SCHOOL OF DATA SCIENCE, UNIVERSITY

Image Processing HW5

李裕硕

May 16, 2022

1 使用 K-MEANS 聚类进行图像分割

code:

```
def k_means(image,k=2,iters=1000):
    m, n, z = image.shape
    new_image = np.reshape(image,(m*n,z))  # 排列成m*n行z列
    cluster = new_image[random.sample(range(m*n), k), :]  # 随机选三个聚类中心
    iter = 0
    tol = 1e-11  # 容忍度
    J_prev = float('inf')
    J = []
    while True:
        iter = iter + 1
        dist = np.dot(np.sum(new_image**2,axis=1).reshape((m*n,1)),np.ones((1,k))) + \
            np.dot(np.sum(cluster**2,axis=1).reshape((k,1)), np.ones((1, m*n))).T - 2*np.dot(new_image, cluster.T)
        label = np.argmin(dist,axis=1)
        for i in range(k):
            cluster[i,:] = np.mean(new_image[label==i,:], axis=0)  # 取新的k个聚类中心
        J_cur = np.sum((new_image - cluster[label,:])**2)
        J.append(J_cur)
        print('iteration:{0},object fucntion:{1}'.format(iter,J_cur))
        if np.abs(J_cur-J_prev) < tol:
            break
        if iter == 1000:
            break
        J_prev = J_cur
    return cluster[label,:].reshape((m,n,z))
```

Figure 1.1: k-means algorithm

1.1 二类分割

code:

```python
# 计算直方图
def nD_histogram(data, dimension, nbins, pInMin, pInMax):
    pHistogram = []  # store histogram points
    pHsize = 1
    for idim in range(dimension):
        pHsize *= nbins[idim]
    for i in range(pHsize):
        pHistogram.append(0)
    pBinSpacings = []   # store bin width
    pBinPos = []   # store bin position
    for i in range(dimension):
        pBinSpacings.append(0)
        pBinPos.append(0)
    for idim in range(dimension): #store bin width of different dimensions
        pBinSpacings[idim] = (pInMax[idim] - pInMin[idim])/nbins[idim]
    for idata in range(len(data)):
        for idim in range(dimension):
            value = data[idata][idim]
            pBinPos[idim] = int((value - pInMin[idim])/pBinSpacings[idim])
            #防止越界
            pBinPos[idim] = max(pBinPos[idim], 0)
            pBinPos[idim] = min(pBinPos[idim], nbins[idim] - 1)
        index = pBinPos[0]
        for idim in range(1, dimension):
            vSize = 1
            for i in range(idim):
                vSize *= nbins[i]
            index += pBinPos[idim] * vSize
        pHistogram[index] += 1
    return np.array(pHistogram)
```

Figure 1.2: n-dimension histogram

```python
def OTSU(imhist):
    L = imhist.shape[0]
    N = np.sum(imhist)
    standn = np.arange(L)
    mu_g = np.sum(standn*imhist)/N   #图片的全局像素平均值
    w_1 = 0
    m_1 = 0
    max_y = -float('inf')
    index = 0
    for i in range(L):  #组内至少有一个元素
        w_1 += imhist[i]/N   # Frequency
        m_1 += i*imhist[i]/N
        if w_1 == 0:
            y = 0
        elif w_1 == 1:
            break
        else:
            y = (mu_g * w_1 - m_1) ** 2 / (w_1 * (1 - w_1))
        if y > max_y:
            max_y = y
            index = i
    return index
```

Figure 1.3: OTSU algorithm

首先使用 OTSU 算法对图片进行二值化阈值处理，结果如下：

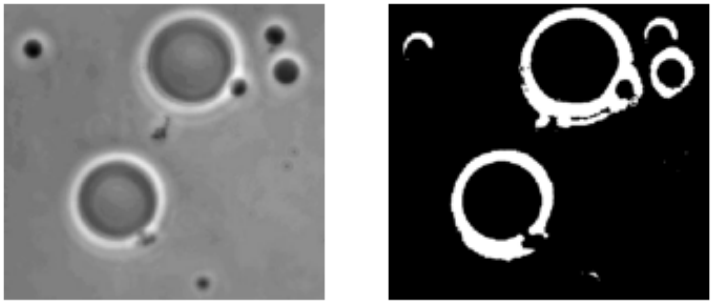

Figure 1.4: OTSU image segmentation

设置不同的随机种子，使用 k-means 算法进行图像分割处理，结果如下：

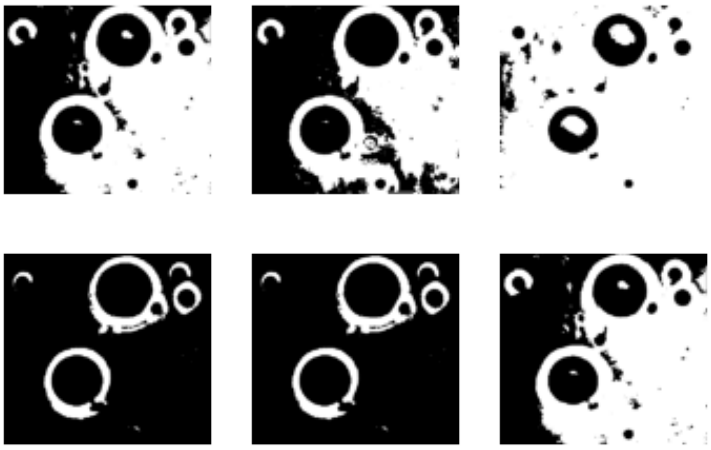

Figure 1.5: k-means image segmentation

```
random seed 1:
iteration:1,object fucntion:52137451
iteration:2,object fucntion:48759315
iteration:3,object fucntion:48314580
iteration:4,object fucntion:48314580
random seed 2:
iteration:1,object fucntion:48150329
iteration:2,object fucntion:48150329
random seed 3:
iteration:1,object fucntion:59479540
iteration:2,object fucntion:58560592
iteration:3,object fucntion:58424137
iteration:4,object fucntion:58424137
random seed 4:
iteration:1,object fucntion:47707957
iteration:2,object fucntion:46752958
iteration:3,object fucntion:46453348
iteration:4,object fucntion:46372314
iteration:5,object fucntion:46372314
random seed 5:
iteration:1,object fucntion:49207343
iteration:2,object fucntion:47707957
iteration:3,object fucntion:46544293
iteration:4,object fucntion:46372314
iteration:5,object fucntion:46372314
random seed 6:
iteration:1,object fucntion:48273025
iteration:2,object fucntion:48314580
iteration:3,object fucntion:48314580
```

Figure 1.6: Object functions of different random seeds

由于 k-means 算法使用了随机的初始化，这可能会产生不同的聚类结果，因为 k-means 只能实现局部最优。我们使用不同的初始化进行探索，重复执行 k-means 算法，发现可以从中选择最后收敛的目标函数（所有像素点到其最近的类中心距离的平方和）最小的随机种子（seed4），作为我们最终的聚类方案。

采用 seed4 进行随机初始化，并对比 k-means 算法和 OTSU 算法的图像分割效果，结果如下：

Figure 1.7: Comparison between k-means and OTSU

可以发现结果图像完全相同，这与我们的理论分析一致。

1.2 多类分割

我们分别将图片进行 2、4、8、16 类分割，结果如下：

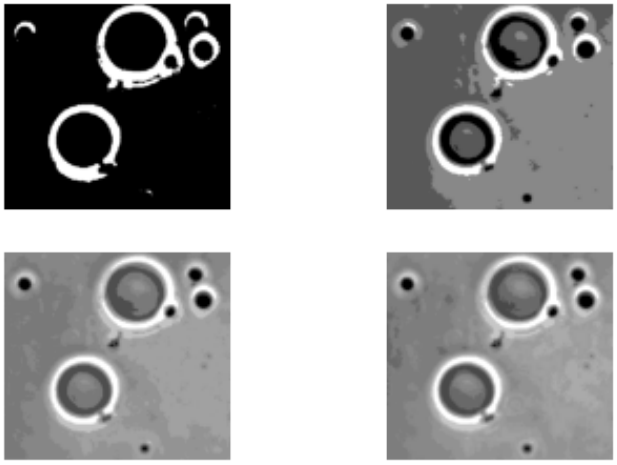

Figure 1.8: Comparison with different k types

可以发现，随着分割类别数目的增加，可以看到图片从视觉上越来越接近原图。

1.3 噪声图像的分割

code:

```python
def Add_SaltAndPepper_Nosie(img, p=0.1):
    """
    添加椒盐噪声
    :param img: 输入图片
    :param p: 噪声产生概率
    :return: 产生噪声的图片
    """
    L = 256
    # 添加盐噪声
    noise = np.random.uniform(0, L - 1, img.shape)
    mask = noise < p * (L - 1)
    img = img * (1 - mask)
    # 添加椒噪声
    mask = noise > (1 - p) * (L - 1)
    img = (L - 1) * mask + img * (1 - mask)
    return img
```

Figure 1.9: Add salt and pepper noise

我们首先观察不同随机数下，k-means 算法对噪声图像的分割效果：

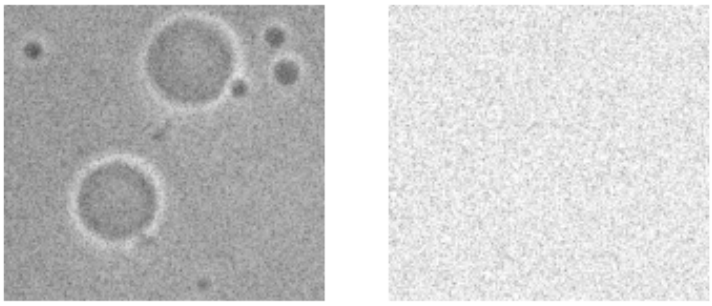

Figure 1.10: random seed 6

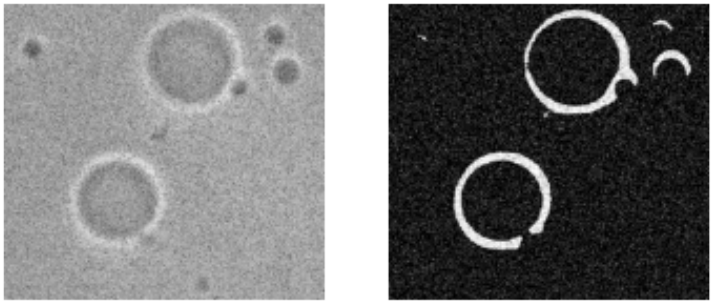

Figure 1.11: random seed 4

可以发现，k-means 不同的初始化，对噪声图像有不同的分割效果。第一种初始化的分割效果很差，第二种初始化有明显的分割效果，虽然仍然能看出些许噪音。
探究原因：
观察添加噪声前后图像的直方图变化，可以看出很多像素点的取值分布在 0 和 255 两处。

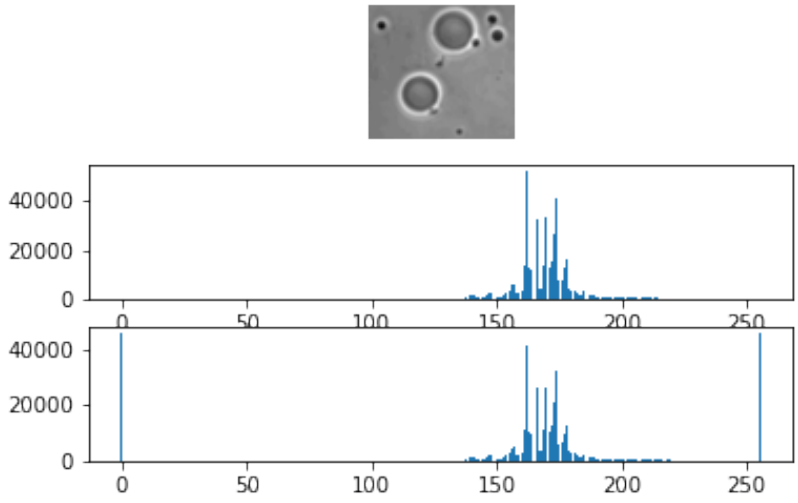

Figure 1.12: Histogram of image with and without noise

由于 k-means 算法的本质和 OSTU 相同，我们用 OSTU 探究图像分割的过程，以下是使用 OSTU 分别对原图和噪声图片产生的阈值，发现噪声图片的阈值为 0.

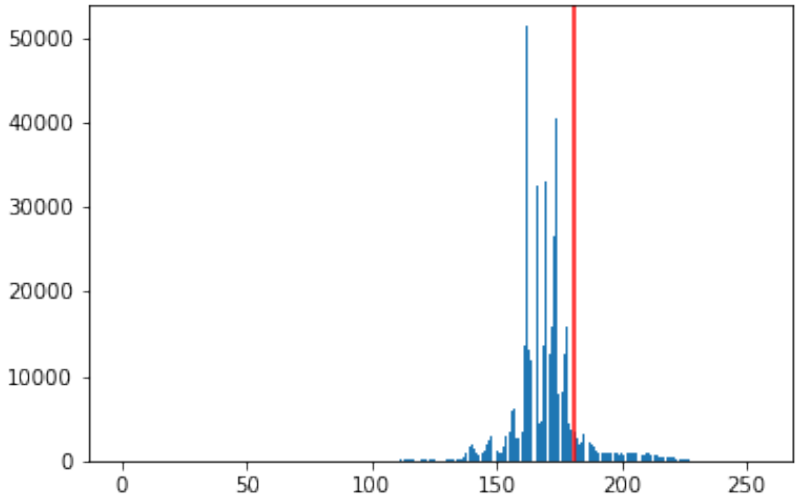

Figure 1.13: Threshold using OSTU of original image

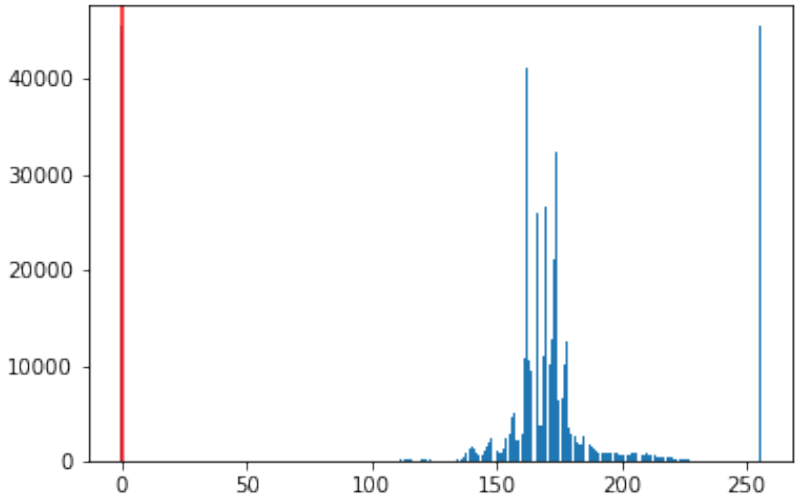

Figure 1.14: Threshold using OSTU of noisy image

我们由此得到启发，可以解释 k-means 算法在不同的初始化下的两种结果。查看两种随机初始化下图片分类结果的具体数值，结果如下：

```
first cluster: 179
second cluster: 0
```

Figure 1.15: random seed 6 cluster value

```
first cluster: 241
second cluster: 148
```

Figure 1.16: random seed 4 cluster value

我们发现，当随机初始化位置位于原图像素点（180 左右）附近时，k-means 算法仍有可能有分割效果；但当随机初始化位置位于 0 附近时，k-means 算法将 0 单独作为一类，其他均为另一类，此时没有分割效果。

从聚类算法的角度分析，离群值（椒盐噪声）对非监督聚类的影响很明显，尤其是对于 k-means 算法，由于其要将每个点都划分到一个簇中，单个噪音点也可以对整个簇造成很大的扰动。

更好的方法，可以考虑使用混合模型，比如高斯混合模型，或者 Soft k-means（一种高斯混合模型的特例），或者改用密度聚类。由于密度聚类能够同时优化簇内的紧密度和簇之间的离散度，因此超过一定阈值的点就可以被当作异常值标记出来，我们就可以一定程度上排除噪声的影响。

2 使用形态学操作实现二值图像补洞和离散点去除

code:

```python
def im_erode_expand(image, target=255, k=3):
    """
    实现腐蚀或膨胀
    :param image: 输入图像
    :param target: 目标物体
    :param k: 腐蚀或膨胀尺度
    :return: 返回腐蚀或膨胀后图像
    """
    m, n = image.shape
    # padding
    edge = k//2
    row = m + edge * 2
    col = n + edge * 2
    if target == 255:
        img = np.ones((row,col),dtype='int32')*255
    else:
        img = np.zeros((row,col),dtype='int32')
    img[edge:row-edge,edge:col-edge] = image
    # kernal
    if target == 255:
        kernal = np.zeros((k, k))
    else:
        kernal = np.ones((k, k))*255
    new_img = img.copy()
    # 遍历
    for i in range(m):
        for j in range(n):
            # 检查边界
            if img[edge+i,edge+j] == target:
                if img[edge+i-1,edge+j] == target and img[edge+i+1,edge+j] == target \
                        and img[edge+i,edge+j-1] == target and img[edge+i,edge+j+1] == target \
                    and img[edge+i-1,edge+j-1] == target and img[edge+i-1,edge+j+1] == target \
                    and img[edge+i+1,edge+j-1] == target and img[edge+i+1,edge+j+1] == target:
                        e = 1
                    else:  # 边界点
                        new_img[i:i+2*edge+1, j:j+2*edge+1] = kernal
    return new_img[edge:row-edge,edge:col-edge]
```

Figure 2.1: image erode and expand operation

```python
def im_open(image, target=255, k=3):
    new_target = 0 if target == 255 else 255
    inter_ary = im_erode_expand(image,target, k) # 先腐蚀
    new_ary = im_erode_expand(inter_ary,new_target, k) # 再膨胀
    return new_ary
def im_close(image, target=255, k=3):
    new_target = 0 if target == 255 else 255
    inter_ary = im_erode_expand(image,new_target, k) # 先膨胀
    new_ary = im_erode_expand(inter_ary,target, k) # 再腐蚀
    return new_ary
```

Figure 2.2: image open and close operation

在我们的腐蚀膨胀函数中，我们采用了 D8-chessboard 的距离衡量方式，先判断像素点是否是目标物体的边界，再进行腐蚀膨胀操作。由于对后景的腐蚀（膨胀）等价于对前景的膨胀（腐蚀），我们的函数只需要实现腐蚀操作，通过改变函数中 target 的值（255 或 0）就可以实现对图片中目标物体的腐蚀、膨胀操作。
在有离散点的二值图像上测试我们的腐蚀膨胀函数，结果如下：

Figure 2.3: erode and expand experiment

发现实现了正常的腐蚀、膨胀效果。
在有离散点的二值图像上使用开、闭操作，结果如下：

Figure 2.4: open and close result

我们知道开操作可以去除离散点，闭操作可以连接断点（补洞），通过先开后闭我们得到了没有离散点和空洞的图像。