Assignment 6

Junyi Liao 20307110289

1 Image Segmentation

1.1 Algorithm

For segmentation, we apply the k-means algorithm to split pixels into different clusters. The clustering result of pixels corresponds to our segmentation result.

The idea of clustering can be very simple. Given centroids $\{\mu_1, \cdots, \mu_k\}$ of k clusters $\{\mathcal{C}_j\}_{j=1}^k$, defined as the mean value of elements in the cluster, then each pixel x_i will be subsumed under the cluster corresponding to the nearest centroid:

$$f(x) = \underset{\mathcal{C}_j,\ j\in\{1,\cdots,k\}}{\arg\min} d(x, \mu_j);$$

Here d is the distance metric, $d(x, \mu_j) = |I(x) - \mu_j|$, $I(\cdot)$ is the gray-scale value of the pixel. k is accessed by some priori knowledge, say k is a hyperparameter, and we need to find the k centroids $\{\mu_1, \cdots, \mu_k\}$.

The k-means clustering algorithm is a common method for cluster analysis. The algorithm initializes the centroids $\{\mu_j\}_{j=1}^k$ with some criteria, then iteratively splits training set $\{x_i\}_{i=1}^N$ to clusters $\mathcal{C}_1, \cdots, \mathcal{C}_k$ and updates centroids according to the latest clustering result. The centroids finally converge to a stable solution, and the clustering result no longer changes.

The pseudocode is shown below.

Input: image $I = \{x_1, \cdots, x_N\}$, number of clusters k;
Output: centroids $\{\mu_1, \cdots, \mu_k\}$;

K-MEANS$(x_1, \cdots, x_N, k)$

Initialize the centroids as $\{\mu_1^{(0)}, \cdots, \mu_k^{(0)}\}$
for $r = 0, 1, \cdots$
    $\mathcal{C}_1 = \cdots = \mathcal{C}_k = \varnothing$
    for $i = 1, \cdots, N$
        $j^* = \underset{j\in\{1,\cdots,k\}}{\arg\min}\, d(x_i, \mu_j^{(r)})$
        $\mathcal{C}_{j^*} = \mathcal{C}_{j^*} \cup \{x_i\}$
    for $j = 1, \cdots, k$
        $\mu_j^{(r+1)} = \sum_{x_i\in\mathcal{C}_j} \frac{I(x_i)}{|\mathcal{C}_j|}$
    if each $\mu_j^{(r+1)} == \mu_j^{(r)}$
        $\{\mu_1, \cdots, \mu_k\} = \{\mu_1^{(r)}, \cdots, \mu_k^{(r)}\}$
        return $\{\mu_1, \cdots, \mu_k\}$

In implement, naive k-means algorithm initializes centroids by randomly choosing k pixels from the input image $\{x_i\}_{i=1}^N$ in line 1. Meanwhile, due to the limited precision of floating-point arithmetic, the terminate condition in line 9 is set as

$$\max_{j\in\{1,\cdots,k\}} |\mu_j^{(r+1)} - \mu_j^{(r)}| < \varepsilon,$$

where ε is a sufficiently small constant.

1.2 Amelioration

In naive k-means clustering, the centroids don't necessarily converge to the optimal solution, and the output might be strongly correlated to initialization. Since the initialization in naive k-means algorithm is random, its performance can be very unstable even over the same image.

To neutralize the randomness of k-means algorithm, we develop an alternative initialization method. At first, we only choose one pixel μ_1 from the input image $I = \{x_i\}_{i=1}^N$. According to the inter-cluster-distance maximization criterion, the subsequent pixels are iteratively chosen as

$$\mu_{j+1} = \arg\max_{x\in\mathcal{S}} \min_{l\in\{1,\cdots,j\}} d(x, \mu_l),\ j = 1, \cdots, k-1,$$

say we choose the pixel value which has the largest distance from the nearest centroid that has been chosen.

The pseudocode is shown below.

Input: image $I = \{x_1, \cdots, x_N\}$, number of clusters k;

Output: initialized vectors $\mathcal{M} = \{\mu_1^{(0)}, \cdots, \mu_k^{(0)}\}$;

K-MEANSINIT$(x_1, \cdots, x_N, k)$

1 Generate $\mu_1^{(0)}$ with probabilities $\mathbb{P}\left(\mu_1^{(0)} = I(x_i)\right) = 1/N,\ i = 1, \cdots, N$

2 $\mathcal{M} = \{\mu_1^{(0)}\}$

3 **for** $j = 1, 2, \cdots, k-1$

4 **for** $i = 1, \cdots, N$

5 $d_{ij} = \min_{\mu_l^{(0)}\in\mathcal{M}} d(x_i, \mu_l^{(0)})$

6 $i^* = \arg\max_{i\in\{1,\cdots,N\}} d_{ij}$

7 $\mu_{j+1}^{(0)} = I(x_{i^*})$

8 $\mathcal{M} = \mathcal{M} \cup \{\mu_{j+1}^{(0)}\}$

9 **return** $\mathcal{M}$

This algorithm chooses a group of initial centroids with large inter-cluster distances, thus yields less iterations as well as more robust result.

1.3 Implementation

The code of k-means algorithm is shown below. As an image includes too many pixels, with each pixel falling in $\{0, 1, \cdots, L-1\},\ L = 2^8$, we apply the k-means algorithm based on the histogram of an image.

```
1 class Kmeans:
2     def __init__(self, img, nbins=256):
```

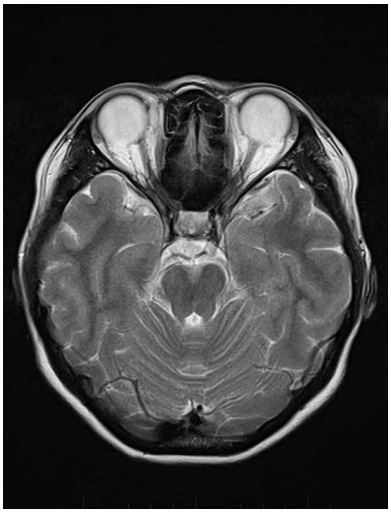
```
        self.img = img
        self.nbins = nbins
        self.hist = self.compute_hist()
        self.k = None
        self.iters = None
        self.centroid = None

    def compute_hist(self): # Compute the histogram of the image.
        hist = []
        for i in range(self.nbins):
            hist.append((self.img == i).astype(int).sum())
        return np.array(hist)

    def fit(self, k: int): # Fitting.
        if k > 1:
            self.k = k
            self.centroid, self.iters = self.clustering(k)
        else:
            raise ValueError

    def clustering(self, k: int):
        # Initialize the centroids.
        centroid = [np.random.choice(self.img.flatten()).astype(int)]
        pix = list(set(self.img.flatten()))
        for j in range(k - 1):
        min_dist = []
        for i in pix:
            min_dist.append(np.abs(i - np.array(centroid)).min())
            centroid.append(pix[np.array(min_dist).argmax()])
        centroid = np.array(centroid).astype(float)

        # Clustering.
        done = 0
        iters = 0
        while not done:
            iters += 1

            label = np.zeros((self.nbins,), dtype=int)  # The cluster label of
    a pixel with given value.
            cluster = [[] for j in range(self.k)] # Initialize clusters.
            for i in range(self.nbins):
```

```
43              # Assign each pixel value to its nearest cluster which is
    represented by the centroid.
44                  label[i] = np.abs(i - centroid).argmin()
45                  cluster[label[i]].append(i)
46              done = 1
47
48              # Update centroids.
49              for j in range(self.k):
50                  centroid_j = (self.hist[cluster[j]] * np.array(cluster[j])).sum
    ()
51                  centroid_j = float(centroid_j) / self.hist[cluster[j]].sum()
52                  if np.abs(centroid_j - centroid[j]) > 1e-6:
53                      done = 0
54                      centroid[j] = centroid_j
55
56              return np.sort(centroid), iters
57
58          def segment(self):
59          # Do segmentation, and return a matrix with each entry the label of the
     pixel.
60              segmentation = np.zeros_like(self.img)
61              for i in range(self.nbins):
62                  segmentation[self.img==i] = np.abs(i - self.centroid).argmin()
63              return segmentation
```

Then we tested the code on some examples, and the result is shown below.

1.3.1 Binary segmentation

We set $k = 2$ and ran the k-means algorithm on `brain.jpg`, the result is shown below. We also applied global Otsu algorithm, which is commonly used in thresholding, to compare with k-means.



(1) Original image brain.jpg

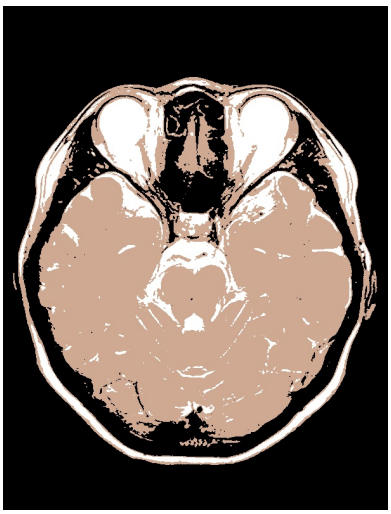
(2) Thresholded by Otsu algorithm

(3) Segmented by k-means, $k = 2$

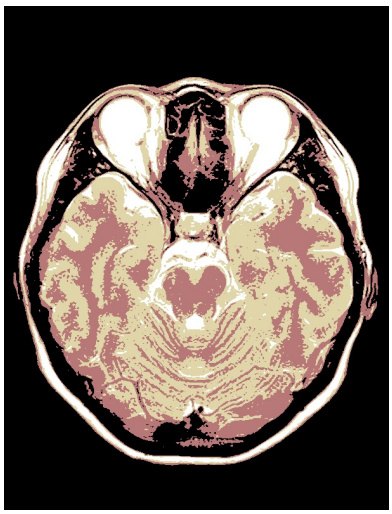
It's shown that both Otsu and k-means extract the scanned brain from the background, and they don't reveal an obvious difference. As both the algorithms tend to maximize the inter-class distance, it's reasonable they yield similar results.

1.3.2 Multiple-class Segmentation

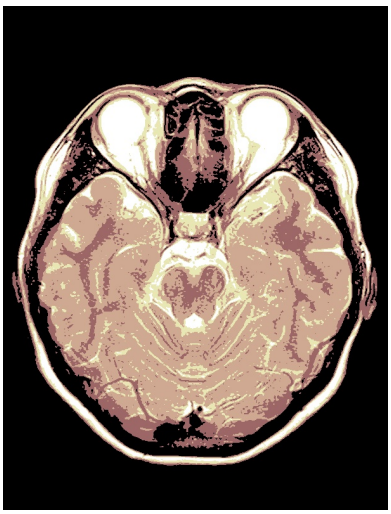
We set k as $\{3, 4, 5\}$ and ran k-means algorithm on `brain.jpg`, the result is shown below. Each class is dyed to a unique color.



(4) Segmented by k-means, $k = 3$



(5) Segmented by k-means, $k = 4$



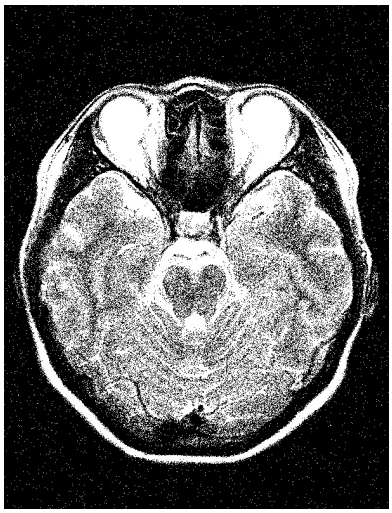
(6) Segmented by k-means, $k = 5$

The result shows that k-means algorithm tends to subsumes pixels of close gray-scale to the same class. It's reasonable since we measure the distance between pixels by the difference of their gray-scale.

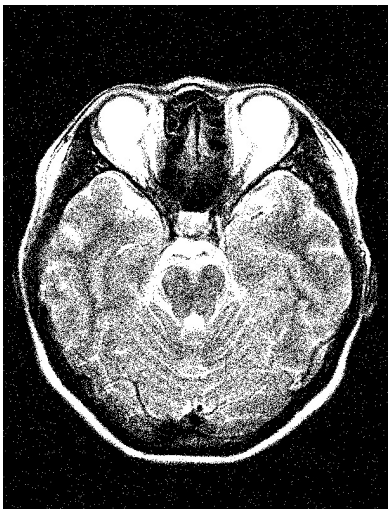
1.3.3 Performance on Image with Noise

We also tested our code on images polluted by noise, and the result is shown below. The image `brain_wn.jpg` we used is polluted by white noise.

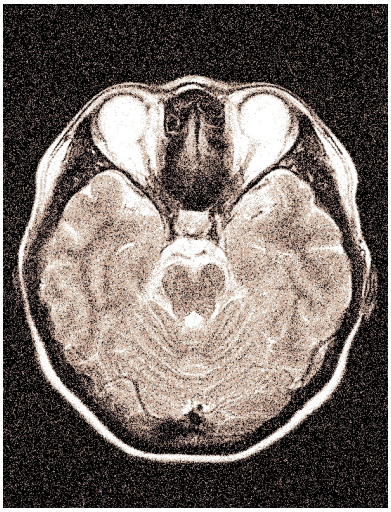
(7) Original image brain_wn.jpg



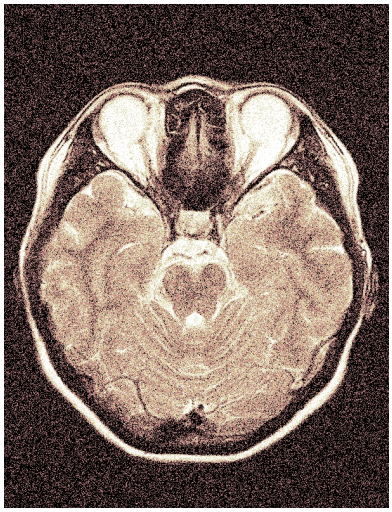
(8) Thresholded by Otsu algorithm



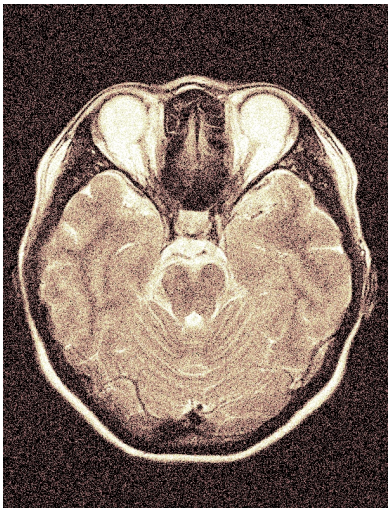
(9) Segmented by k-means, $k = 2$



(10) Segmented by k-means, $k = 3$



(11) Segmented by k-means, $k = 4$



(12) Segmented by k-means, $k = 5$

It's seen that the k-means algorithm doesn't perform well over images polluted by noise. The segmentation result is severely impacted by the noise. There are irregular dots within one segment, meanwhile pixels of different classes are mixed in mess.

1.4 Ideas of Solution

The performance of k-means algorithm on image of low quality (i.e. polluted by noise) is not satisfactory. Here we put forward some idea to improve its performance.

(1) Preprocessing or Postprocessing: Denoising and Smoothing

We know the image to be segmented is polluted by noise, so we can estimate the noise term and try to restore the original image. Considering that pixels within the same class tend to get together, we can also smooth the image to reduce the irregular fluctuation within a small region. Also, these operations can be applied after segmentation.

(2) Within processing: Spatial Constraint

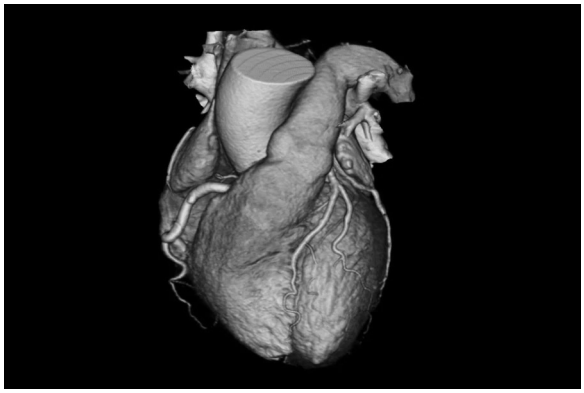
In our k-means algorithm, the distance metric we applied only takes the grayscale of pixels (say the color) into consideration, however the spatial position of pixels is omitted. Thus the segmentation result is vulnerable to noise. To improve the performance of algorithm, we can add some spatial constraint to the optimization problem. In implement, we can add a term to the distance meric in our algorithm:

$$d(x_1, x_2) = |I(x_1) - I(x_2)| + \lambda \|x_1 - x_2\|_2,$$

where x_1, x_2 is the coordinates of pixels and $I(\cdot)$ is the corresponding gray-scale value. The hyperparameter λ adjusts the proportions of spatial distance and gray-scale distance.

(3) Priori Knowledge In some cases we have some priori knowledge about the segmentation result. The priori knowledge can be applied to design a heuristic method. E.g. if we have an atlas of the image, we can add to the distance metric a term which measures the distance between pixels in the atlas.

1.5 Additional Result

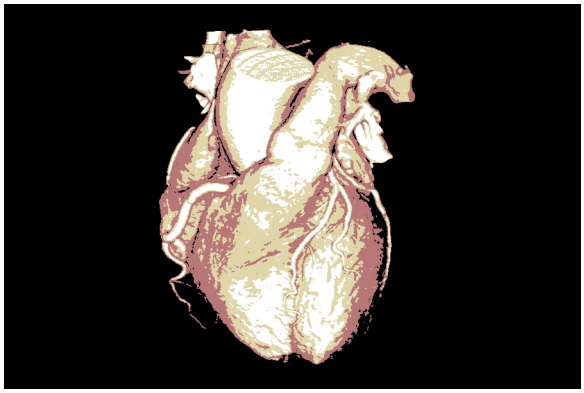


(13) Original image heart.jpg

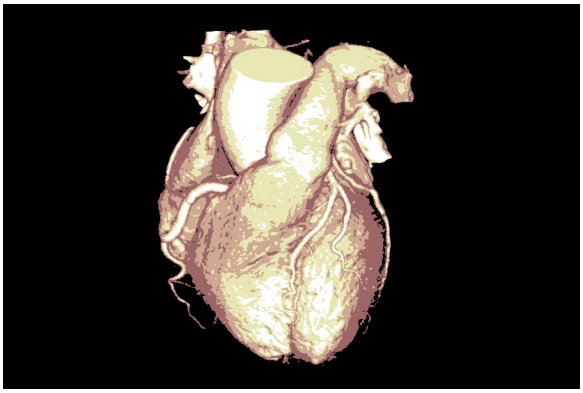
(14) Thresholded by Otsu algorithm

(15) Segmented by k-means, $k = 2$

(16) Segmented by k-means, $k = 3$



(17) Segmented by k-means, $k = 4$



(18) Segmented by k-means, $k = 5$

2 Morphology

2.1 Operations

In image morphology, let E be the entire space, then some operations are defined on structure A with a given structure B:

(I) Dilation The structure A dilated by structure B is given by

$$A \oplus B = \{z \in E \,|\, B_z \cap A \neq \varnothing\},$$

where B_z is structure B centered at position z, say $B_z = \{b + z \,|\, b \in B\}$, $\forall\, z \in E$.

Suppose I is a thresholded image with all pixels 0 or 1, and the structure A is the set of pixels with value 1, say $A = \{(x, y) \,|\, I(x, y) = 1\}$. In implement, we can traverse all the pixels in I with sturcture B, and judge if A and B are intersected. The image with dilated structure is given by

$$I_{A \oplus B}(x, y) = \bigvee_{(s,t) \in B^s_{xy}} I(s, t)$$

and we use `np.any` to implement this.
In our experiment, the structure B is predefined as a circle with radius r and center (x, y). To deal with the boundary, we preprocess the pending image by zero-padding.

```
def circle(r):
    # Construct a circle with radius r.
    circ = np.zeros((2 * r + 1, 2 * r + 1))
    for i in range(2 * r + 1):
        for j in range(2 * r + 1):
            if (i - r) ** 2 + (j - r) ** 2 <= r ** 2:
                circ[i, j] = 1
    return circ.astype(bool)

def pad(img, r):
    # Zero-padding.
    h, w = img.shape
    img_pad = np.zeros((h + 2 * r, w + 2 * r))
    img_pad[r: h + r, r: w + r] = img
    return img_pad
```

The code for dilation is shown below.

```
def dilation(img, r):
    # Dilation the object in the image.
    circ = circle(r)
    img_pad = pad(img, r)
```

```
    dil = np.zeros_like(img)

    # Traverse all pixels.
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            # Detect intersection.
            block = img_pad[i: i + 2 * r + 1, j: j + 2 * r + 1]
            dil[i, j] = np.any(block[circ])
    return dil.astype(int)
```

(II) Erosion The structure A eroded by structure B is given by

$$A \ominus B = \{z \in E \,|\, (B^s)_z \subseteq A\},$$

where B^s is symmetric structure of B, say $B^s = \{-b \,|\, b \in B\}$.

Suppose I is a thresholded image with all pixels 0 or 1, and the structure A is the set of pixels with value 1, say $A = \{(x, y) \,|\, I(x, y) = 1\}$. In implement, we can traverse all the pixels in I with sturcture B, and judge if B is included in A. The image with dilated structure is given by

$$I_{A \ominus B}(x, y) = \bigwedge_{(s,t) \in B^s_{xy}} I(s, t),$$

and we use `np.all` to implement this. The code for erosion is shown below.

```
def erosion(img, r):
    # Erosion the object in the image.
    circ = circle(r)
    img_pad = pad(img, r)
    ero = np.zeros_like(img)

    # Traverse all pixels.
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            # Detect inclusion.
            block = img_pad[i: i + 2 * r + 1, j: j + 2 * r + 1]
            ero[i, j] = np.all(block[circ])
    return ero.astype(int)
```

(III) Opening The structure A opened by structure B is given by

$$A \circ B = \{A \ominus B\} \oplus B.$$

The code is shown below.

```
def imgopen(img, r):
    ero = erosion(img, r)
    dil = dilation(ero, r)
    return dil
```

The opening operation breaks narrow junctions between objects and erases small dots over the background.

(IV) Closing The structure A closed by structure B is given by

$$A \bullet B = \{A \oplus B\} \ominus B.$$

The code is shown below.

```
def imgclose(img, r):
    dil = dilation(img, r)
    ero = erosion(dil, r)
    return ero
```

The closing operation fills up narrow gaps between objects and small holes within objects.

2.2 Experiment

Now we apply the morphological operations to restore the following images. Our goal is to remove the dot and stripe noises.

Figure 1: Original image zmic_fdu_noise.jpg

We first applied a closing operation to remove the white dots and stripe within the characters. We set the radius of circle as 3, the result is shown below.

```
img_path = 'images/zmic_fdu_noise.jpg',
img = cv2.imread(img_path, 0)
# Binarize.
img_thr = (1 - img / 255).astype(int)
img_re = imgclose(img_thr, r=3)
```

(1) Dilated by circle with $r = 3$

(2) Eroded by circle with $r = 3$

Then we applied an opening operation to remove the black dots and stripe over the background. We set the radius of circle as 2, the code is shown below.

```
img_re = imgopen(img_re, r=2)
# De-binarize.
img_re = 255 * (1 - img_re)
```

(3) Eroded by circle with $r = 2$

(4) Dilated by circle with $r = 2$

We show the images before and after operations below for contrast.

(5) Original image

(6) Restored image

The result shows the effectiveness of our method based on morphological operations, as the dot and stripe noises are removed, meanwhile the original characters remain almost intact. Because the structure we use for opening and closing operations is a circle, some sharp angles of characters are smoothed, but the overall shape is not damaged.

3 Appendix

3.1 Libraries

We use the `numpy` library to support the large-scale computation of matrices which store the grey level information of images.
We use `opencv` library (`cv2`) to read the input images and save the output images.
We use `matplotlib` to show images before and after processing.
We also use `argparse` to read in the arguments for processing, which can be specified in the command line.

3.2 IDE

All our experiments are done on PyCharm Community Edition 2022.1.2.