Lab 2: Image Segmentations

CS4243, Semester 1, 2020

TA: Kai Xu
Due on: 01-10-2020 23:59

Objective

In this lab you will implement image segmentation with K-means and mean-shift algorithms.

Part 1: Image Prepossesing (5%)

In the lecture, we talked about two key challenges of image segmentation. The first challenge is to find suitable feature representations for the pixels. In this part, you will need to generate different features from original image. Follow steps in the jupyter notebook, implement functions below:

a) In order to make the color more uniform, you shall first smooth image using 5x5 Guassian filter with $\sigma = 5.0$, you may use `cv2.blur()` or `cv2.GaussianBlur()`;

☐ Implement following function: `smoothing()`.(3 points)

b) Here we introduce a new color space called L*a*b, L*a*b color is designed to approximate human vision. It aspires to perceptual uniformity, and its L component closely matches human perception of lightness. To convert the image from RGB to L*a*b color space. You may use functions from `skimage.color` library.

☐ Implement following function: `RGBtoLab()`.(2 points)

Part 2: Clustering (80%)

In part 2, you will implement K-means and mean-shift clustering algorithms.

2.1 K-Means Clustering

The basic idea of K-Means algorithm is to randomly initialize the k cluster centers, and iterate between assigning membership and computing cluster centers. For our setting, assume the input data are $P = \{p_1, ..., p_n\}$, for each $p_i \in R^d$. n is the number of points, d is the number of features.

Here are some steps you may follow for your implementation:

1. Randomly pick k points from P as the centers: $c_i, c_2, ..., c_k \in R^d$
2. Iterate the process until max iterations or centers are no longer change up to some threshold:
   i. Assign each point to nearest center:

$$y_i = \underset{j}{\operatorname{argmin}} ||p_i - c_j||^2$$

   y_i is the cluster id for point p_i.

   ii. Re-estimate each center as mean of points assigned to it:

$$c_j := \frac{\sum_{i=1}^{n} \mathbb{1}(y_i == j) p_i}{\sum_{i=1}^{n} \mathbb{1}(y_i == j)}$$

$\mathbb{1}(y_i == j)$ is the indicator function which equals 1 if point i belong to cluster j, 0 otherwise.

☐ Implement following function: `k_means_clustering()`.

■ Prohibited function: `cv2.kmeans()`.

2.2 Mean Shift Clustering

The main idea behind mean shift clstering is to find modes or maxima in the local density function in the feature space for each pixel. To achieve this, mean shift iteratively computes the mean of all the pixels that lies within a spherical window of certain radius and shifting the window center to the mean until convergence (until max iterations or centroids stop moving up to some threshold).

The vanilla mean shift algorithm has an issue of computational complexity, because we need to shift as many windows as how many points we have. Yet we still covered some methods to speed up mean shift in the lecture.

In this section, you will implement an accelerated mean shift, by tessellating the feature space with windows by sampling, and run the mean shift procedure on these windows in parallel. Noted that this method is sightly different from both of two speed up methods covered in the lecture. Please refer following instructions for your implementation.

The steps are:

1. Generate windows by tessellating. We also call the center of these windows 'bin seeds', which means it's the starting point for each mean shift thread. For the sake of simplicity and time complexity, we will do tessellating by following steps:
   i. For all points, compress all coordinates within square of bandwidth length into one coordinates:

      Compute

      `np.round(X/bandwidth, Y/bandwidth)`

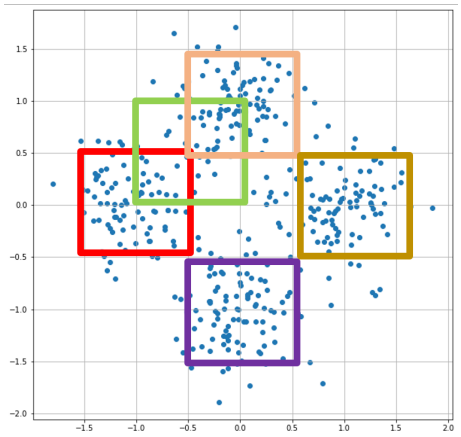
      for all the points.
   ii. Group pixels with same value, suppose the value is $u^m \in R^d$. Compute the total number N_{u^m} of points belong to this group. we also call u^m the seed of this group.
   iii. For all the seeds $u^1, u^2, ..., u^m$, check the total number of points belong to this seed, filter out all the seeds with total number less than the threshold.

      $$N_{u^m} \geq \texttt{min\_bin\_freq}$$

      `min_bin_freq` is used to drop out-liners with low density, in this lab we use `min_bin_freq=1`, means that we do not drop any points when we do tessellating.
   iv. Multiply bandwidth to reproject seeds to original size and return all the seeds as a list.

      $$u^m * \texttt{bandwidth}$$

Here's the graph illustrating these steps. Each color represents a bin seed.



☐ Implement following function: `get_bin_seeds()`.

The reason why we use squares instead of spheres is because computing sphere needs more computations. Also for the seeds, we do not use random sampling because it tends to give us more points at high density area. In case we only want a reasonable starting point set which can cover most of the space that distribution span. It's quite enough.

2. Now we can run standard mean shift algorithm for each seed. Here you need to implement single thread mean shift for one starting point.
    1. Iterate following steps until convergence to find peak.
        i. Find neighbor points with bandwidth, you may use nearest neighbors libraries from `sklearn.neighbors`.
        ii. Compute mean of all points within, shift center of window to new mean.
    2. Return peak and total number of points within bandwidth of the peak.

    ☐ Implement following function: `mean_shift_single_seed()`.

3. Complete mean shift pipeline. Run `mean_shift_single_seed` in parallel. Merge peaks within bandwidth. Assign points to the closest cluster peak.
    i. We already run parallel mean shift for you in the code. You now have a dict `center_intensity_dict` for all the peaks. The key is the tuple of peak coordinates, the value is the number of points within bandwidth of the peak.
    ii. Remove near duplicate peaks if the distance between two peaks is less than the bandwidth, remove the one with fewer points. You may use `NearestNeighbors` library.
    iii. Iterate all the points, assign points to the nearest cluster peak. You may use `NearestNeighbors` library.

    ☐ Implement following function: `mean_shift_clustering()`.

    ■ Prohibited function: `cv2.meanShift`.

Part 3: Image Segmentation (15%)

Use functions you implemented above to do image segmentation. You should first reshape image to meet input format. And to get a better result, you should tweak k for K-means and

b for mean shift. We also provide more pictures for you to test your implementation.

- ☐ Implement following function: `k_means_segmentation()`.
- ☐ Implement following function: `mean_shift_segmentation()`.

Hand in

Files to be submitted are `lab2.py` and `lab2.ipynb`. Please zip them into a file named `AXXX_AXXX_AXXX.zip`, where AXXX is the student number of the group members. Each group should submit only once. Note that we will use hold-out testing examples to test the functions in `lab2.py`. Therefore, `lab2.py` is essential and `lab2.ipynb` is for reference only in case you fail some testing cases. Groups with missing files or incorrectly formatted code that does not run will be penalized. The submission deadline is `01/10/2020 23:59`. Q&A sessions for Lab 2 will be held on `15/9/2020 15:00-17:00` & `18/9/2020 09:00-11:00`.