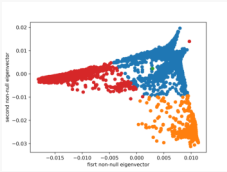
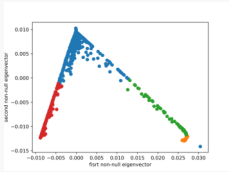
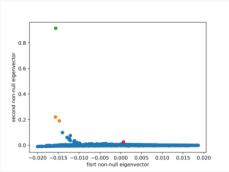
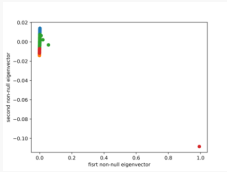
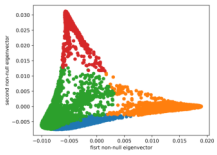
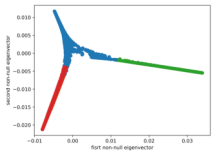
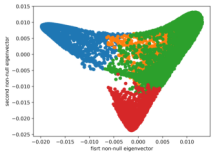
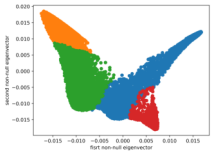
	image1.png $(\gamma_c, \gamma_s) = (10^{-4}, 10^{-3})$	**image1.png** $(\gamma_c, \gamma_s) = (10^{-3}, 10^{-3})$	**image2.png** $(\gamma_c, \gamma_s) = (10^{-4}, 10^{-3})$	**image2.png** $(\gamma_c, \gamma_s) = (10^{-3}, 10^{-3})$
Original image				
2 Clusters (Ratio-Cut, random)				
2 Clusters (N-Cut, random)				
2 Clusters (Ratio-Cut, K-Means++)				
2 Clusters (N-Cut, K-Means++)				
3 Clusters (Ratio-Cut, random)				
3 Clusters (N-Cut, random)				
3 Clusters (Ratio-Cut, K-Means++)				
3 Clusters (N-Cut, K-Means++)				
4 Clusters (Ratio-Cut, random)				

	image1.png $(\gamma_c, \gamma_s) = (10^{-4}, 10^{-3})$	**image1.png** $(\gamma_c, \gamma_s) = (10^{-3}, 10^{-3})$	**image2.png** $(\gamma_c, \gamma_s) = (10^{-4}, 10^{-3})$	**image2.png** $(\gamma_c, \gamma_s) = (10^{-3}, 10^{-3})$
4 Clusters (N-Cut, random)				
4 Clusters (Ratio-Cut, K-Means++)				
4 Clusters (N-Cut, K-Means++)				

References

1. Dhillon, I. S., Guan, Y., & Kulis, B. (2004, August). Kernel k-means: spectral clustering and normalized cuts. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 551-556). ↩

2. Course slides. IOC5191 Machine Learning. NCTU. ↩

3. Arthur, D., & Vassilvitskii, S. (2006). *k-means++: The advantages of careful seeding*. Stanford. ↩