Clustering-based Image Segmentation Techniques

March 14, 2023

[1]:
```
# Import the essential libraries
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import cv2
import os

from skimage import io
from sklearn.cluster import MeanShift, estimate_bandwidth
from sklearn.cluster import KMeans

from mpl_toolkits.mplot3d import Axes3D

from skimage import data, segmentation, color
from skimage.future import graph
from skimage.color import rgb2gray

from mpl_toolkits.mplot3d import Axes3D

import warnings
warnings.filterwarnings("ignore")
```

This is a Python script that begins by importing various libraries that will be used in the script. Each import statement pulls in a specific library that the script needs to access to perform certain functions.

The libraries being imported in this script are:

matplotlib.pyplot: a library used for creating graphs and visualizations.

pandas: a library used for data manipulation and analysis.

numpy: a library used for numerical operations.

cv2: a library used for computer vision and image processing.

os: a library used for interacting with the operating system and its files.

In addition to these libraries, the script also imports specific modules from other libraries:

skimage.io: a module from the scikit-image library used for reading and writing image files.

sklearn.cluster: a module from the scikit-learn library used for performing clustering operations on data.

mpl_toolkits.mplot3d: a module from the matplotlib library used for creating 3D visualizations.

skimage.data: a module from the scikit-image library used for accessing sample data.

skimage.segmentation: a module from the scikit-image library used for segmenting images into regions.

skimage.color: a module from the scikit-image library used for color space conversion.

skimage.future.graph: a module from the scikit-image library used for performing graph-based operations on images.

warnings: a module used for handling warning messages in the script Finally, the last line of the script disables any warnings that may be raised during execution, so they will not be printed to the console.

[ ]:
```
def display_image(image):
    """
    params:
    image: Any input image.

    TO DO:
    plot the image
    """
    plt.axis('off')
    io.imshow(image);
```

The code is a function that takes an input image as a parameter and displays it using matplotlib's imshow function, with axis off. The imshow function is part of the scikit-image library (skimage.io), which is imported at the beginning of the code.

The function simply displays the image, without performing any modifications to it.

The plt.axis('off') line ensures that the axis are not displayed around the image. The function's docstring states that its purpose is to plot the input image, indicating that it is designed to be a simple helper function for displaying images.

[ ]:
```
def display_segmented_image(input_image, resultant_image,␣
 ↪clustering_method_and_image_type):
    """
    params:
    input_image: to display the original image.
    resultant_image: input_image after performing some operation.
    image_file_name: name of the final image that is to be saved.
    clustering method: clustering method used.

    TO DO:
    Display and save input image and resultant image.
    """
```

```
rows = 1
columns = 2

fig = plt.figure(figsize=(8, 5))

fig.add_subplot(rows, columns, 1)
plt.axis('off')
io.imshow(input_image);

fig.add_subplot(rows, columns, 2)
plt.axis('off')
io.imshow(resultant_image);

plt.title(f'{clustering_method_and_image_type}.jpg')
```

This is a Python function that takes several parameters and displays two images side by side using matplotlib's imshow function. The function also sets a title for the displayed image.

The parameters are:

input_image: the original image to be displayed on the left side of the plot.

resultant_image: the modified image to be displayed on the right side of the plot.

clustering_method_and_image_type: a string that is used to create the title of the plot. It indicates the clustering method used and the type of image being displayed.

The function begins by defining the number of rows and columns to use for the plot (1 row and 2 columns), and creating a new figure using plt.figure().

It then adds two subplots to the figure, one for the input image and one for the resultant image, using fig.add_subplot(rows, columns, index). The index parameter specifies the position of the subplot within the grid, which in this case is the first (1) or second (2) column of the first (and only) row.

Each subplot has its axis turned off using plt.axis('off'), and the input and resultant images are displayed in the respective subplots using io.imshow().

Finally, the function sets the title of the plot using plt.title(). The title is constructed using an f-string that includes the clustering_method_and_image_type parameter, followed by the file extension ".jpg".

Overall, this function is designed to display and save two images side by side, and to provide a meaningful title that includes information about the clustering method and image type used.

[ ]:
```
def colors_from_centers(centers):
    """
    params:
    centers: centers from clustering method.

    TO DO:
    get the colors from center values
```

```
    return colors
    """
    colors = []
    for each_color in centers:
        colors.append(each_color)

    return colors
```

This is a Python function that takes an input parameter centers, which is assumed to be the centers obtained from a clustering method, and returns a list of colors extracted from those centers.

The function begins by initializing an empty list called colors, which will hold the resulting color values.

Next, the function loops through each center value in the centers parameter, and appends it to the colors list.

Finally, the function returns the colors list.

Overall, this function is a simple helper function that extracts and returns the colors corresponding to the centers obtained from a clustering method. It assumes that the centers input parameter is a list of color values in a specific color space (e.g. RGB or LAB).

```
[ ]: def perform_kmeans(flat_image, num_clusters):
    """
    params:
    Input: fallted image
    num_clusters: number of cluster

    TO DO:
    Perform kmeans clustering.

    return:
    centers and labels
    """
    km = KMeans(n_clusters = num_clusters)
    km.fit(flat_image)
    centers = km.cluster_centers_
    labels = km.labels_
    return centers, labels
```

This is a Python function that takes two input parameters: flat_image and num_clusters. It performs k-means clustering on the flattened image data and returns the resulting cluster centers and labels.

The function begins by creating a new KMeans object with num_clusters as the number of clusters, using km = KMeans(n_clusters = num_clusters).

Next, the km object is fit to the flattened image data using km.fit(flat_image), which performs k-means clustering and finds the optimal cluster centers.

The resulting cluster centers are extracted using km.cluster_centers_, and the corresponding labels for each pixel are obtained using km.labels_.

Finally, the function returns both the centers and labels.

Overall, this function is a simple helper function that performs k-means clustering on a flattened image and returns the resulting centers and labels. It assumes that the flat_image input parameter is a flattened version of the original image, and that num_clusters is the desired number of clusters for k-means clustering.

[ ]:
```python
def perform_segmentation(input_image, colors, labels):
    """
    params:
    input_image: original image
    colors: colors from centers that are calculated from clustering
    labels: labels from clustering.

    TO DO:
    Create a segmented image using colors and labels

    return:
    segmented image.
    """
    segmented_image = np.zeros((input_image.shape[0] * input_image.shape[1],␣
  ↪input_image.shape[2]),dtype='uint8')
    for i in range(segmented_image.shape[0]):
      segmented_image[i] = colors[labels[i]]

    segmented_image = segmented_image.reshape((input_image.shape))
    return segmented_image
```

This is a Python function that takes three input parameters: input_image, colors, and labels. It creates a segmented image using the input image, the color values corresponding to each cluster center, and the corresponding labels for each pixel.

The function begins by creating a new numpy array called segmented_image with the same number of rows and columns as the input image, but with a single channel for storing the cluster labels.

Next, the function loops through each pixel in segmented_image and sets its value to the color value corresponding to its label. This is done by setting segmented_image[i] to colors[labels[i]] for each index i.

Finally, the function reshapes segmented_image to the same shape as the input image and returns it.

Overall, this function is a simple helper function that creates a segmented image from an input image, the corresponding cluster center colors, and the cluster labels for each pixel. It assumes that input_image is an RGB image and that colors is a list of color values corresponding to the centers of a clustering method, and that labels is an array of labels assigned to each pixel by the clustering method.

[ ]:
```python
# import and display the original image
flower1 =  io.imread('flower1.jpg')
display_image(flower1)
```

This code loads an image file named "flower1.jpg" using the io.imread function from the skimage module, and assigns the resulting image object to the variable flower1.

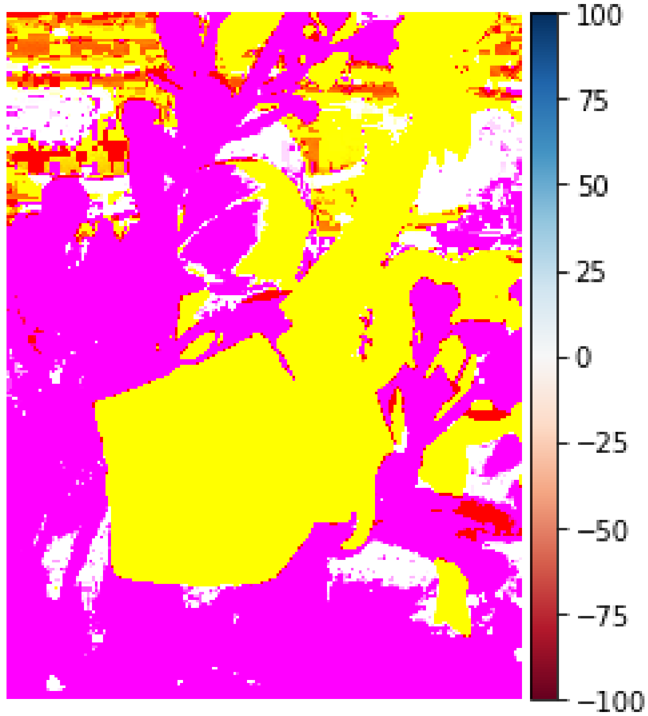
Next, the display_image function is called with flower1 as its input parameter, which displays the image using matplotlib and the skimage io.imshow function, with the axis turned off.

Overall, this code imports and displays an image file named "flower1.jpg".

[ ]:
```python
####################################
# input image in lab format
####################################

flower1_lab = color.rgb2lab(flower1)
display_image(flower1_lab)
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
```



This code first converts the RGB image flower1 to the CIELAB color space using the color.rgb2lab function from the skimage module. The resulting image object in the CIELAB color space is then assigned to the variable flower1_lab.

Next, the display_image function is called with flower1_lab as its input parameter, which displays the image using matplotlib and the skimage io.imshow function, with the axis turned off.

Overall, this code converts the RGB image flower1 to the CIELAB color space using the color.rgb2lab function and displays the resulting image.

K-Means

[ ]:
```
def kmeans_segmentation(image):
    """
    params:
    image: original image

    TO DO:
    1. flatten the image.
    2. perform k-means clustering and get the centers and labels.
    3. get the colors from centers.
    4. get the segmented image using colors and labels.

    return: segmented image
    """
```

```python
    flat_image  = image.reshape((-1,3))

    ###############################################################
    # For K-means clustering, use K = 5.
    ###############################################################

    kmeans_centers, kmeans_labels = perform_kmeans(flat_image, 5)
    kmeans_colors = colors_from_centers(kmeans_centers)
    kmeans_segmented_image = perform_segmentation(image, kmeans_colors,␣
↪kmeans_labels)
    return kmeans_segmented_image
```

This code defines a function named kmeans_segmentation that takes an input image as a parameter. The function performs the following steps to obtain a segmented image using k-means clustering:

The input image is flattened into a 2D array of shape (number of pixels, 3) using the reshape method of the NumPy array.

K-means clustering is performed on the flattened image using the perform_kmeans function, with the number of clusters set to 5. The resulting cluster centers are converted to colors using the colors_from_centers function.

The perform_segmentation function is called with the input image, the colors, and the cluster labels to obtain the segmented image. Finally, the function returns the resulting segmented image.

Overall, this function performs k-means clustering on the input image with 5 clusters and returns the resulting segmented image.

[ ]:
```python
# Segmentation using k-means
kmeans_segmented_rgb_image = kmeans_segmentation(flower1)
display_segmented_image(flower1, kmeans_segmented_rgb_image, 'KM_RGB')
```

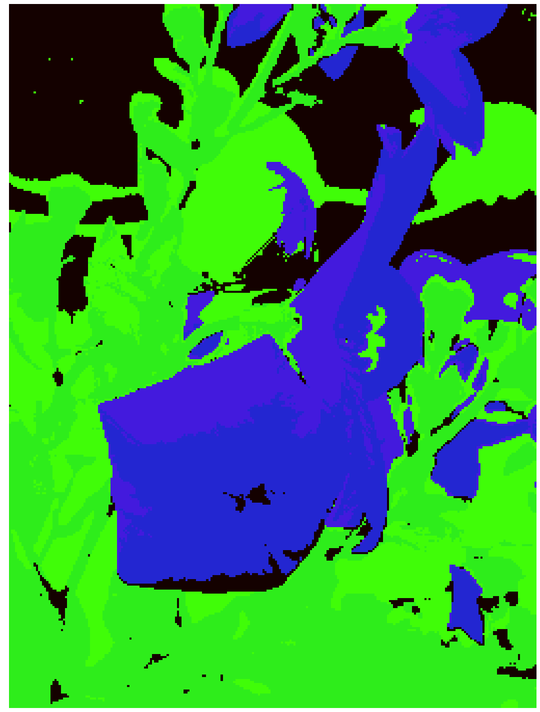
This code performs segmentation on the input image flower1 using k-means clustering. The kmeans_segmentation function is called with the input image as a parameter, and the resulting segmented RGB image is stored in the variable kmeans_segmented_rgb_image.

Then, the display_segmented_image function is called with the original input image, the segmented image, and a string 'KM_RGB' indicating that k-means clustering was used to obtain the segmentation result.

This function displays the original image and the segmented image side by side in a figure and saves the resulting figure with the specified name.

In summary, this code applies k-means clustering to the input image flower1, obtains the segmented RGB image, and displays both the original and the segmented images side by side using the display_segmented_image function.

[ ]:
```
# segmentation using k-means
kmeans_segmented_lab_image = kmeans_segmentation(flower1_lab)
display_segmented_image(flower1, kmeans_segmented_lab_image, 'KM_lab')
```



This code is performing segmentation on the input image 'flower1_lab' using k-means clustering algorithm.

The 'kmeans_segmentation' function is called on the input image which returns a segmented image. The returned segmented image is then passed to the 'display_segmented_image' function along with the original input image and a string 'KM_lab' as arguments.

The 'display_segmented_image' function displays the input image and the segmented image side by side using the 'imshow' function of the 'io' module. It also sets the axis to off to remove the axis labels. Finally, it sets the title of the plot using the provided string argument.

So, this code is displaying the original input image and the segmented image using k-means clustering in the LAB color space, side by side with the title 'KM_lab'.

Mode-Seeking

[ ]:
```
def perform_mean_shift(flat_image):
    """
    params:
    flat_image: flatten image of the input image

    TO DO:
    perform mean shift algorithm and get the centers and labels.

    return:
```

```
    centers and labels.
    """
    flat_image = np.float32(flat_image)
    bandwidth = estimate_bandwidth(flat_image, quantile=.06, n_samples=3000)

   ␣
 ↪##########################################################################################
    # For (ii), use either the MeanShift class in sklearn.cluster or quickshift␣
 ↪function in
    # skimage.segmentation module.
   ␣
 ↪##########################################################################################
    mean_shift = MeanShift(bandwidth = bandwidth, max_iter=50, bin_seeding=True)
    mean_shift.fit(flat_image)
    centers = mean_shift.cluster_centers_
    labels = mean_shift.labels_

    return centers, labels
```

This code defines a function perform_mean_shift that takes a flattened image as input and performs mean shift clustering algorithm to get the centers and labels.

First, the input flattened image is converted to float32 data type. The bandwidth is estimated using the estimate_bandwidth function from sklearn.cluster module. The bandwidth parameter controls the smoothness of the resulting clusters, with higher values resulting in fewer and smoother clusters.

Then, MeanShift class from sklearn.cluster module is used to perform the clustering. The bandwidth and max_iter parameters are set to the estimated bandwidth and 50, respectively. The bin_seeding parameter is set to True, which initializes cluster centers with the bin centers of a uniform grid. Finally, the fit method is used to perform clustering and obtain the centers and labels.

The function returns the centers and labels obtained from the mean shift algorithm.

[ ]:
```
def mean_shift_segmentation(image):
    """
    params:
    image: original image

    TO DO:
    1. flatten the image.
    2. perform mean-shift  and get the centers and labels.
    3. get the colors from centers.
    4. get the segmented image using colors and mean-shift labels.

    return: segmented image
    """
    flat_image  = image.reshape((-1,3))
```

```
    mean_shift_centers, mean_shift_labels = perform_mean_shift(flat_image)
    mean_shift_colors = colors_from_centers(mean_shift_centers)
    mean_shift_segmented_image = perform_segmentation(image, mean_shift_colors,
 ↪mean_shift_labels)

    return mean_shift_segmented_image
```

This code defines a function mean_shift_segmentation which performs segmentation of an input image using mean-shift clustering algorithm.

The function takes an input image as a parameter and performs the following steps:

Reshape the input image into a flattened array of pixel values. Perform mean-shift clustering on the flattened image array and obtain the cluster centers and labels.

Obtain the colors from the cluster centers using the colors_from_centers function defined earlier.

Generate a segmented image using the colors and mean-shift labels using the perform_segmentation function defined earlier.

The function returns the segmented image.

Overall, the mean_shift_segmentation function performs image segmentation using mean-shift clustering algorithm by grouping similar pixels together based on their color and intensity values.

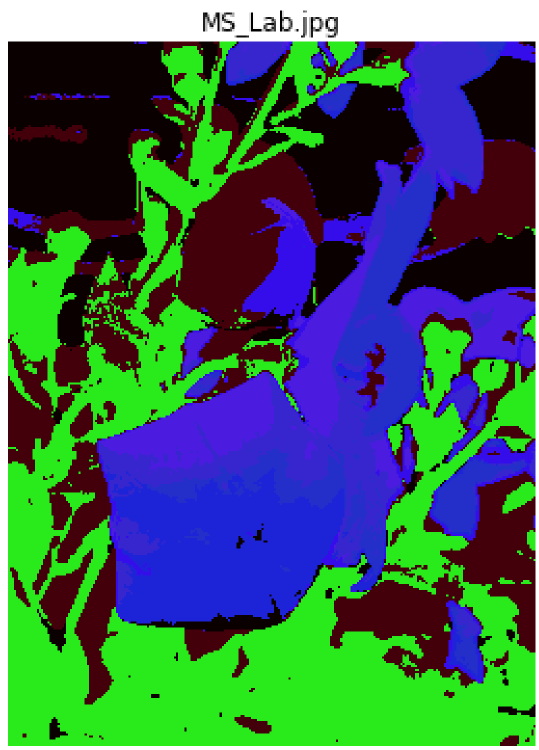
[ ]:
```
# segmentation using mode-seeking
mean_shift_segmented_rgb_image = mean_shift_segmentation(flower1)
display_segmented_image(flower1, mean_shift_segmented_rgb_image, 'MS_RGB')
```

This code performs segmentation on the original image flower1 using mode-seeking with mean shift algorithm.

It calls the mean_shift_segmentation() function with flower1 as the input and assigns the returned segmented image to the mean_shift_segmented_rgb_image variable.

Finally, it displays the original image and its corresponding segmented image using the display_segmented_image() function with flower1, mean_shift_segmented_rgb_image, and 'MS_RGB' as the input parameters.

[ ]:
```
# segmentation using mode-seeking
mean_shift_segmented_lab_image = mean_shift_segmentation(flower1_lab)
display_segmented_image(flower1, mean_shift_segmented_lab_image, 'MS_Lab')
```



This code performs mode-seeking segmentation on the input image "flower1_lab" and displays the resulting segmented image with a label "MS_Lab".

Mode-seeking segmentation is performed by the function "mean_shift_segmentation". The input to this function is the original image in the lab format, "flower1_lab".

The function first flattens the image and then applies the mean-shift algorithm to get the centers and labels for segmentation.

The colors are extracted from the centers and then a segmented image is obtained using the colors and the mean-shift labels by calling the "perform_segmentation" function.

The resulting segmented image is then displayed using the "display_segmented_image" function with the original image "flower1" and the label "MS_Lab".

Normalized Cut

```
[ ]: def normalized_cut(image, convert_2_lab):
     ␣
     ↪#################################################################################
      # Some of the existing functions for image
      # segmentation have "convert2lab" parameters. If you can't convert the color␣
     ↪space using
      # parameter setting, you can use rgb2lab function in skimage package to␣
     ↪convert an RGB image
      # to Lab color space.
     ␣
     ↪#################################################################################

      # In particular, functions in segmentation and graph
      # modules of scikit-image package and/or cluster module of scikit-learn␣
     ↪package would be useful.
      labels = segmentation.slic(image, compactness=30, n_segments=100,␣
     ↪start_label=1, convert2lab = convert_2_lab)

      g = graph.rag_mean_color(image, labels, mode='similarity')
      labels2 = graph.cut_normalized(labels, g)
      normalized_cut_image = color.label2rgb(labels2, image, kind='avg', bg_label=0)

      return normalized_cut_image.astype(np.uint8)
```

The code defines a function named normalized_cut that performs normalized cut segmentation on an input image.

The function takes two parameters:

image: the original image to be segmented convert_2_lab: a boolean value indicating whether to convert the image to the LAB color space before segmentation. If True, the image is converted to LAB using the rgb2lab function from the skimage package.

The function performs the following steps:

Segment the image using the SLIC algorithm with compactness=30, n_segments=100, start_label=1, and convert2lab set to the value of convert_2_lab. This produces a label image with each pixel assigned to a superpixel.

Compute a region adjacency graph (RAG) using the mean color of each superpixel as the node weight, and the similarity between neighboring superpixels as the edge weight.

Use the normalized cut algorithm to partition the RAG into disjoint regions, and assign each superpixel to one of these regions.

Generate a segmented image by replacing each superpixel with its assigned region label, and colorizing the result using the average color of each region.

The function returns the resulting segmented image, converted to the uint8 data type.

[ ]:
```
# Segmentation using Ncut
normalized_cut_rgb_image = normalized_cut(flower1, False)
display_segmented_image(flower1, normalized_cut_rgb_image, 'Ncut_RGB')
```

This code performs image segmentation using Normalized Cut (Ncut) algorithm.

The function normalized_cut takes two inputs, an image and a boolean variable convert_2_lab. If convert_2_lab is True, the function converts the input image from RGB to Lab color space using the convert2lab parameter of the segmentation.slic function. If convert_2_lab is False, the function assumes that the input image is already in Lab color space.

The function then performs image segmentation using the SLIC (Simple Linear Iterative Clustering) algorithm from the segmentation module of the scikit-image package. The compactness parameter controls the balance between color proximity and spatial proximity. The n_segments parameter specifies the desired number of segments. The start_label parameter controls the label value for the first segment.

Next, the function computes the RAG (Region Adjacency Graph) for the segmented image using the graph.rag_mean_color function from the graph module of the scikit-image package. The mode parameter specifies the type of similarity metric to use.

Finally, the function computes the normalized cut of the RAG using the graph.cut_normalized function from the graph module of the scikit-image package. The resulting labels are then converted to an RGB image using the color.label2rgb function from the color module of the scikit-image package.

The resulting normalized cut segmented image in RGB format is stored in the variable normalized_cut_rgb_image, which is then displayed using the display_segmented_image function with the label 'Ncut_RGB'.

[ ]:
```
# Segmentation using Ncut
normalized_cut_lab_image = normalized_cut(flower1, True)
display_segmented_image(flower1, normalized_cut_lab_image, 'Ncut_Lab')
```

Ncut_Lab.jpg

This code performs image segmentation using the Normalized Cut (Ncut) algorithm.

The function normalized_cut takes two arguments: image, which is the original image to be segmented, and convert_2_lab, a boolean variable that specifies whether to convert the image to the Lab color space or not.

The function first uses the slic function from the segmentation module of the scikit-image package to generate an initial segmentation of the image.

This function partitions the image into a number of superpixels (regions of pixels with similar colors) based on their color and spatial proximity. The compactness parameter controls the balance between color and spatial proximity, while the n_segments parameter specifies the number of superpixels.

The convert2lab parameter is used to specify whether to convert the image to the Lab color space before performing the segmentation or not.

Next, the rag_mean_color function from the graph module of the scikit-image package is used to construct a region adjacency graph (RAG) from the initial segmentation.

This graph represents the superpixels as nodes and the edges between them as the similarity between their mean color values.

The mode parameter is set to 'similarity' to use the color similarity between regions to calculate the edge weights.

The cut_normalized function from the graph module of the scikit-image package is then used to perform the normalized cut algorithm on the RAG. This algorithm cuts the graph into subgraphs by minimizing the sum of the weights of the edges that are cut.

The resulting subgraphs represent the final segmentation of the image.

Finally, the label2rgb function from the color module of the scikit-image package is used to color the segments of the image with the average color of the pixels in each segment.

The kind parameter is set to 'avg' to use the average color, and the bg_label parameter is set to 0 to specify the label of the background.

The segmented image is returned as an array of type np.uint8. The segmented image is displayed using the display_segmented_image function, along with a title indicating the segmentation method used.