HW2

赵心怡 19307110452

1.Restate the Basic Global Thresholding (BGT) algorithm so that it uses the histogram of an image instead of the image itself. (Please refer to the statement of OSTU algorithm)

基于直方图的BGT算法过程如下：

1. 选择初始阈值T0
2. 利用T0把图片分成前景和背景
3. 此时 $\mu_{BG}=m_{BG}/w_{BG}$,其中$m_{BG}=\sum_{i\in[0,T]}I_p(i)$,$w_{BG}=\sum_{i\in[0,T]}p(i)$

   $\mu_{FG}=m_{FG}/w_{FG}$,其中$m_{FG}=\sum_{i\in[T+1,255]}I_p(i)$,$w_{FG}=\sum_{i\in[T+1,255]}p(i)$
4. 由此计算新的阈值$T_1=\frac{\mu_{BG}^0+\mu_{FG}^0}{2}$
5. 重复步骤2-4直到收敛

BGT算法的代码如下：

```python
def BGT(img):
    T_initial = np.min(img) /2+ np.max(img)/2
    curr_T = int(np.ceil(T_initial)) # round
    hist = np.zeros(256)
    counter = Counter(img.reshape(-1)) # count frequency
    for i in range(256):
        hist[i] = counter[i]
    hist = hist / np.sum(hist) # get frequency hist

    # begin iteration
    for i in range(10000):
        # calculate the mean of BG and FG
        w_bg = np.sum(hist[:curr_T]) # probability of BG
        vec_bg = np.array(range(curr_T)) # form vectors of gray value
        mu_bg = vec_bg.dot(hist[:curr_T]) # dot with frequency
        mu_bg = mu_bg / w_bg if w_bg else 0 # get prob

        w_fg = np.sum(hist[curr_T:]) # probability of FG
        vec_fg = np.array(np.arange(curr_T, 256))
        mu_fg = vec_fg.dot(hist[curr_T:])
        mu_fg = mu_fg / w_fg if w_fg else 0

        next_T = int(np.ceil((mu_bg + mu_fg) / 2)) # new mean
        if next_T ==curr_T: # finish iteration condition
            final_T = next_T
            newImg = (img > final_T) * 255 #separate image to 0-1 gray
            return newImg

        curr_T = next_T # next iteration

    print('算法在10000步内仍不收敛')#max iteration exceed
    return None
```

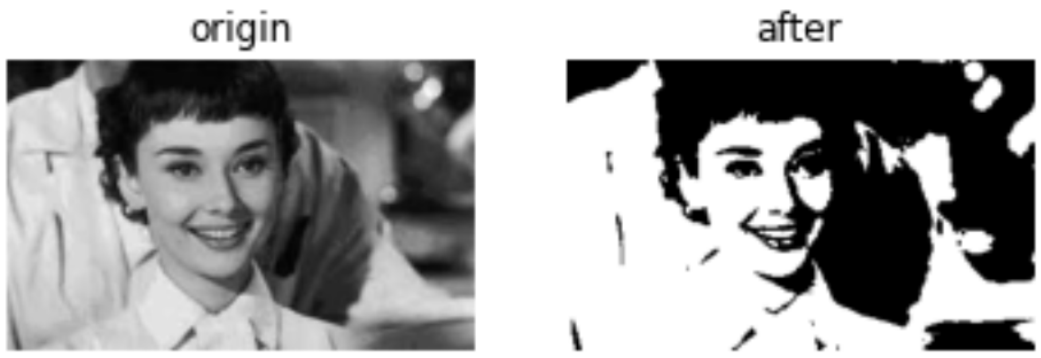
初始阈值curr_T设置为图片灰度值的最大值和最小值之间的中心值。用counter来获取图片灰度值的直方图分布。

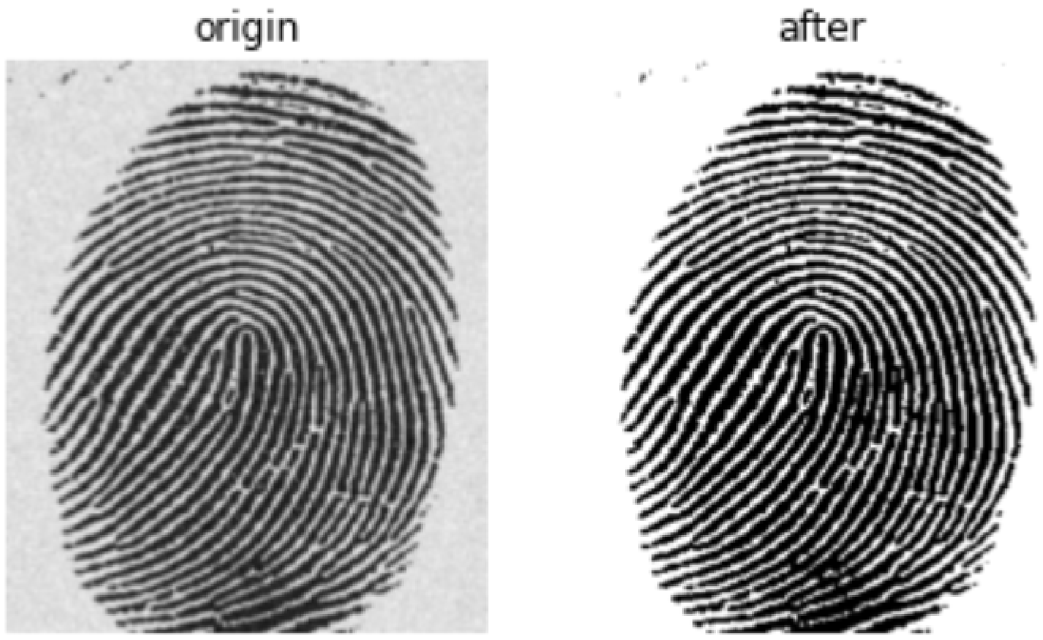
在此基础上分割两部分然后利用如上的公式进行迭代。

我们将迭代次数上限设为10000步，因为curr_T是严格取整的值，所以我们可以设停止迭代的条件是curr_T=next_T。 画图方法用的是第一次作业的画图函数，用到了PIL的库。

从结果上看算法可以较好的分割图像，而且计算速度非常快。

最后的输出结果：





2.Design an algorithm of locally adaptive thresholding based on local OSTU or maximum of local entropy; implement the algorithm and test it on exemplar image(s).

实现局部自适应的阈值分割。

首先我们参考上次作业的局部适应灰度值均衡化算法，写出更新局部直方图函数：

```python
def update_hist(center_pixel, img,previous_hist,side_length):
    # update hist based on past pixel
    (x, y) = center_pixel # pixel center
    half_patch = (side_length - 1) // 2
    if center_pixel == (half_patch, half_patch):    # first pixel
        newhist = Counter(img[:side_length, :side_length].reshape(-1))
    # z-shaped scanning
    if (x - half_patch) % 2 == 0:
        if y == half_patch:  # when x even: move down
            newhist =  previous_hist + Counter(img[x + half_patch, :side_length]) \
            - Counter(img[x - 1 - half_patch, :side_length])
            return newhist
        else: # move on right
            newhist = previous_hist + Counter(img[(x - half_patch):(x + half_patch + 1), y + half_patch]) \
                - Counter(img[(x - half_patch):(x + half_patch + 1), y - 1 - half_patch])
            return newhist
    pic_width = np.size(img, 1)
    if y == pic_width - half_patch - 1:  # when x odd, move down
        newhist = previous_hist + Counter(img[x + half_patch, pic_width - side_length:]) \
            - Counter(img[x - 1 - half_patch, pic_width - side_length:])
        return newhist
    else: # move on left
        newhist = previous_hist + Counter(img[(x - half_patch):(x + half_patch + 1), y - half_patch]) \
            - Counter(img[(x - half_patch):(x + half_patch + 1), y + half_patch + 1])
        return newhist
```

类似于上一次的作业，这次作业也需要用到小窗的功能，我们设小窗的大小patch size, algorithm表示输入的算法。采用z字形移动的方法进行小窗的更新。从高效更新函数中得到当前小窗的直方图，然后转化成概率值进行计算。

然后是局部阈值更新的算法代码：

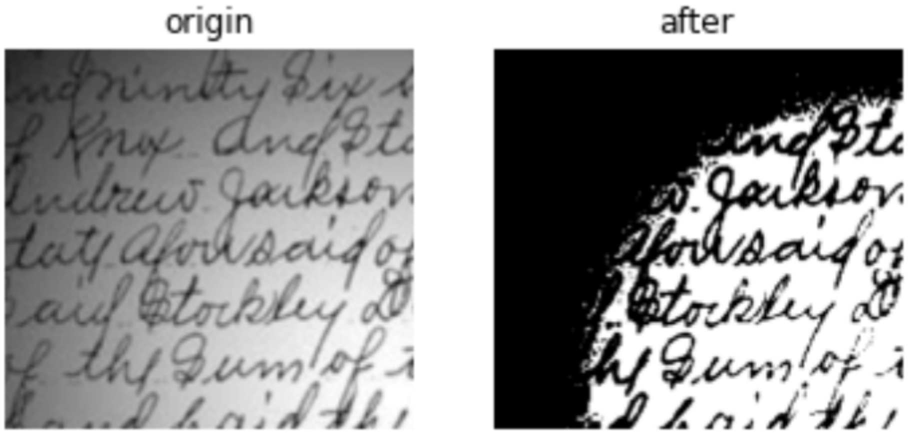
```python
def local_threshold(img, len_patch, alg):
    height, width = img.shape
    half_patch = (len_patch - 1) // 2
    tempImg = np.zeros((height + len_patch - 1, width + len_patch - 1))  # a larger temp matrix
    tempImg[half_patch: height + half_patch, half_patch: width + half_patch] = img
    newImg = np.zeros((height + len_patch - 1, width + len_patch - 1))
    local_hist = Counter() # local histogram
    for x in range(half_patch, height + half_patch):
        Y_order = range(half_patch, width + half_patch)
        if (x - half_patch) % 2:
            Y_order = reversed(Y_order) # z-shape movement window
        for y in Y_order: #
            curr_patch = img[x - half_patch:x + half_patch + 1, y - half_patch:y + half_patch + 1].flatten()  # current data scanned
            patch_max = np.max(curr_patch)   # max and min gray value in current patch
            patch_min = np.min(curr_patch)
            criterion_value = np.zeros(256)
            local_hist = update_hist((x, y), tempImg, local_hist, len_patch) # get current hist
            hist = np.zeros(256)
            for threshold in range(256):
                hist[threshold] = local_hist[threshold]
            hist = hist/np.sum(hist) # get hist
            for threshold in range(patch_min,patch_max+1):
                w0 = np.sum(hist[:threshold]) # frontground
                w1 = np.sum(hist[threshold:]) # background
                if alg == 'OSTU_hist':
                    u0 = np.array(range(threshold)).dot(hist[:threshold]) / w0 if w0 else 0 # mean of frontground
                    u1 = np.array(np.arange(threshold,256)).dot(hist[threshold:]) / w1 if w1 else 0 # mean of background
                    criterion_value[threshold] = w0 * w1 * (u0-u1)**2 # var between groups
                if alg == 'entropy':
                    prob_bg = hist[:threshold]/w0 if w0 else np.zeros(threshold) # probs of each group
                    prob_fg = hist[threshold:]/w1 if w1 else np.zeros(255-threshold)
                    prob_bg[np.where(prob_bg==0)] = 1.0 # normalization for easy calculate
                    prob_fg[np.where(prob_fg==0)] = 1.0
                    h0 = np.array(prob_bg).dot(np.log2(prob_bg)) # cal entropy
                    h1 = np.array(prob_fg).dot(np.log2(prob_fg))
                    criterion_value[threshold]=-h0-h1

            best_threshold = np.argmax(criterion_value) # choose max value for best threshold
            newImg[x][y] = tempImg[x][y] >= best_threshold  # relationship between the current center point and threshold

    newImg = newImg[half_patch:height + half_patch, half_patch:width + half_patch] # get new image
    return newImg * 255
```

在大津算法中要最大的值为组间方差， 而在熵算法中是熵，我们分两种情况计算criterion value, 并对每个小窗中的最优阈值保存下来再在新的图片中判断更新为0或者1，填入新图像。遍历完所有x,y之后，得到最终图像，并转化为灰度值范围。

我们选择课件上的照片作为样例进行分析，因为OSTU计算速度非常慢，我们截取了图片的一个角进行分析，并且比较了不同算法结果下的输出图片。

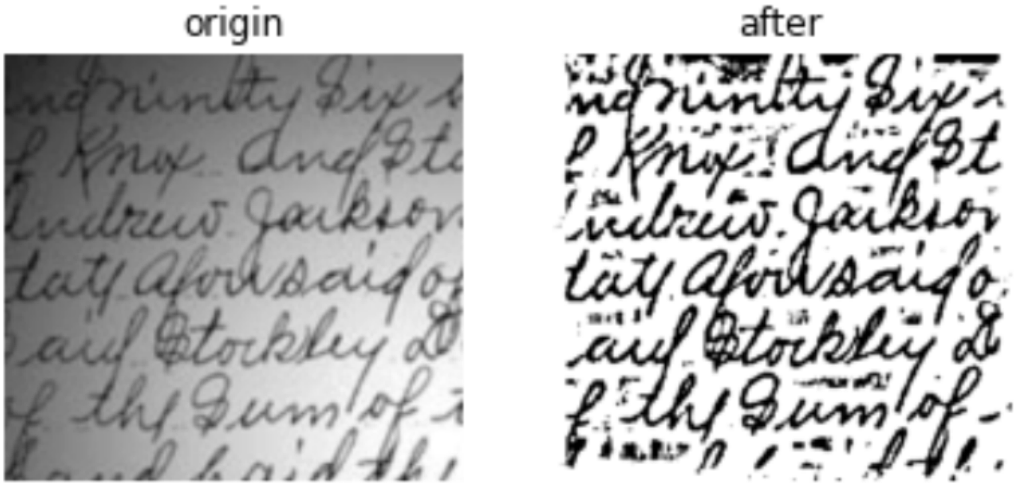
首先用第一种方法的BGT算法，发现结果左上角的字变成了全黑看不到了。

direct BGT 的结果：



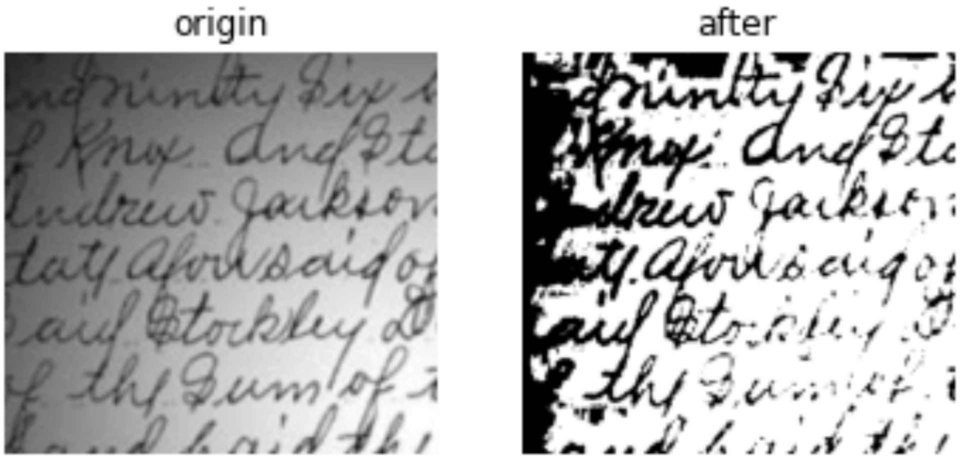
然后我们测试了基于直方图算均值的OSTU算法，在小窗的大小是25的情况下得到的结果如图。

OSTU_hist size=25:



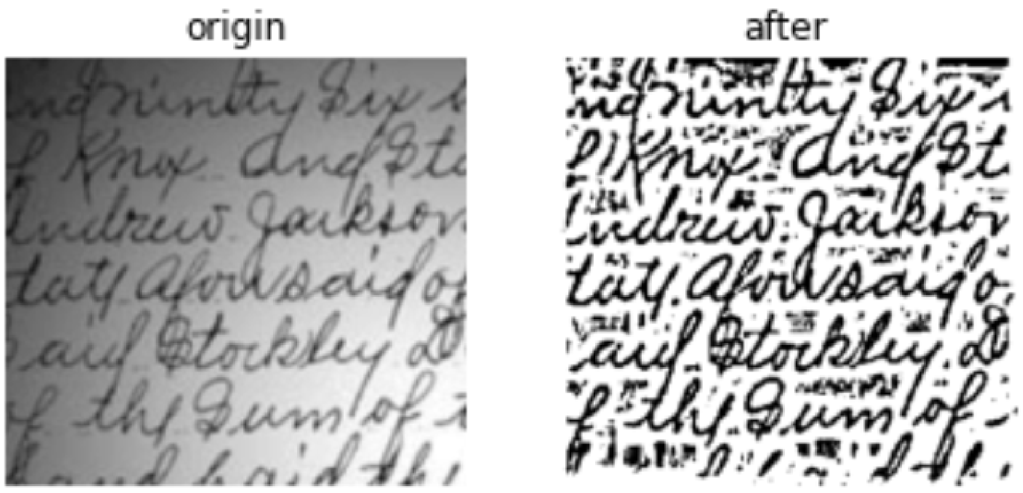
在同样的条件下测试了熵算法在小窗大小是25的情况下得到的结果。

OSTU entropy size=25：



出现了很多的大面积色块，猜测是因为小窗太大导致，重新调整小窗的大小：

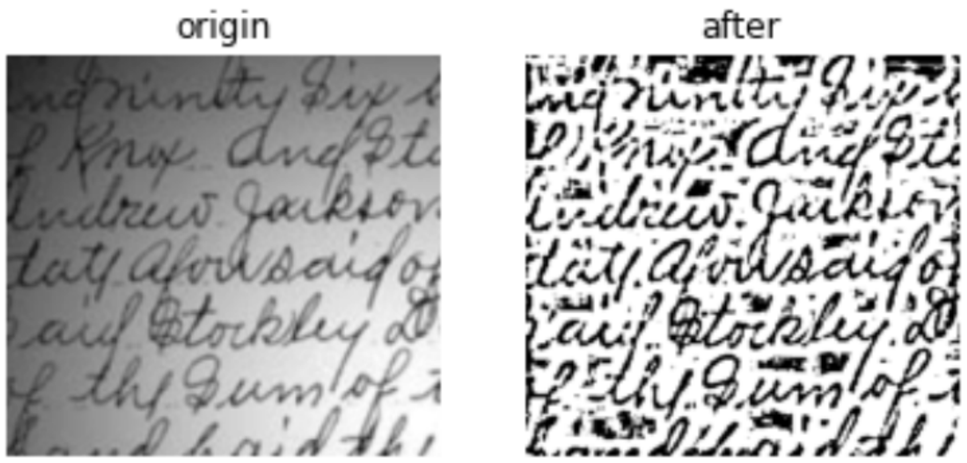
OSTU entropy size=19:



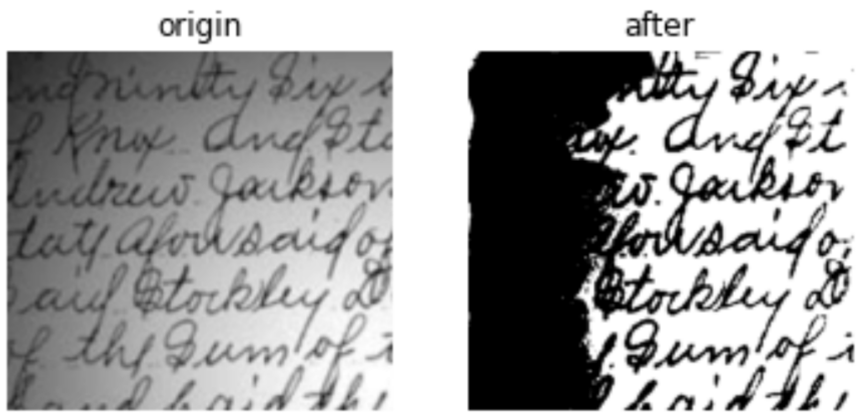
猜测导致熵算法和大津算法的结果不同的原因可能与这两个算法需要计算大量小数相对不是那么精确有关。加上原本的截图可能有噪点导致最后的结果不是特别好看。

在比较不同小窗大小的时候我们发现小窗的大小会对结果产生很大的影响。在size较小的时候输出的图像会有很多背景的噪点，在白色区域出现黑块。 而小窗过大时候会出现正片的大面积黑色，越来越接近全局阈值分割的效果，失去局部的良好性质。说明分割效果不是很理想。

OSTU hist size = 20



OSTU hist size = 30



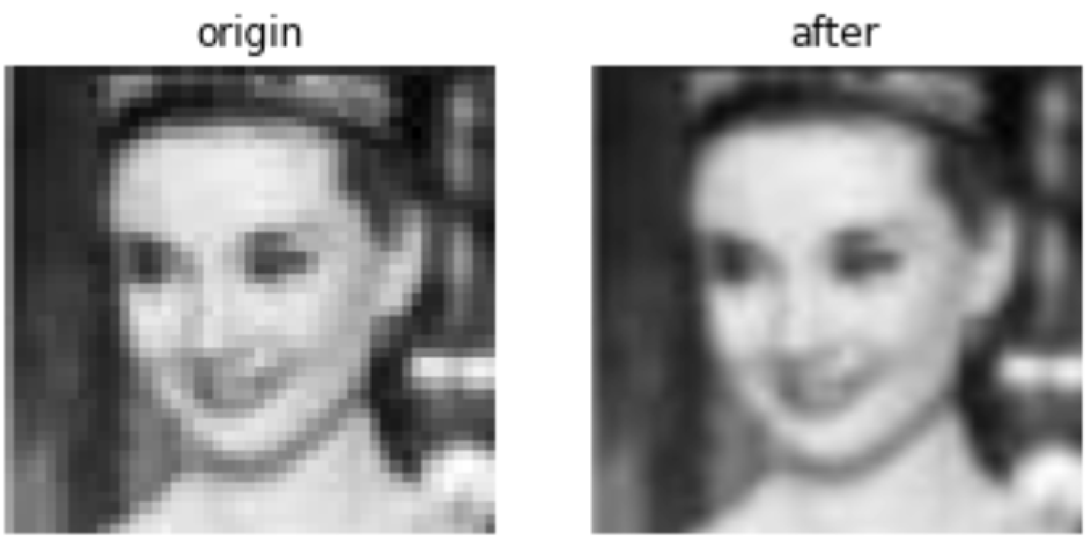
3.编程实现线性插值算法(不能调用某个算法库里面的插值函数)，并应用：读出一幅图像，利用线性插值把图片空间分辨率放大N倍，然后保存图片。

双线性插值是有两个变量的插值函数的线性插值扩展.双线性插值是分别在两个方向计算了单线性插值，先在x方向求2次单线性插值，获得R1(x, y1)、R2(x, y2)两个临时点，再在y方向计算1次单线性插值得出P(x, y)

代码如下：

```
def bilinear_intercept(img, N):
    # bilinear intercept algorithm
    height, width = img.shape
    new_height, new_width = int(height * N), int(width * N) # new height and width of image
    new_img = np.zeros((new_height, new_width))
    prop_of_height, prop_of_width = (height - 1) / (new_height- 1), (width - 1) / (new_width - 1) # proportion of resize

    # add margin with zero value
    temp_img = np.zeros((height+1, width+1))
    temp_img[0 : height,0 :width] = img

    for i in range(new_height):
        for j in range(new_width): # for each pixel
            neigh_i, neigh_j = int(i * prop_of_height), int(j * prop_of_width)   # nearest neighbor
            u, v = i * prop_of_height - neigh_i, j * prop_of_width - neigh_j # estimate the intensity at given locations
            new_img[i][j] = u * v * temp_img[neigh_i + 1][neigh_j + 1] + (1 - u) * (1 - v) * temp_img[neigh_i][neigh_j] + \
                               u * (1 - v) * temp_img[neigh_i + 1][neigh_j] +(1 - u) * v * temp_img[neigh_i][neigh_j + 1]
                               # interpolation
    return new_img
```

方大倍数为10的图像处理结果：



图像的像素大小从45*47放大到450*470

观察对比可以发现图片变得更加细了，脸部可以观察到五官，没有明显的马赛克。但是变换后的图像还是很模糊。猜测这是因为线性插值算法求平均的结果，图像的信息并没有增加，所以无法返回清晰的结果。

我们测试了放大倍数更大，结果没有很明显的差别。说明线性插值的算法提升是有限的，不可能增加图片已有的信息，让图片更清晰。