\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Contents:**

**0. Preparations**

**1. Code with detailed explanations**

    i. Part 1 code with explanations

a) Kernel K-Means

b) Spectral clustering with Normalized Cut

c) Spectral clustering with Ratio Cut

    ii. Part 2 code with explanations

…..

    iii. Part 3 code with explanations

…..

    iv. Part 4 code with explanations

…..

**2. Experiments settings and results & discussion**

    i. Part 1 result & discussion

…..

    ii. Part 2 result & discussion

…..

    iii. Part 3 result & discussion

…..

    iv. Part 4 result & discussion

…..

**3. Observations and discussion**

    i. Compare the performance between different clustering methods.

…..

    ii. Compare the execution time of different settings.

…..

    iii. Anything you want to discuss.

…..

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Note: Important pieces are also explained directly by commenting inside the code for better clarity.**

**I provide extensive commenting in my codes, not just copy paste, to make it easier to understand what I am doing!**

0. Preparations **(The code is in python programming language)**

First step is to mount the drive from where I will access all the input files for my code:

0001.

from google.colab import drive

drive.mount("/content/drive")

Then I added a custom print function, preprint(), to save the outputs of print() in a txt file:

0002.

# Define a custom "pseudo personal print" function

def perprint(message, end = None, file = output\_file):

    print(message, end = end)

    print(message, end = end, file = file)

Homework Objective:

*Use whatever your favorite language to implement kernel k-means, spectral clustering (both normalize cut and ratio cut), using a given defined new kernel, on two 100\*100 images provided (image1.png & image2.png).*

1. Code with detailed explanations

**Note: Important pieces are also explained directly by commenting inside the code for better clarity.**

We first need to import all the required libraries for clarity at the top of the code:

0003.

import os

import time

import numpy as np

import random

from PIL import Image

from scipy.spatial.distance import pdist, cdist

import imageio.v2 as imageio

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

import matplotlib.cm as cm

Here is what they each do:

os: Used for particularly for path manipulations and directory operations.

time: Useful for tracking the execution time of different parts of the code.

numpy: Very very essential for numerical operations, especially matrix operations here.

random: For generating random numbers, used in initialization steps.

PIL.Image: Python Imaging Library for opening, manipulating, and saving different image file formats.

scipy.spatial.distance: Contains functions to compute distances between points, very helpful.

imageio.v2: Will help us create the GIFs later.

matplotlib.pyplot and mpl\_toolkits.mplot3d: For plotting and visualizing data, essential for displaying clustering results.

Matplotlib.cm: Provides a large set of colormaps, useful for visualizing clusters.

Setting Paths and Directories:

* Paths for input images and output directories are set. This organization facilitates managing input data and storing results.

0004.

# Variables for input image paths

path\_image\_1 = '/content/drive/MyDrive/NYCU/00- Sem 01 - Fall 2023/Machine Learning/HMW\_06/Input\_Data/image1.png'

path\_image\_2 = '/content/drive/MyDrive/NYCU/00- Sem 01 - Fall 2023/Machine Learning/HMW\_06/Input\_Data/image2.png'

# Variables for output directories

dir\_kernel\_K\_means = '/content/drive/MyDrive/NYCU/00- Sem 01 - Fall 2023/Machine Learning/HMW\_06/My\_Draft\_Code/Kernel\_K\_Means'

dir\_spectral\_clustering\_normalized\_cut = '/content/drive/MyDrive/NYCU/00- Sem 01 - Fall 2023/Machine Learning/HMW\_06/My\_Draft\_Code/Spectral\_Clustering\_Normalized\_Cut'

dir\_spectral\_clustering\_ratio\_cut = '/content/drive/MyDrive/NYCU/00- Sem 01 - Fall 2023/Machine Learning/HMW\_06/My\_Draft\_Code/Spectral\_Clustering\_Ratio\_Cut'

    i. Part 1 code with explanations

a) Kernel K-Means: Clustering & Visualization

I built the kernel k-means algorithm as a combination of many defined functions for clarity.

The first function is for reading the images one at a time from the path, taken as input, where the image is located. The function returns img, a variable with an array of shape (100\*100, 3), 3 channels RBG image, digitally representing the image read.

0005.

# Function for reading images

def read\_image(image\_path):

  ''' Read image from path '''

  img\_ = Image.open(image\_path) # RGB

  ''' Flatten the image <==> color information C(x),

      instead of np.array(img\_) <==> (100, 100, 3) '''

  img = np.array(img\_.getdata()) # Shape (10000, 3)

  return img

Explanation of the Custom Kernel Matrix Computation Function Below, which computes the Gram matrix.

Here we implement the computation of the distance similarity based on the new kernel defined below:



where S(x) is the spatial information of data x , and C(x) is the color information of data x. Both γs and γc are hyper-parameters.

You can see in the comments inside the code how each part is pecifically implemented.

S\_x = np.hstack((grid[0], grid[1])): Combines the x and y coordinates (originally the image is 100\*100\*3, i.e. x\*y\*depth) into a single array, representing the spatial information of each pixel.

C(x) is the same as the variable returned by the read\_image() function above.

0006.

# Function to compute the custom kernel matrix, the Gram matrix

def compute\_kernel(x, gamma\_s, gamma\_c):

    '''

    k(𝑥, 𝑥′) = 𝑒xp( −𝛾\_𝑠\* ∥𝑆(𝑥)− 𝑆(𝑥′)∥^2 ) \* 𝑒xp(−𝛾\_𝑐\* ∥𝐶(𝑥)− 𝐶(𝑥′)∥^2 )

             = 𝑒xp( −( 𝛾\_𝑠 \* ∥𝑆(𝑥)− 𝑆(𝑥′)∥^2 + 𝛾\_𝑐 \* ∥𝐶(𝑥)− 𝐶(𝑥′)∥^2 ) )

    𝛾\_𝑠 = gamma\_s

    𝛾\_c = gamma\_c

    𝑆(𝑥) : Spatial information (i.e. the coordinate of the pixel) of data 𝑥.

    𝐶(𝑥) : Color information (i.e. the RGB values) of data 𝑥, == x

    '''

    grid = np.indices((100, 100)).reshape(2, 10000, 1)

    ''' Spatial information '''

    S\_x = np.hstack((grid[0], grid[1]))

    ''' ∥𝑆(𝑥)− 𝑆(𝑥′)∥^2 '''

    dist\_s = cdist(S\_x, S\_x, 'sqeuclidean')

    ''' Color information, reshape to (10000, 3) '''

    img\_flat = x.reshape(-1, x.shape[-1])

    ''' ∥𝐶(𝑥)− 𝐶(𝑥′)∥^2 '''

    dist\_c = cdist(x, x, 'sqeuclidean')

    ''' k(𝑥, 𝑥′) '''

    kernel = np.exp( - (gamma\_s \* dist\_s + gamma\_c \* dist\_c ) )

    return kernel

This function sq\_kernel\_distance calculates the squared distance in the feature space using a precomputed kernel matrix, which will be obtained from the function above in code 0006. Here's a brief explanation:

This function takes three parameters:

kernel: The precomputed kernel matrix, generated using the compute\_kernel function.

i: Index of the first data point.

j: Index of the second data point.

The squared distance between two data points in the feature space is calculated using the formula:

 and thus:



ϕ(xi​) and ϕ(xj​) are the ) are the feature representations of data points at indices xi and xj.

∥⋅∥ denotes the Euclidean norm, representing the distance between the feature representations.

The function then returns this squared distance.

0007.

# Function to compute the squared distance in the feature space

def sq\_kernel\_distance(kernel, i, j):

    distance\_squared = kernel[i, i] - 2 \* kernel[i, j] + kernel[j, j]

    return distance\_squared

Sure, initial\_centroids initializes K centroids for all the algorithms in this homework. If mode is 0, it selects them randomly; otherwise, it uses K-means++. The function returns an array of centroid indices.

The comments in the code give exact explanations of each step.

0008.

# Function for initializing the centroids

def initial\_centroids(kernel, K, number\_data\_points, mode):

    ''' K is the number of centroids'''

    if mode == 0:

        ''' Random strategy: Randomly select K centroids in the data points '''

        centroids = np.random.choice(number\_data\_points, K, replace=False) # False avoids choosing same indice twice

        return centroids

    else:

        ''' K-means++ for initializing centroids '''

        ''' First randomly select the first cluster center '''

        centroids = list(random.sample(range(0,number\_data\_points), 1))

        ''' Then iterate to select the remaining centroids '''

        for number\_center in range(1, K):

            ''' np.inf for positive infinite '''

            min\_dist = np.full(number\_data\_points, np.inf)

            for i in range(number\_data\_points):

                for j in range(number\_center):

                    ''' ∥ϕ(xi​) − μ\_j​∥^2 , μ\_j is center of cluster j '''

                    dist = sq\_kernel\_distance(kernel, i, centroids[j])

                    if dist < min\_dist[i]:

                        min\_dist[i] = dist

            min\_dist /= np.sum(min\_dist)

            ''' Will choose a single point which is biased by the probabilities p'''

            centroids.append(np.random.choice(np.arange(number\_data\_points), 1, p=min\_dist)[0])

        return np.array(centroids)

initial\_assignment assigns each data point to the nearest centroid in the feature space, based on precomputed distances in the Gram matrix (kernel). It returns an array of initial cluster assignments. This function is used by kernel k-means clustering algorithm. A clear explanation of each step can be found in the code’s comments.

0009.

# Function for the initial cluster assignment

def initial\_assignment(kernel, K, initial\_centroids, number\_data\_points):

    ''' Assign each data point to the nearest centroid in the feature space '''

    ''' Initialize an array to store the cluster assignments '''

    initial\_clusters = np.zeros(number\_data\_points, dtype = int)

    ''' Iterate over each data point '''

    for i in range(number\_data\_points):

        ''' Initialize distances for K clusters '''

        dist = np.zeros(K)

        ''' Iterate over each centroid '''

        for j in range(K):

            ''' Distance from the data point to this centroid '''

            dist[j] = sq\_kernel\_distance(kernel, i, initial\_centroids[j])

        ''' Update the cluster assignment for the data point '''

        initial\_clusters[i] = np.argmin(dist)

    return initial\_clusters

In the kernel k-means algorithm, after the initial assignment based on actual data points, we will no longer compute the center of clusters so we will be using a kernel-based method to determine instead which cluster each data point belongs to without computing any centroids.

subsequent\_assignment updates cluster assignments in kernel K-means. It precomputes cluster sums, sizes, and diagonal kernel elements. Then, it calculates squared distances to centroids and assigns new clusters.

Returns updated assignments.

It is based on this formula:  
A math equations on a white background

Description automatically generated

A black background with white text

Description automatically generated

Please take a look at the explicit commenting in the code snippet for detailed implementation.

0010.

# Function for subsequent cluster assignments

def subsequent\_assignment(kernel, clusters, K, number\_data\_points):

    ''' Initialize an array to store the cluster assignments '''

    new\_clusters = np.zeros(number\_data\_points, dtype=int)

    ''' Pre-compute the sum of kernel values within each cluster '''

    cluster\_kernel\_sums = np.array([np.sum(kernel[np.ix\_(clusters == k, clusters == k)]) for k in range(K)])

    ''' Sums the True's, number of elements (clusters == k) in each cluster k '''

    cluster\_sizes = np.array([np.sum(clusters == k) for k in range(K)])

    ''' Diagonal elements (self-similarity or kernel[i, i]) for each data point '''

    diag\_kernel = np.diag(kernel)

    ''' Iterate over each cluster for all data points at once '''

    dist = np.full((number\_data\_points, K), np.inf)

    ''' Iterate over each centroid '''

    for k in range(K):

        ''' Make sure there is no division by 0 '''

        if cluster\_sizes[k] > 0:

            ''' Square kernel distance to centroid k '''

            sum\_kernel\_to\_cluster = np.sum(kernel[:, clusters == k], axis=1)

            dist[:, k] = diag\_kernel - (2 / cluster\_sizes[k]) \* sum\_kernel\_to\_cluster \

                        + (1 / cluster\_sizes[k]\*\*2) \* cluster\_kernel\_sums[k]

    ''' Assign new clusters by choosing index of minimum distance '''

    new\_clusters = np.argmin(dist, axis=1)

    return new\_clusters

write\_image generates and saves color images based on cluster assignments. It uses predefined and random colors for clusters. The colors are actually not that random, I experimented with a lot of combinations.

Parameters include the number of data points, clusters, model identifier, image number, and output directory. Please look at the comments for a very detailed explanation.

0011.

# Function for generating color images based on clusters

def write\_image(number\_data\_points, number\_clusters, clusters, model, image\_number, output\_dir):

    ''' Colors to assign to each cluster '''

    colors = np.array([[150, 255, 200],

                       [0, 255, 0],

                       [0, 0, 255],

                       [0, 200, 255],

                       [255, 200, 0],

                       [255, 0, 150]])

    if number\_clusters > 3:

        colors = np.append(colors, np.random.choice(256, (number\_clusters - 3, 3)), axis = 0)

    ''' Assigning colors to datapoints '''

    data = colors[clusters]

    ''' Reshape data points to look like an image '''

    image = data.reshape(100, 100, 3)

    ''' Create an image with the data points '''

    image = Image.fromarray(np.uint8(image))

    ''' Save the created image '''

    image.save(os.path.join(output\_dir, f"{model}{image\_number:02d}.png"))

The compose\_gif function creates and saves a GIF from a series of images.

Parameters:

model: A string identifier for the model.

number\_images: The total number of images to include in the GIF.

output\_dir: Directory to save the generated GIF.

It iterates over the images, reads each image path, and appends them to a list.

The list of images is then transformed into a GIF and saved in the specified output directory with the model identifier.

The frames per second (fps) for the GIF are set to 1.

0012.

# Function for saving GIF images

def compose\_gif(model, number\_images, output\_dir):

    ''' Temporary list to save the images read in directory'''

    gif\_images = []

    ''' Iterate to read images '''

    for i in range(number\_images + 1):

        ''' Path of image '''

        gif\_image\_path = os.path.join(output\_dir, f"{model}{i:02d}.png")

        ''' Read image into the list '''

        gif\_images.append(imageio.imread(gif\_image\_path))

    ''' Transform and save list as GIF inside the directory '''

    imageio.mimsave(os.path.join(output\_dir, f"{model}.gif"), gif\_images, fps=1)

Below is the function that will be called, and it will lead to all the above functions being called as well.

As you can see with the support of the comments in the code, it is running by mostly calling upon other functions we already defined above.

The kernel\_k\_means function performs Kernel K-Means clustering on a given image.

Parameters:

image\_number: Identifier for the image.

kernel: Precomputed kernel matrix.

K: Number of clusters.

max\_iter: Maximum number of iterations.

mode: Mode for initializing centroids.

directory: Directory to save the results.

It initializes centroids and performs the initial cluster assignment.

Iterates for a maximum of max\_iter times, updating cluster assignments, saving images for each iteration, and checking for convergence.

Creates a GIF from the saved images and saves it in the specified directory with a filename based on the parameters and image number.

By calling kernel\_k\_means and passing number of clusters K=2, mode = 0 (random centroid initialization) and the other parameters, we effectifely complete Part 1 for kernel k-means.

0013.

# Function for computing  Kernel K-Means

def kernel\_k\_means(image\_number, kernel, K, max\_iter, mode, directory):

    perprint(f"\t\tImage {image\_number} \n\t\t\tStarting Kernel K-Means with K = {k} ...")

    ''' Initializing the centroids '''

    Total = kernel.shape[0]

    centroids = initial\_centroids(kernel, K, Total, mode)

    ''' Initial cluster assignment '''

    clusters = initial\_assignment(kernel, K, centroids, Total)

    ''' Loop for cluster assignment update and save images for GIF'''

    for iter in range(max\_iter):

        ''' Update cluster assignment '''

        new\_clusters = subsequent\_assignment(kernel, clusters, K, Total)

        ''' Create image based on the new\_clusters '''

        write\_image(Total, K, new\_clusters, f'{mode}\_{image\_number}\_Kernel\_{K}\_Means', iter, directory)

        ''' Check for convergence '''

        if np.array\_equal(new\_clusters, clusters):

            last\_iteration = iter

            break

        last\_iteration = iter

        clusters = new\_clusters

    ''' Create GIF from the saved images and save it '''

    compose\_gif(f'{mode}\_{image\_number}\_Kernel\_{K}\_Means', last\_iteration, directory)

    print(f"\t\t\tFinished!")

b) Spectral clustering with Normalized Cut

The compute\_laplacian function computes the Laplacian matrix based on normalized spectral clustering (Ng, Jordan, Weiss, 2002).

Parameters:

W: Similarity matrix.

symmetric: A boolean for symmetric normalized Laplacian (default is True).

Computes the degree matrix D and the unnormalized Laplacian L=D−W.

If symmetric is True, efficiently applies symmetric normalization.

Returns the Laplacian matrix.

0014.

# Normalized spectral clustering according to Ng, Jordan, and Weiss (2002)

def compute\_laplacian(W, symmetric=True):

    ''' Degree Matrix D '''

    D = np.diag(np.sum(W, axis = 1))

    ''' Unnormalized Laplacian '''

    L = D - W

    if symmetric:

        ''' Symmetric Normalized Laplacian '''

        ''' Efficient computation of D^(-1/2) '''

        D\_inv\_sqrt = np.sqrt(1 / D.diagonal())

        ''' Element-wise multiplication by D^(-1/2) for rows and columns '''

        L\_sym = D\_inv\_sqrt[:, np.newaxis] \* L \* D\_inv\_sqrt

        return L\_sym

    return L

The eigen\_decomposition function performs eigenvalue decomposition on a Laplacian matrix and retrieves the corresponding eigenvector matrix.

Parameters:

Laplacian: The Laplacian matrix.

num\_clusters: Number of clusters.

Computes eigenvalues and eigenvectors using NumPy's np.linalg.eigh.

Selects the eigenvectors corresponding to the smallest non-zero eigenvalues, skipping the first eigenvalue (which is always 0).

Returns the eigenvector matrix U.

0015.

# Function for Eigenvalue Decomposition and get Eigenvector Matrix

def eigen\_decomposition(Laplacian, num\_clusters):

    ''' Compute the eigenvalues and eigenvectors '''

    eigenvalues, eigenvectors = np.linalg.eigh(Laplacian)

    ''' Select the eigenvectors corresponding to the smallest

        non-zero eigenvalues '''

    '''     while skipping the first eigenvalue because = 0 '''

    indices = np.argsort(eigenvalues)[1: num\_clusters + 1]

    U = eigenvectors[:, indices]

    return U

The normalize\_rows function normalizes the rows of a matrix, ensuring each row has a unit Euclidean norm of 1.

Parameters:

U: The matrix to be normalized.

Normalizes each row of the matrix by dividing it by its Euclidean norm.

Returns the normalized matrix.

0016.

# Function to Normalize the Rows of the Eigenvector Matrix

def normalize\_rows(U):

    ''' Normalize each row to have unit Euclidean norm 1 '''

    U\_normalized = U / np.linalg.norm(U, axis=1, keepdims=True)

    return U\_normalized

The clustering function assigns each data point to the nearest center based on squared Euclidean distances.

Parameters:

array\_: The data points.

centers: The center points.

Calculates squared Euclidean distances from each data point to each center.

Assigns each data point to the nearest center.

Returns an array of cluster assignments for each data point.Top of Form

0017.

# Function to assign each data point the nearest center

def clustering(array\_, centers):

    ''' Calculate distances from each point to each center '''

    distances = cdist(array\_, centers, 'sqeuclidean')

    ''' Assign each point to the nearest center '''

    clusters = np.argmin(distances, axis=1)

    return clusters

Herer comes a very long function, the longest one in the entire code.

The k\_means function performs K-Means clustering within the Spectral Clustering framework.

Parameters:

matrix: The data matrix.

num\_clusters: Number of clusters.

iterations: Maximum number of K-Means iterations.

output\_dir: Directory to save results.

mode: A mode identifier.

image\_number: Identifier for the image.

Initializes centroids and associates them with centers.

Assigns clusters based on the initial centroids and creates/saves an image.

Iteratively updates clusters by performing K-Means, creating/saving images, and checking for convergence.

Creates a GIF from the saved images and returns the final cluster assignments.

Please look into it for a very detailed explanation of the implementation.

0018.

# Function for performing K-Means in Spectral Clustering

def k\_means(matrix, num\_clusters, iterations, output\_dir, mode, image\_number, title):

    if mode == 0:

        title = 'ratio'

    else:

        title = 'normalized'

    Total = len(matrix)

    ''' Get indices of centroids '''

    centroids = initial\_centroids(kernel, num\_clusters, Total, mode)

    ''' Associate centroids to corresponding centers'''

    centers = []

    for i in range(num\_clusters):

        centers.append(matrix[centroids[i]])

    centers = np.array(centers)

    ''' Assign clusters '''

    clusters = clustering(matrix, centers)

    ''' Create and save image based on clusters '''

    write\_image(Total, num\_clusters, clusters, f'{mode}\_{image\_number}\_{title}\_{num\_clusters}\_', 0, output\_dir)

    ''' Iterate over loop to update clusters by K-Means '''

    for i in range(1, iterations):

        ''' Initialize new\_clusters as empty list '''

        new\_centers = []

        for j in range(num\_clusters):

            ''' Find "indices" that match with the cluster j '''

            mask = clusters == j

            ''' Update a new center as a mean of the data points in cluster j '''

            new\_centers.append(np.sum(matrix[mask], axis=0) / len(matrix[mask]))

        ''' Transform the list into an array '''

        new\_centers = np.array(new\_centers)

        ''' Create and save image based on new\_clusters '''

        new\_clusters = clustering(matrix, new\_centers)

        write\_image(Total, num\_clusters, clusters, f'{mode}\_{image\_number}\_{title}\_{num\_clusters}\_', i, output\_dir)

        ''' Check for convergence '''

        if(np.linalg.norm((centers - new\_centers), ord=2) < 1e-2):

            last\_iteration = i

            break

        last\_iteration = i

        centers = new\_centers

    ''' Create GIF from the saved images and save it '''

    compose\_gif(f'{mode}\_{image\_number}\_{title}\_{num\_clusters}\_', last\_iteration, output\_dir)

    return new\_clusters

This function is for bringing everything together and doing spectral clustering with Normalized Cut. It also mostly calls on other functions, just like kernel k-means function.

You may notice that there is the option of saving matrix U, the eigenvectors matrix, because it takes time to compute that matrix. So I save it and reload it as needed during training.

The spectral\_clustering\_normalized function computes Spectral Clustering with Normalized Cut.

Parameters:

matrix\_kernel: The kernel matrix.

W: Affinity matrix.

K: Number of clusters.

iterations: Maximum number of K-Means iterations.

mode: A mode identifier.

image\_number: Identifier for the image.

dir\_normalized\_cut: Directory for saving results.

Steps:

Compute Symmetric Normalized Laplacian Matrix: This step can be commented out.

Eigenvalue Decomposition: Compute the eigenvector matrix U containing the k smallest eigenvectors. It can also be commented out if already saved.

Save/Load U: For rapid testing, U is saved and loaded based on the image index.

Normalize Rows: Normalize the rows of U to create matrix T.

Apply K-Means: Use K-Means on the normalized U normalized or U\_norm.

Returns the final cluster obtained from K-Means clustering and the U\_norm, as well as U itself.

By calling spectral\_clustering\_normalized and passing number of clusters K=2, mode = 0 (random centroid initialization) and the other parameters, we effectifely complete Part 1 for spectral clustering with normalized cut.

0019.

# Function to compute Spectral Clustering Normalized Cut

def spectral\_clustering\_normalized(matrix\_kernel, W, K, iterations, mode, image\_number, dir\_normalized\_cut):

    ''' Step 1: Compute Symmetric Normalized Laplacian Matrix '''

    L\_sym = compute\_laplacian(W, symmetric=True)

    ''' Step 2: Eigenvalue Decomposition '''

    ''' Eigenvector Matrix containing k smallest eigenvectors of L\_sym '''

    U = eigen\_decomposition(L\_sym, K)

    ''' Save U for rapid code testing '''

    if image\_number == 1:

        np.save(os.path.join(dir\_normalized\_cut, "\_1\_normalized\_eigenvector.npy"), U)

        pass

    elif image\_number == 2:

        np.save(os.path.join(dir\_normalized\_cut, "\_2\_normalized\_eigenvector.npy"), U)

        pass

    else:

        perprint("\nYour input image index is neither 1 nor 2. \n ")

    ''' Load U for rapid code testing '''

    if image\_number == 1:

        U = np.load(os.path.join(dir\_normalized\_cut, "\_1\_normalized\_eigenvector.npy"))

    elif image\_number == 2:

        U = np.load(os.path.join(dir\_normalized\_cut, "\_2\_normalized\_eigenvector.npy"))

    else:

        perprint("\nYour input image index is neither 1 nor 2. \n ")

    ''' Step 3: Normalize Rows, this is matrix T in the algorithm '''

    U\_norm = normalize\_rows(U) # AKA matrix T

    ''' Step 4: Apply K-means '''

    cluster = k\_means(U\_norm, K, iterations, dir\_normalized\_cut, mode, image\_number)

    ''' Return final cluster obtained from K-Means clustering and U '''

    return cluster, U

c) Spectral clustering with Ratio Cut

spectral\_clustering\_ratio used all the same functions as spectral\_clustering\_normalized defined above in code 0020.  
The only differences are:

- it passes symmetric = False to compute\_laplacian defined in code 0014, which will make it return the unnormalized Laplacian matrix L = D-W.

- it doesn’t normalize U, but passes it directly to k-means define in 0018.

Those are the only differences.

By calling spectral\_clustering\_ratio and passing number of clusters K=2, mode = 0 (random centroid initialization) and the other parameters, we effectifely complete Part 1 for spectral clustering with ratio cut.

0020.

# Function to compute Spectral Clustering Ratio Cut

def spectral\_clustering\_ratio(matrix\_kernel, W, K, iterations, mode, image\_number, dir\_ratio\_cut):

    ''' Step 1: Compute unnormalized Laplacian Matrix '''

    L = compute\_laplacian(W, symmetric=False)

    ''' Step 2: Eigenvalue Decomposition '''

    ''' Eigenvector Matrix containing k smallest eigenvectors of L '''

    U = eigen\_decomposition(L, K)

    ''' Save U for rapid code testing '''

    if image\_number == 1:

        np.save(os.path.join(dir\_ratio\_cut, "\_1\_ratio\_eigenvector.npy"), U)

        pass

    elif image\_number == 2:

        np.save(os.path.join(dir\_ratio\_cut, "\_2\_ratio\_eigenvector.npy"), U)

        pass

    else:

        perprint("\nYour input image index is neither 1 nor 2. \n ")

    ''' Load U for rapid code testing '''

    if image\_number == 1:

        U = np.load(os.path.join(dir\_ratio\_cut, "\_1\_ratio\_eigenvector.npy"))

    elif image\_number == 2:

        U = np.load(os.path.join(dir\_ratio\_cut, "\_2\_ratio\_eigenvector.npy"))

    else:

        perprint("\nYour input image index is neither 1 nor 2. \n ")

    ''' Step 3: Apply K-means '''

    cluster = k\_means(U, K, iterations, dir\_ratio\_cut, mode, image\_number)

    ''' Return final cluster obtained from K-Means clustering and U '''

    return cluster, U

    ii. Part 2 code with explanations: Try more clusters.

I built:

kernel\_k\_means in code 0013,

spectral\_clustering\_normalized in code 0019,

and spectral\_clustering\_ration in 0020,

all with the common parameters mode and K, where mode is either 0 or 1, and K can take values of 2, 3, 4 or greater.

K is the number of clusters,

mode = 0 is for random inititialization of centroids, and

mode = 1 is for initializing the centroids with k-means++ as explained in code 008 with function initial\_centroids, which is shared by all 3 algorithms.

Now with that clear, to perform Part 2 we simply call again those functions by passing:

mode = 0 every time, and K = 3 then K = 4.

    iii. Part 3 code with explanations: Try different initializations.

For Part3, we follow the same explanation given in Part 2.

Now we set mode = 1 every time and K = 2, then 3, then 4.

mode = 1 will initialize the centroids with k-means++.

    iv. Part 4 code with explanations

For both spectral clustering with Normalized Cut and spectral clustering with Ratio Cut, to examine whether the data points within the same cluster do have the same coordinates in the eigenspace of graph Laplacian or not we implement this new function.

The plot\_eigenspace\_clusters function visualizes clusters in the eigenspace.

So we simply need to call this function and pass the parameters as indicated for normalized and ratio cut.

Parameters:

U: The eigenvectors matrix.

clusters: Cluster assignments.

K: Number of clusters.

title: Title for the plot.

output\_dir: Directory to save the plot.

image\_number: Identifier for the image.

Selects the first two or three eigenvectors for plotting.

Generates a color map based on the number of clusters.

Plots each cluster in the eigenspace.

Labels axes and adds a title.

Adds a legend.

Saves and displays the plot.

0021.

# Function to plot eigenspace clusters

def plot\_eigenspace\_clusters(U, clusters, K, title, output\_dir, image\_number):

    ''' Select the first two or three eigenvectors for plotting '''

    if U.shape[1] >= 3:

        ''' for 3D plotting '''

        eig\_vecs = U[:, :3]

        fig = plt.figure()

        ax = fig.add\_subplot(111, projection='3d')

    else:

        ''' for 2D plotting '''

        eig\_vecs = U[:, :2]

        fig, ax = plt.subplots()

    ''' Generate a color map '''

    cmap = plt.colormaps['viridis']

    colors = [cmap(i / K) for i in range(K)]

    ''' Plot each cluster '''

    for i in range(K):

        cluster\_points = eig\_vecs[clusters == i]

        ax.scatter(\*cluster\_points.T, label=f'Cluster {i+1}')

    ''' Label axes and add title '''

    ax.set\_xlabel('Eigenvector 1')

    ax.set\_ylabel('Eigenvector 2')

    if eig\_vecs.shape[1] == 3:

        ax.set\_zlabel('Eigenvector 3')

    ax.set\_title(title)

    ''' Add a legend '''

    ax.legend()

    ''' Save the plot '''

    image\_path = os.path.join(output\_dir, f"{mode}\_{image\_number}\_{title}\_eigenspace\_cluster.png")

    plt.savefig(image\_path)

    plt.show()

2. Experiments settings and results & discussion

    i. Part 1 result & discussion

As explained in functions at code 0013 (kernel\_k\_means), at code 0019 (spectral\_clustering\_normalized), and at code 0020 (spectral\_clustering\_ratio), the settings for part 1 are to simply use K= 2, mode = 0 and call the other parameters as instructed.

We also use gamma\_s (or s = 0.0001) and gamma\_c (or c = 0.001), hyperparameters of the compute\_kernel function defined in code 006.

These hyperparameter values were found to give the best results empirically.

Here are the results of running the experiment for K = 2 and with random initialization (mode =0):

Table part1\_1

|  |  |  |
| --- | --- | --- |
| 0\_1\_Kernel\_2\_Means.gif | 0\_1\_Ncut\_2\_.gif | 0\_1\_RatioCut\_2\_.gif |
| Kernel k-means | Normalized cut | Ratio cut |
|  |  |  |

Table part1\_2

|  |  |  |
| --- | --- | --- |
| 0\_2\_Kernel\_2\_Means.gif | 0\_2\_Ncut\_2\_.gif | 0\_2\_RatioCut\_2\_.gif |
| Kernel k-means | Normalized cut | Ratio cut |
|  |  |  |

    ii. Part 2 result & discussion

As explained in functions at code 0013 (kernel\_k\_means), at code 0019 (spectral\_clustering\_normalized), and at code 0020 (spectral\_clustering\_ratio), the settings for part 2 are to simply use K= {2, 3, 4}, mode = 0 and call the other parameters as instructed.

We also use gamma\_s (or s = 0.0001) and gamma\_c (or c = 0.001), hyperparameters of the compute\_kernel function defined in code 006.

Here are the results of running the experiment for K= {2, 3, 4}, and with random initialization (mode =0):

Table part2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| K |  | Kernel k-means | Normalized cut | Ratio cut |
| 2 | Image1.png  Gifts from left to right:  0\_1\_Kernel\_2\_Means.gif  0\_1\_Ncut\_2\_.gif  0\_1\_RatioCut\_2\_.gif |  |  |  |
| 3 | Image1.png  Gifts from left to right:  0\_1\_Kernel\_3\_Means.gif  0\_1\_Ncut\_3\_.gif  0\_1\_RatioCut\_3\_.gif |  |  |  |
| 4 | Image1.png  Gifts from left to right:  0\_1\_Kernel\_4\_Means.gif  0\_1\_Ncut\_4\_.gif  0\_1\_RatioCut\_4\_.gif |  |  |  |
| 2 | Image2.png  Gifts from left to right:  0\_2\_Kernel\_2\_Means.gif  0\_2\_Ncut\_2\_.gif  0\_2\_RatioCut\_2\_.gif |  |  |  |
| 3 | Image2.png  Gifts from left to right:  0\_2\_Kernel\_3\_Means.gif  0\_2\_Ncut\_3\_.gif  0\_2\_RatioCut\_3\_.gif |  |  |  |
| 4 | Image2.png  Gifts from left to right:  0\_2\_Kernel\_4\_Means.gif  0\_2\_Ncut\_4\_.gif  0\_2\_RatioCut\_4\_.gif |  |  |  |

    iii. Part 3 result & discussion

As explained in functions at code 0013 (kernel\_k\_means), at code 0019 (spectral\_clustering\_normalized), and at code 0020 (spectral\_clustering\_ratio), the settings for part 3 are to simply use K= 4, mode = {0, 1} and call the other parameters as instructed.

We also use gamma\_s (or s = 0.0001) and gamma\_c (or c = 0.001), hyperparameters of the compute\_kernel function defined in code 006.

Here are the results of running the experiment for K= 4, and with random initialization (mode = {0, 1}):

**Image 1.png and K = 4**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Kernel K-means | Normalized cut | Ratio cut |
| Random (mode = 0) |  |  |  |
| K-mean++ (mode =1) |  |  |  |

Table Part 3

**Image 2.png and K = 4**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Kernel K-means | Normalized cut | Ratio cut |
| Random (mode = 0) |  |  |  |
| K-mean++ (mode =1) |  |  |  |

    iv. Part 4 result & discussion

Taking the plot\_eigenspace\_cluster discussed in code 0021, we we will run it after normalized cut finishes and also after ratio cut finishes running.

**Plotting the eigenspace gives this at K = 2.**

|  |  |  |
| --- | --- | --- |
| Image 1:  Eigenspace of graph Laplacian  Normalized cut on the left  and  ratio cut on the right | A graph with orange and blue lines  Description automatically generated |  |
| Image 2:  Eigenspace of graph Laplacian  Normalized cut on the left  and  ratio cut on the right |  |  |

**Plotting the eigenspace gives this at K = 3.**

|  |  |  |
| --- | --- | --- |
| Image 1:  Eigenspace of graph Laplacian  Normalized cut on the left  and  ratio cut on the right |  |  |
| Image 2:  Eigenspace of graph Laplacian  Normalized cut on the left  and  ratio cut on the right |  |  |

3. Observations and discussion

    i. Compare the performance between different clustering methods.

In Part1:

Looking at Table part1\_1 and Table part1\_2, it is obvious that at K=2 and random initialization the kernel k-means performs the best visually.

In second is the spectral clustering with normalized cut and at last is the spectral clustering with ratio cut.

**Thoughts**

Kernel K-Means Dominance: Part 1 analysis indicates that, particularly at K=2 with random initialization, Kernel K-Means outperforms, providing visually superior results compared to Spectral Clustering with Normalized Cut and Ratio Cut.

In Part2:

In part 2, looking at Table part2 where we group results of images 1 and 2 when they run on mode =0, i.e. random initialization, it can also see clearly:

- At K=2, kernel k-means is the best followed by normalized cut.

- At K = 3, looking at image 1 and 2 together we can agree also that kernel k-means is still the best at clustering in a way that gives enough details to inform about the shape of the animal and about the topology of the beach. The normalized cut and the ratio cut seem to be tied but the normalized cut is still better at K =3.

- At K = 4, surprisingly the normalized cut is more balanced than the kernel k-means. At K=4 and in image 1, the kernel k-means is still preferable with more usable clear and crisp details. But looking at image 2 at the same time we find that the cluster assigned to the indigo color is overtaking most of the data points. The ratio cut undoubtably performed the less even at K =4.

**Thoughts**

Part 2 Insights: Kernel K-Means maintains its lead at K=2 and K=3, offering detailed and clear results. However, at K=4, Normalized Cut shows a more balanced performance, especially in image 2.

In Part3:

In part3, we compared image 1 and 2 for K = 4, because overall K=4 seems to have the best results.

So toggling between mode 0 and mode 1, i.e. between random and K-means++ initializations, when looking at image 1 and 2 together, from Table Part 3 we see that the normalized cut definitely seems to be gaining an advantage over the kernel k-means.

The normalized cut becomes best, the kernel k-means second and the ratio cut still ranks last.

**Thoughts**

Mode Comparison (Part 3): Switching between random and K-means++ initialization, Normalized Cut gains an advantage over Kernel K-Means at K=4, suggesting sensitivity to initialization methods.

In Part4:

In results of part 4, looking at the 2 tables, table for K =2 and table for K = 3, taking image 1 and image 2 into account, we quickly notice that for the same image, the visualization for normalized cut and ratio cut’s eigenspace seem very similar, but not exactly the same, some minor differences can be spotted.

That could come from the fact that their algorithms tend to be similar in execution.

It could also be due to the choice of kernel hyperparameters.

In this case, they are very difficult to tune because the code takes a long time to run every time, for the spectral clustering algorithms.

  ii. Compare the execution time of different settings.

I managed to capture some logs of the execution time with time.time().

Talking about the 2 spectral clustering algorithms first, I note that it take about 6 minutes to compute the Laplacian matrix or normalized Laplacian + the eigenvectors matrix. Most of that time is definitely attributed to the eigenvectors matrix, taking almost all **the 6 minutes** to run.  
With the efficient implementation for the inverse of the Degree matrix, the Laplacian matrix takes only an instant to run.

Thus, as mentioned in code 0019 and 0020 for the spectral clustering algorithms, I saved the eigenvectors matrix so that it can be loaded everytime instead of waiting over each loop, and thus the code can finish 1 run over one image in about 5 seconds or less.

Below lets look at how the kernel matrix is faster if we suppose that the spectral clustering algorithms need to always compute their eigenvectors matrix, but also at how slower it is if the spectral clustering algorithms only need to load the eigenvectors matrix from memory.

Overall, the execution time is not constant and can vary by over 20 seconds at time depending on the total run time.

Here is the value for kernel k-means running at all K values of 2 to 4 and both modes for both images below. Image 2 in general tends to take a little more time to run.

Also, the runtime seems to often be shortened a lot during the mode 1, k-means++ initialization. That is a clear indication that the initialization has great impact on the run time.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Mode 0

Clusters = 2

Image 1

Total Execution Time for Image: 61 seconds

Image 2

Total Execution Time for Image: 66 seconds

Clusters = 3

Image 1

Total Execution Time for Image: 66 seconds

Image 2

Total Execution Time for Image: 76 seconds

Clusters = 4

Image 1

Total Execution Time for Image: 74 seconds

Image 2

Total Execution Time for Image: 74 seconds

Mode 1

Clusters = 2

Image 1

Total Execution Time for Image: 33 seconds

Image 2

Total Execution Time for Image: 42 seconds

Clusters = 3

Image 1

Total Execution Time for Image: 78 seconds

Image 2

Total Execution Time for Image: 56 seconds

Clusters = 4

Image 1

Total Execution Time for Image: 68 seconds

Image 2

Total Execution Time for Image: 81 seconds

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

    iii. Anything you want to discuss.

Spectral Clustering Runtime: Spectral Clustering, particularly the computation of eigenvectors, takes around 6 minutes. Efficiently saving and loading eigenvectors significantly reduces runtime to approximately 5 seconds per run.

Kernel K-Means Time Dynamics: Kernel K-Means runtime varies, generally taking less time during K-means++ initialization. The runtime is influenced by K values and image differences.

General Insights:

Cluster Visualization: Grouping results reveal clarity in visualizing land-sea junctions.

Algorithm Performance: Iterative convergence of K-means may not always lead to improved results.

Spectral Clustering Factors: Optimal K values vary with image characteristics. K>4 may result in sparse clusters. Initial clustering methods impact results, but the gap between random and K-means++ diminishes with iterations.

Overall Reflection:

Spectral Clustering Trade-offs: While Spectral Clustering has fewer iterations, eigenvector computations contribute to longer runtimes. Optimal K values depend on image complexity.

Initialization Impact: Initial methods significantly influence early results, but convergence narrows the gap.

Eigenspace Coordinates: Contrary to the misconception, data points within the same cluster in the eigenspace of the graph Laplacian tend to have similar, not identical, coordinates.

Concluding Thoughts:

Balancing Act: The choice between Kernel K-Means and Spectral Clustering depends on the desired balance between performance and runtime efficiency, with careful consideration of initialization methods.