Федеральное государственное автономное образовательное учреждение высшего образования

«Национальный исследовательский университет ИТМО»

Отчет по лабораторной работе №5
«Сегментация изображений»
по дисциплине «Техническое зрение»

Выполнил: студент гр. R3338,
Кирбаба Д.Д.
Преподаватель: Шаветов С.В.,
канд. техн. наук, доцент ФСУ и Р

Санкт-Петербург, 2022

Цель работы

Освоение основных способов сегментации изображений на семантические области.

Теоретическое обоснование применяемых методов

Сегментация – это процесс разделения изображения на несколько сегментов (областей). Основная цель – упростить или поменять представление изображение в нечто более осмысленное и удобное для анализа.

Самый простой вид сегментации – бинаризация. Это разделение изображение на объект и фон.

Например, бинаризация с одним пороговым значением:

$$I_{new} = \begin{cases} 0, I(x,y) \le t, \\ 1, I(x,y) > t. \end{cases}$$

Или двойная бинаризация:

$$I_{new} = \begin{cases} 0, I(x,y) \le t_1, \\ 1, t_1 < I(x,y) \le t_2, \\ 0, I(x,y) > t_2. \end{cases}$$

Естественно, от значений порогов t, t_1, t_2 зависит результат применения преобразования. Выбирать их можно как вручную, так и используя специальные алгоритмы.

Рассмотрим 4 алгоритма вычисления пороговых значений для бинаризации изображения:

1. Среднее от максимальной и минимальной интенсивности изображения:
$$t = \frac{I_{max} - I_{min}}{2};$$
2. Вычисление на основе модуля градиента интенсивности пикселей:
$$G(x,y) = \max\{|I(x+1,y) - I(x-1,y)|, |I(x,y+1) - I(x,y-1)|\},$$
$$t = \frac{\sum_{x=0}^{X-1}\sum_{y=0}^{Y-1} I(x,y)G(x,y)}{\sum_{x=0}^{X-1}\sum_{y=0}^{Y-1} G(x,y)};$$
3. Метод Оцу, который делит пиксели на 2 класса так, что дисперсия между классами максимальна, а дисперсия внутри классов минимальна;
4. Адаптивный метод, который воздействует только на некоторые области изображения. Часто используется в случаях неравномерно освещенных объектов.

Следующий алгоритм сегментации – алгоритм, основанный на принципе Вебера.

Принцип Вебера предполагает, что человеческий глаз почти не различает уровни серого между интенсивностями $I(n)$ и $I(n) + W(I(n))$, где $W(I(n))$ – функция Вебера, которая считается по формуле:

$$W(I) = \begin{cases} 20 - \frac{12I}{88}, 0 \le I \le 88, \\ 0.002(I - 88)^2, 88 < I \le 138, \\ \frac{7(I - 138)}{117} + 13, 138 < I \le 255. \end{cases}$$

Таким образом можно слить уровни серого из диапазона $[I(n), I(n) + W(I(n))]$, заменив одним значением интенсивности.

Разберем ещё один алгоритм сегментации – сегментация по цвету кожи.

Суть алгоритма – определить критерий близости интенсивности пикселя к цвету кожи.

Сегментацию по цвету кожи можно использовать в различных приложениях, таких как распознавание лиц, анализ жестов рук, фильтрация содержимого изображений и т. д.

Однако существуют некоторые проблемы при обнаружении цвета кожи, такие как широкий спектр условий освещенности и появление объектов цвета кожи на фоне изображения.

Есть несколько аналитически описаний для изображений в цветовом пространстве RGB, которые позволяют присвоить пиксель к определенному классу.

Условия для равномерного дневного освещения:

$$\begin{cases} R > 95, \\ G > 40, \\ B > 20, \\ maxR, G, B - minR, G, B > 15, \\ |R - G| > 15, \\ R > G, \\ R > B. \end{cases}$$

Условия при свете вспышки или при дневном боковом освещении:

$$\begin{cases} R > 220, \\ G > 210, \\ B > 170, \\ |R - G| \le 15, \\ G > B, \\ R > B. \end{cases}$$

Условия с помощью нормализованных значений RGB:

$$\begin{cases} r = \dfrac{R}{R+G+B}, \\ g = \dfrac{G}{R+G+B}, \\ b = \dfrac{B}{R+G+B}, \\ \dfrac{r}{g} > 1.185, \\ \dfrac{rb}{(r+g+b)^2} > 0.107, \\ \dfrac{rg}{(r+g+b)^2} > 0.112. \end{cases}$$

Сегментация изображения может быть легко произведена с помощью перевода изображения в цветовое пространство *CIE Lab*.

Изображение в пространстве *CIE Lab* имеет следующие три компоненты:

1. L – значение яркости, изменяется от 0 (темнота) до 100 (свет).
2. a – оттенок, обозначает цветовую позицию от зеленого (–128) до красного (127).
3. b – насыщение, обозначает цветовую позицию от синего (–128) до желтого (127).

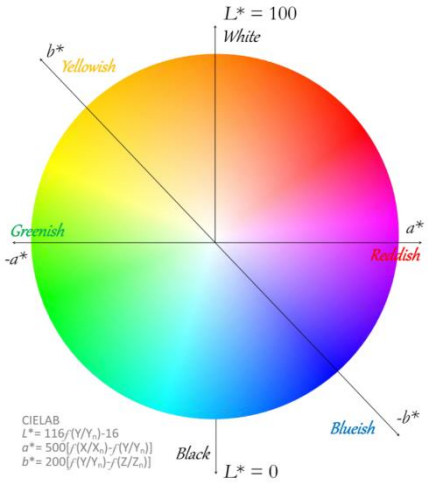

Figure 1: цветовое пространство CIE Lab.

Суть алгоритма – разделение изображения на кластеры, в которых доминируют определенные цвета.

Рассмотрим следующий алгоритм сегментации, основанный на кластеризации k-средних.

Целью данного метода является разделение всех пикселей на k кластеров, при этом каждый пиксель относится к тому кластеру, к центру (центроиду) которого оно ближе всего. Алгоритм k-средних заключается в перевычислении на каждом шаге центроида для каждого кластера, полученного на предыдущем шаге.

Последний метод, рассматривающийся в это работе – текстурная сегментация.

Вообще говоря, текстуру можно описать тремя методами: статистически, структурно и спектрально. Тут будем рассматривать только статистический метод, то есть мы рассматриваем структуру как гладкую, грубую или зернистую.

Будем рассматривать интенсивность изображения I как случайную величину z, которая имеет функцию распределения $p(z)$, которая может быть вычислена из гистограммы изображения (делали так в первой лабораторной работе).

Центральный момент порядка n случайной величины z это следующая величина:

$$\mu_n(z) = \sum_{i=0}^{L-1} (z_i - m)^n p(z_i),$$

где L – количество уровней интенсивности в изображении, m – математическое ожидание случайной величины:

$$m = \sum_{i=0}^{L-1} z_i p(z_i).$$

Можем вычислить моменты порядков 0 и 1:

$$\mu_0(z) = 1, \qquad \mu_1(z) = 0.$$

При описании структуры дисперсия случайной величины играет важную роль. Она равна второму центральному моменту $\sigma^2(z) = \mu_2(z)$ и является мерой контраста яркости.

Дисперсия может быть использована для вычисления степени гладкости изображения. Например, введем меру относительной гладкости R:

$$R = 1 - \frac{1}{1 + \sigma^2(z)},$$

она принимает значение 0 в областях с постоянной интенсивностью (нулевая дисперсия) и приближается к 1 при больших значения дисперсии.

Необходимо также помнить, что для черно-белых изображений (интенсивность которых находится в диапазоне $[0, 255]$) нужно нормализовывать значение дисперсии в диапазон $[0, 1]$, так как изначальные значения дисперсии будут слишком большими. Проводить нормализацию нужно разделив дисперсию $\sigma^2(z)$ на $(L-1)^2$, где L – количество уровней интенсивности в изображении.

Также в качестве характеристики текстуры часто используется среднеквадратичное отклонение $\sigma(z)$.

Третий момент случайной величины характеризует симметрию гистограммы изображения.

Для оценки особенностей текстуры используется функция энтропии E, которая определяет разброс интенсивностей соседних пикселей:

$$E = -\sum_{i=0}^{L-1} p(z_i) \log_2 p(z_i).$$

Ещё одна величина, описывающая текстуру – мера однородности U, которая определяет однородности гистограммы:

$$U = \sum_{i=0}^{L-1} p^2(z_i).$$

Texture	m	s	$R \in [0,1]$
Smooth	82,64	11,79	0,0020
Rough	143,56	74,63	0,0079
Periodical	99,72	33,73	0,0170

Texture	$\mu_3(z)$	U	E
Smooth	−0,105	0,026	5,434
Rough	−0,151	0,005	7,783
Periodical	0,750	0,013	6,674

Figure 1: Значения параметров, определяющих текстуру.

Итак, после вычисления одной или нескольких параметров необходимо сгенерировать маски изображения для отделения одних текстур от других.

Ход выполнения работы

1. Бинаризация

Figure 2: исходное изображение.

Проведем сегментацию методами бинаризации данного изображения.

1.1. Бинаризация средним значением

Listing 1. Бинаризация средним значением на Python.

```
# Arithmetic mean of max and min intensities
t = (np.max(img) - np.min(img)) / 2
ret, bin_img = cv.threshold(img, t, 255, cv.THRESH_BINARY)
```

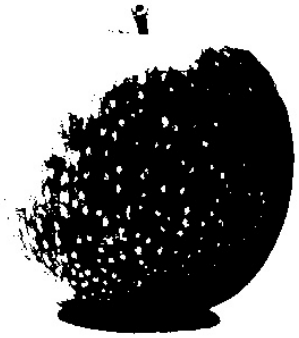

Figure 3: изображение после бинаризации средним значением интенсивности.

Результат довольно неплохой, однако нам повезло с изображением (объект хорошо выделяется на фоне).

Данный метод не подойдет при применении к светлым или темным изображениям.

1.2. Двойная бинаризация

Listing 2. Двойная бинаризация на Python.

```
# 2. Double binarization
t1 = 50
t2 = 120
ret, bin_img = cv.threshold(img, t2, 255, cv.THRESH_TOZERO_INV)
ret, bin_img = cv.threshold(bin_img, t1, 255, cv.THRESH_BINARY)
```

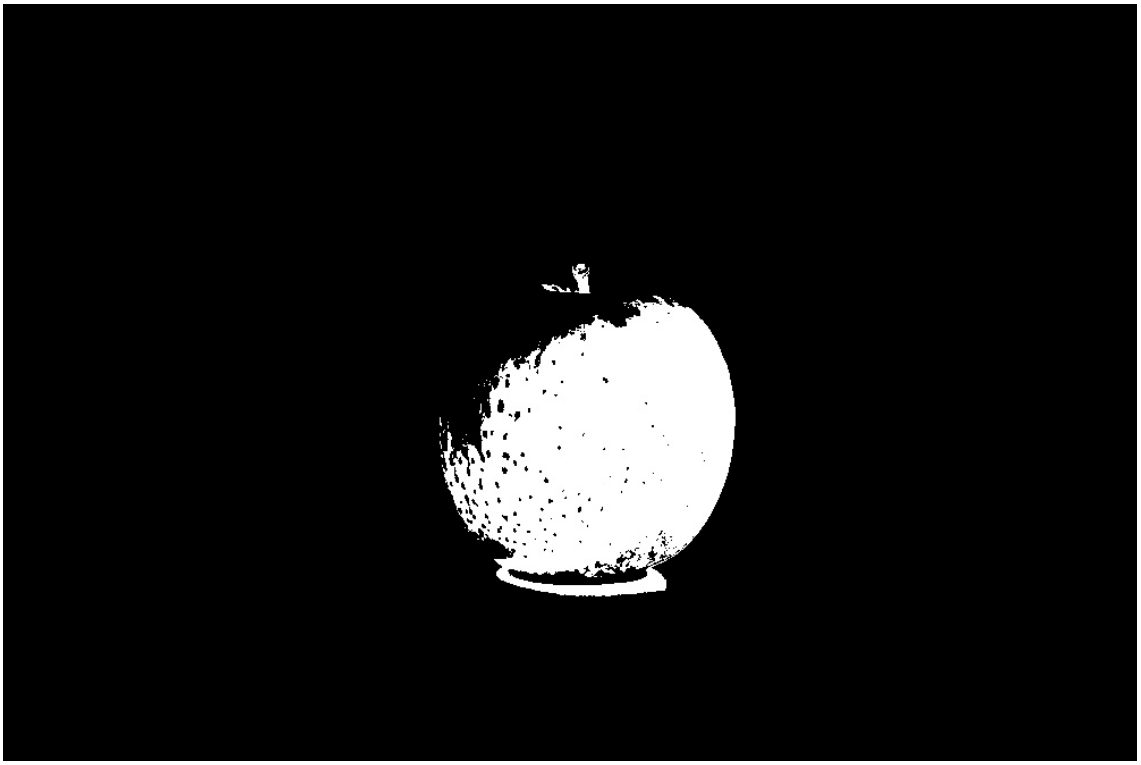

Figure 4: сегментация методом двойной бинаризации.

Данный метод имеет преимущества перед обычной бинаризацией за счет большей гибкости настроек границ, по которым будет проведена сегментация.

1.3. Бинаризация Оцу

Для автоматического поиска параметра t можно воспользоваться методом Оцу. Он основывается на исследовании гистограммы изображения, пытаясь найти оптимальный порог.

Listing 3. Бинаризация Оцу на Python.

```
# Otsu binarization
ret, bin_img = cv.threshold(img, 0, 255, cv.THRESH_OTSU)
```

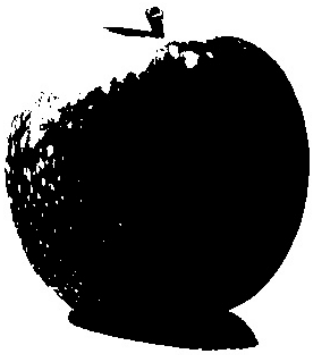

Figure 5: бинаризация изображения методом Оцу.

Найденное значение $t = 106.5$.

1.4.Бинаризация на основе градиента яркости

Listing 4. Бинаризация на основе градиента яркости на Python.

```
# Based on brightness gradient
mx = np.array([[0, 0, 0], [-1, 0, 1], [0, 0, 0]])
my = np.array([[0, -1, 0], [0, 0, 0], [0, 1, 0]])
# Compute gradients
grad_x = cv.filter2D(img, -1, mx, borderType=cv.BORDER_REFLECT)
grad_y = cv.filter2D(img, -1, my, borderType=cv.BORDER_REFLECT)
# Find modulus
grad = np.maximum(np.abs(grad_x), np.abs(grad_y))
# Compute t
t = np.sum(np.multiply(grad, img)) / np.sum(grad)
```

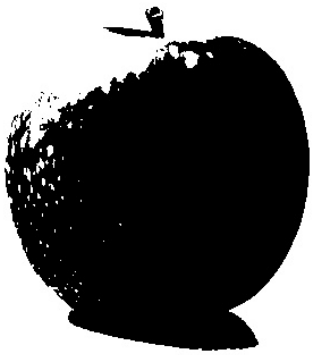

Figure 6: изображение после бинаризации на основе градиента яркости.

Найденное значение $t = 48.858$.

1.5. Бинаризация адаптивным методом.

Адаптивные методы, работающие не со всем изображением, а лишь с его фрагментами. Такие подходы зачастую используются при работе с изображениями, на которых представлены неоднородно освещенные объекты.

В *OpenCV* большое количество методов адаптивной бинаризации (что говорит об их популярности и эффективности), здесь приведу примеры адаптивной бинаризации методом Гаусса и медианную.

Listing 5. Бинаризация адаптивным методом на Python.

```
# Adaptive method
bin_img = cv.adaptiveThreshold(img, 255, cv.ADAPTIVE_THRESH_MEAN_C,
cv.THRESH_BINARY, 11, 2)
```

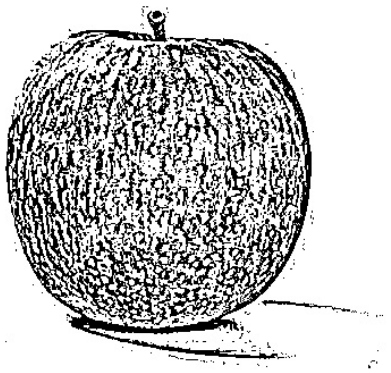

Figure 7: адаптивная бинаризация методом Гаусса.

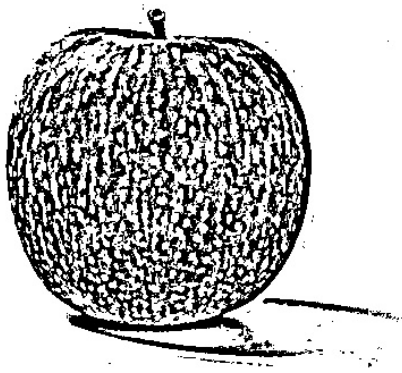

Figure 8: медианная адаптивная бинаризация.

Результаты схожи.

Итого, ожидаемо, лучше всего с задачей бинаризации справляются адаптивные методы.

И данные способы имеют место применяться для задачи сегментации в простых контролируемых условиях.

2. Сегментация лица

Рассмотрим несколько основных методов сегментации лица.

2.1. Сегментация на основе принципа Вебера

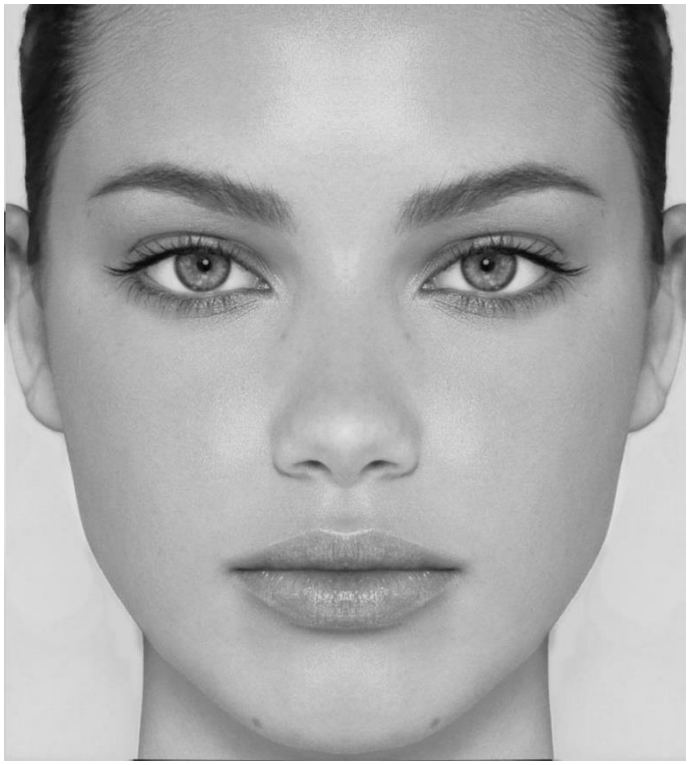

Figure 9: исходное изображение.

Listing 6. Сегментация по принципу Вебера на Python.

```
def weber(i):
    if i <= 88:
        return 20 - 12 * i / 88
    elif i <= 138:
        return 0.002 * np.power(i - 88, 2)
    else:
        return 7 * (i - 138) / 117 + 13
```

```
# Initial conditions
n = 0  # First class number
i_n = 0  # Grayscale level
i_w = weber(i_n)
i_min = np.min(img)
i_max = np.max(img)


new_img = np.zeros_like(img)
seg_val = []  # Intensity - segment boundaries

# Calculate the first intensity boundary
while i_min > (i_w + i_n):
    i_n += (i_w + 1)
    i_w = weber(i_n)

while i_max > (i_w + i_n):
    new_img[(img >= i_n) & (img <= i_n + i_w)] = i_n
    seg_val.append(i_n)
    i_n += (i_w + 1)
    n += 1
    i_w = weber(i_n)
new_img = np.clip(new_img, 0, 255).astype(np.uint8)


seg_val = np.uint8(seg_val)


# Show q-segment
q = 35
seg_q = np.zeros_like(new_img)
seg_q[new_img == seg_val[q - 1]] = 255


# Show [q1, q1] interval of segments
q1 = 1
q2 = 35
seg_int = np.zeros_like(new_img)
seg_int[(new_img >= seg_val[q1 - 1]) & (new_img <= seg_val[q2 - 1])] = 255
```

Итого, после выполнения программы, мы имеем $n = 36$ слоев данного изображения.

Отобразим все слои.

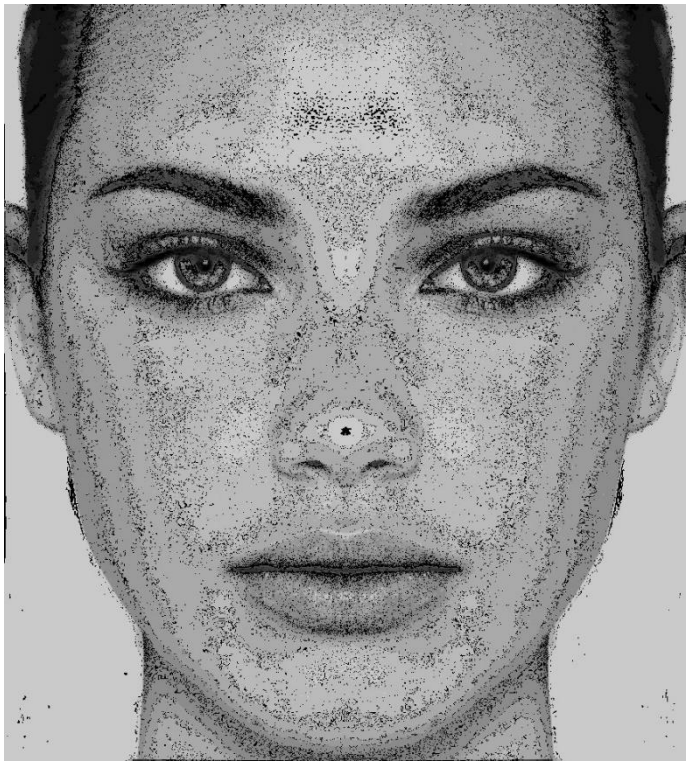

Figure 10: все слои сегментации.

Уже тут можем сказать, что так как фон имеет одну интенсивность, то выбрав нужные слои и скомбинировав их, мы сможем сегментировать лицо.

Выведем слой с наибольшей яркостью (последний в массиве seg_val).

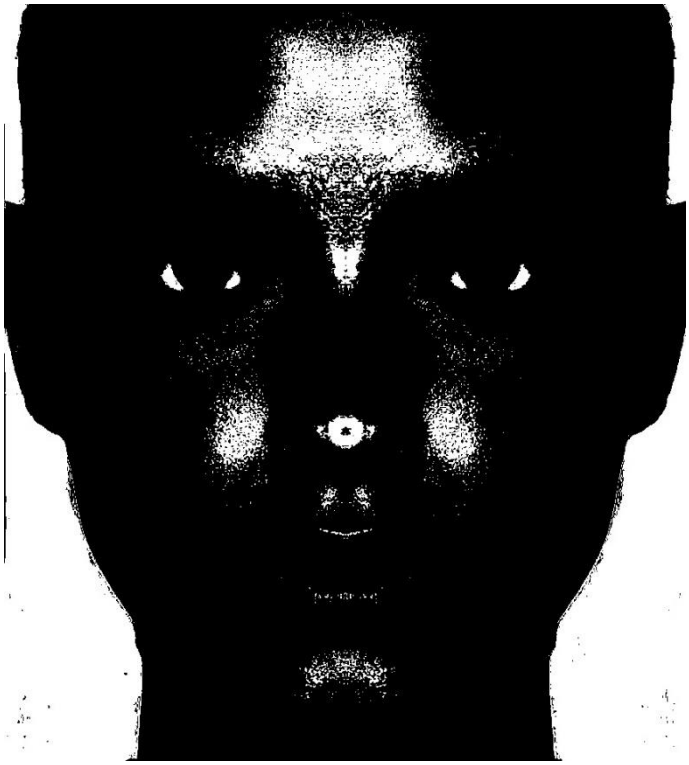

Figure 11: последний слой сегментации.

Действительно, данный слой содержит фон, а значит отделив его сможем выделить лицо.

Figure 12: все слои, кроме фонового.

Естественно, далее можно использовать морфологические фильтры для удаления внутренних и внешних дефектов, однако в данном случае нам важна сегментация, а не идеальная картинка.

Теперь присвоим всем пикселям интенсивности исходного изображения.

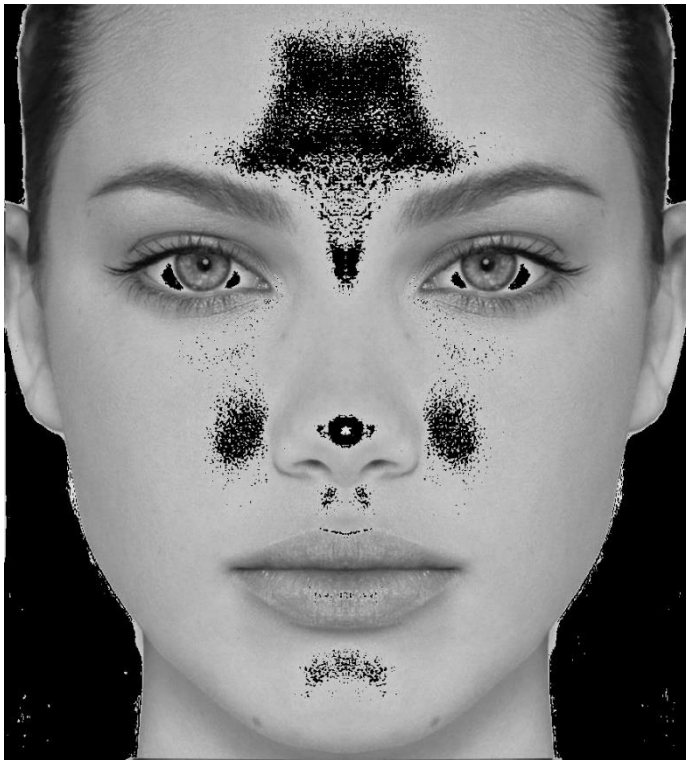

Figure 13: результат сегментации лица по принципу Вебера.

Результат сегментации довольно неплохой, однако все же присутствуют дефекты при одинаковой интенсивности фона и пикселей лица.

2.2. Сегментация по цвету кожи

Применим различные формулы к изображениям с различными условиями освещения.

2.2.1. Равномерное дневное освещение

Figure 14: исходное изображение.

Listing 7. Сегментация по цвету кожи при равномерном дневном освещении на Python.

```
def weighted_median_filtering(img, kernel_size=5):
    # Set parameters
    n_rows, n_cols = img.shape[:2]
    if kernel_size == 3:
        kernel = np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]])
    elif kernel_size == 5:
        kernel = np.array([[1, 2, 3, 2, 1], [2, 4, 5, 4, 2], [3, 5, 7, 5, 3], [2, 4,
5, 4, 2], [1, 2, 3, 2, 1]])
    kernel_shape = kernel.shape

    # Convert to float and make image with border
    img_copy = img.astype(np.float32) / 255
    img_copy = cv.copyMakeBorder(img_copy, int((kernel_shape[0] - 1) / 2),
int(kernel_shape[0] / 2), int((kernel_shape[1] - 1) / 2), int(kernel_shape[1] / 2),
cv.BORDER_REPLICATE)

    # Fill arrays for each kernel item
    img_with_areas = np.zeros(img.shape + (np.sum(kernel),), dtype=np.float32)
    cur_inx = 0
    for i in range(kernel_shape[0]):
        for j in range(kernel_shape[1]):
            # form array with same pixels
            expanded_arr = np.expand_dims(img_copy[i:i + n_rows, j:j + n_cols],
axis=2)
            res = expanded_arr
            for k in range(kernel[i, j] - 1):
                res = np.concatenate((res, expanded_arr), axis=2)
            # filling
```

```
            img_with_areas[:, :, cur_inx:(cur_inx + kernel[i, j])] = res
            cur_inx += kernel[i, j]

    # Sort arrays
    img_with_areas.sort()

    # Choose layer with concrete rank
    img_new = img_with_areas[:, :, np.prod(kernel_shape) // 2]

    # Convert back
    img_new = np.clip(255 * img_new, 0, 255).astype(np.uint8)

    return img_new


# Uniform daylight illumination
mask = np.logical_and(np.logical_and(img[:, :, 0] > 20, img[:, :, 1] > 40), img[:,
:, 2] > 95)  # R>95, G>40, B>20 conditions
mask = np.logical_and(np.logical_and(mask, img[:, :, 2] > img[:, :, 1]), img[:, :,
2] > img[:, :, 0])  # R>G, R>B conditions
mask = np.logical_and(mask, np.abs(img[:, :, 2] - img[:, :, 1]) > 15)  # |R-G|>15
condition
mask = np.logical_and(mask, (np.max(img, axis=2) - np.min(img, axis=2)) > 15)  #
maxRGB - minRGB > 15 condition


# Apply mask
new_img = np.zeros_like(mask, dtype=np.uint8)
new_img[mask] = 255

# Apply median filtering
new_img = weighted_median_filtering(new_img, 3)


# Create unbinarized image
new_img = np.zeros_like(img, dtype=np.uint8)
new_img[mask, :] = img[mask]
new_img = cv.cvtColor(new_img, cv.COLOR_BGR2RGB)
```

Figure 15: бинаризация результата сегментации.

Figure 16: сегментированное лицо.

Полученная маска действительно содержит лицо, однако также захватывает большое количество фона.

2.2.2. Освещение фонарем либо боковое дневное

Figure 17: исходное изображение с боковым освещением.

Listing 8. Сегментация лица по цвету кожи при боковом освещении на Python.

```
# Under flashlight or daylight lateral illumination
mask = np.logical_and(np.logical_and(img[:, :, 0] > 170, img[:, :, 1] > 210), img[:,
:, 2] > 220)  # R>220, G>210, B>170 conditions
mask = np.logical_and(np.logical_and(mask, img[:, :, 1] > img[:, :, 0]), img[:, :,
2] > img[:, :, 0])  # G>B, R>B conditions
mask = np.logical_and(mask, np.abs(img[:, :, 2] - img[:, :, 1]) <= 15)  # |R-G|<=15
condition
# Apply mask
new_img = np.zeros_like(mask, dtype=np.uint8)
new_img[mask] = 255

# Apply median filtering
new_img = weighted_median_filtering(new_img, 3)
```

Figure 18: маска сегментации.

Полученная маска действительно является частью лица, однако в каких-то местах качество такой сегментации, как и ожидалось, страдает.

2.2.3. Используя нормализованные значения RGB

Figure 19: исходное изображение.

Listing 9. Сегментация по цвету кожи, используя нормализованные значения RGB на Python.

```
# Using normalized RGB values
# Convert to initial img to float
img = img.astype(np.float64)
# Convert to normalized RGB
bgr_sum_img = np.sum(img, axis=2)
img_norm_b = np.divide(img[:, :, 0], bgr_sum_img)
img_norm_g = np.divide(img[:, :, 1], bgr_sum_img)
img_norm_r = np.divide(img[:, :, 2], bgr_sum_img)

# Create mask
denum = np.power(img_norm_r + img_norm_g + img_norm_b, 2)  # Auxiliary array
mask = np.logical_and(np.divide(np.multiply(img_norm_r, img_norm_b), denum) > 0.107,
np.divide(np.multiply(img_norm_r, img_norm_g), denum) > 0.112)
mask = np.logical_and(mask, np.divide(img_norm_r, img_norm_g) > 1.185)

# Apply mask
new_img = np.zeros_like(mask, dtype=np.uint8)
new_img[mask] = 255

# Apply median filtering
new_img = weighted_median_filtering(new_img, 3)
```

Figure 20: маска сегментации.

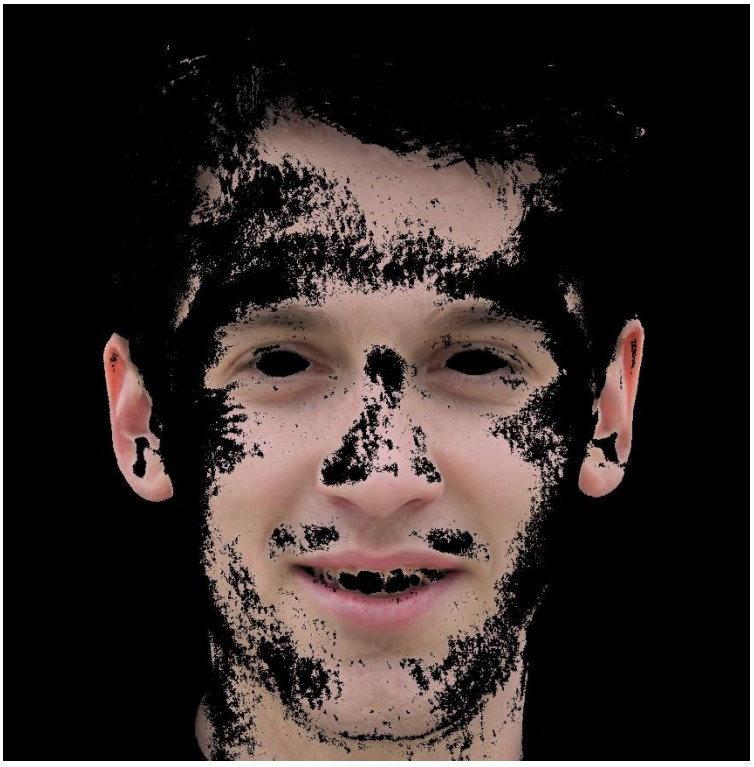

Figure 21: сегментированное изображение.

Результат содержит большую часть лица, также не захватил фоновые пиксели.

3. Сегментация 2

В данном пункте будем пытаться сегментировать изображение на участки с объектами одинаковой цветности.

Исходное изображение:

Figure 22: исходное изображение с объектами различных цветов.

3.1. Сегментация методом ближайших соседей.

Напомним идею алгоритма: разбиение цветного изображения на сегменты доминирующих цветов.

Listing 10. Сегментация методом ближайших соседей на Python.

```
# Convert to CIE Lab color space and then split into components
img_lab = cv.cvtColor(img, cv.COLOR_BGR2Lab)
img_lab_l, img_lab_a, img_lab_b = cv.split(img_lab)


# Set number of color segments
col_seg_num = 4


# Define areas
sample_areas = []
def mouse_handler(event, x, y, flags, param):
    if event != cv.EVENT_LBUTTONDBLCLK:
        return
```

```
    sample_areas.append((x, y))

cv.imshow("Image", img)
cv.setMouseCallback("Image", mouse_handler)
while len(sample_areas) < col_seg_num:  # Interactive loop
    cv.waitKey(20)
cv.setMouseCallback("Image", lambda *args : None)
cv.destroyAllWindows()


# Compute means in the circle areas of selected pixels
color_marks = []
color_marks_bgr = []
for pix in sample_areas:
    mask = np.zeros_like(img_lab_l)
    cv.circle(mask, pix, 10, 255, -1)  # Draw circle
    a = img_lab_a.mean(where = mask > 0)
    b = img_lab_b.mean(where = mask > 0)
    color_marks.append((a, b))
    color_marks_bgr.append(img[mask > 0, :].mean(axis=0))


# Calculate the distance between all pixels in initial image and previously selected
pixels
distance = []
for color in color_marks:
    distance.append(np.sqrt(np.power(img_lab_a - color[0], 2) + np.power(img_lab_b -
color[1], 2)))


# Estimate minimum of all pixels
distance_min = np.minimum.reduce(distance, axis=0)


# Find labels of all pixels
labels = np.zeros_like(img[:, :, 0], dtype=np.uint8)
for i in range(len(color_marks)):
    mask = distance_min == distance[i]
    labels[mask] = i


# Segment the source image to a set of segmented images
segmented_frames = []
for i in range(len(color_marks)):
    img_buf = np.zeros_like(img)
    mask = labels == i
    img_buf[mask] = img[mask]
    segmented_frames.append(img_buf)


# Plot distribution of image pixel colors in (a, b) coordinate system
img_plot = np.full((256, 256, 3), 255, dtype=np.uint8)
for i in range(len(color_marks)):
    img_buf = np.zeros_like(img)
    mask = labels == i
    img_plot[img_lab_a[mask], img_lab_b[mask], :] = color_marks_bgr[i]


# Show distribution plot
plt.imshow(cv.cvtColor(img_plot, cv.COLOR_BGR2RGB))
```

Были выбраны 4 сегментируемых класса:

Figure 23: образцы сегментируемых классов

После этого были определены цветовые метки для каждого из сегментов путем расчета среднего значения цветности в каждой выделенной области.

Средние значения в пространстве BGR:

$$1\ segment - 40.34384858,\quad 216.3785489,\quad 245.05678233;$$

$$2\ segment - 40.34069401,\quad 187.42271293,\quad 173;$$

$$3\ segment - 60.68138801,\quad 54.94006309,\quad 195.03785489;$$

$$4\ segment - 29.60567823,\quad 34.03470032,\quad 38.170347.$$

Затем использовался принцип ближайшей окрестности для классификации пикселей путем вычисления евклидовых метрик между пикселями и метками: чем меньше расстояние до метки, тем лучше пиксель соответствует данному сегменту.

Выведем график распределения пикселей.

Figure 24: график распределения пикселей.

Выведем сегментированные участки.

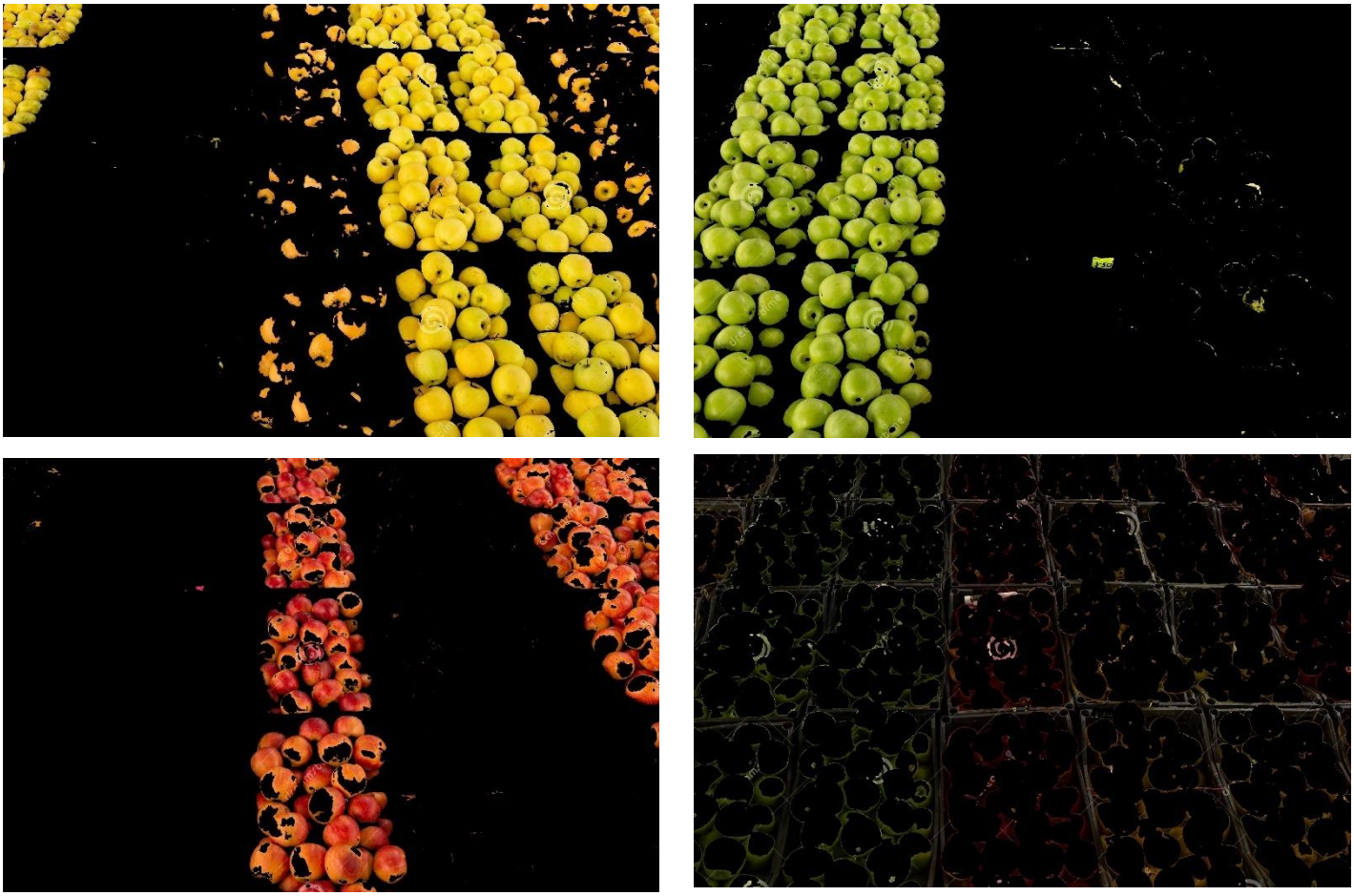

Figure 25: сегменты изображения.

Для построения общего изображения закрасим исходные классы их средним значением.

Listing 11. Построение общего изображения по сегментам со средними значениями на Python.

```
# Construct general photo
# Assign the segments with its mean intensities
img_gen = np.copy(img)
for i in range(col_seg_num):
    mask = labels == i
    img_gen[mask] = np.mean(img[mask], axis=0)
plt.imshow(cv.cvtColor(img_gen, cv.COLOR_BGR2RGB))
```

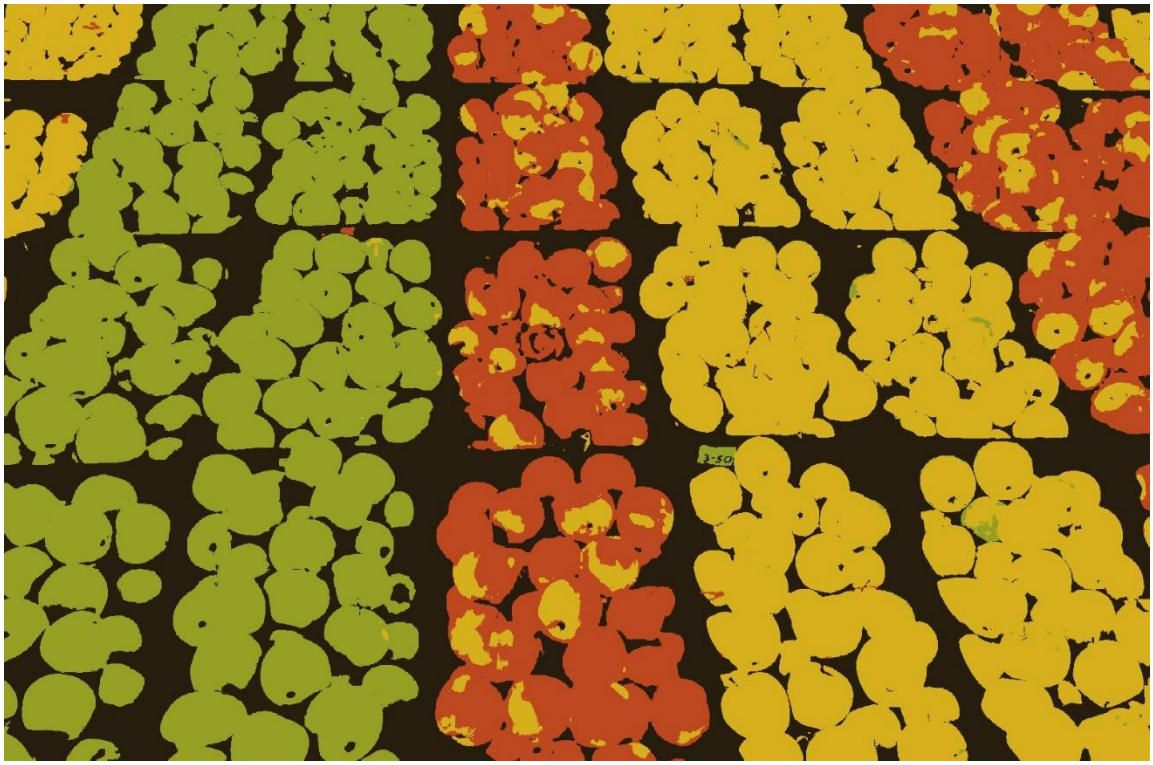

Figure 26: сегментация изображения методом ближайших соседей.

Результат сегментации довольно неплохой, все классы были отделены корректно.

3.2. Сегментация методом *k*-средних

Идея метода заключается в определении центров *k*-кластеров и отнесении к каждому кластеру пикселей, наиболее близко относящихся к этим центрам.

Listing 12. Сегментация методом k-средних в CIE Lab на Python.

```
# Convert to CIE Lab color space and then split into components
img_lab = cv.cvtColor(img, cv.COLOR_BGR2Lab)
img_lab_l, img_lab_a, img_lab_b = cv.split(img_lab)

# Set number of color segments
col_seg_num = 4

# Merge 'a' and 'b' components and then reshape it
ab = cv.merge([img_lab_a, img_lab_b])
ab = ab.reshape(-1, 2).astype(np.float32)

# Apply k-means clustering
# Stop criteria is defined as not more than 10 iterations of difference between
steps less than 1
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 10, 1.0)
#  Starting points are selected randomly (due to cv2.KMEANS_RANDOM_CENTERS is used)
and selection is done 10 times
```

```
ret, labels, centers = cv.kmeans(ab, col_seg_num, None, criteria, 10,
cv.KMEANS_RANDOM_CENTERS)
# Reshape labels back to an original image
labels = labels.reshape(img_lab_l.shape)

# Segment image to a set of images
segmented_frames = []
for i in range(col_seg_num):
    mask = labels == i
    img_buf = np.zeros_like(img)
    img_buf[mask] = img[mask, :]
    segmented_frames.append(img_buf)
```

Построим сразу общую картинку со всеми сегментами, со средними интенсивностями.

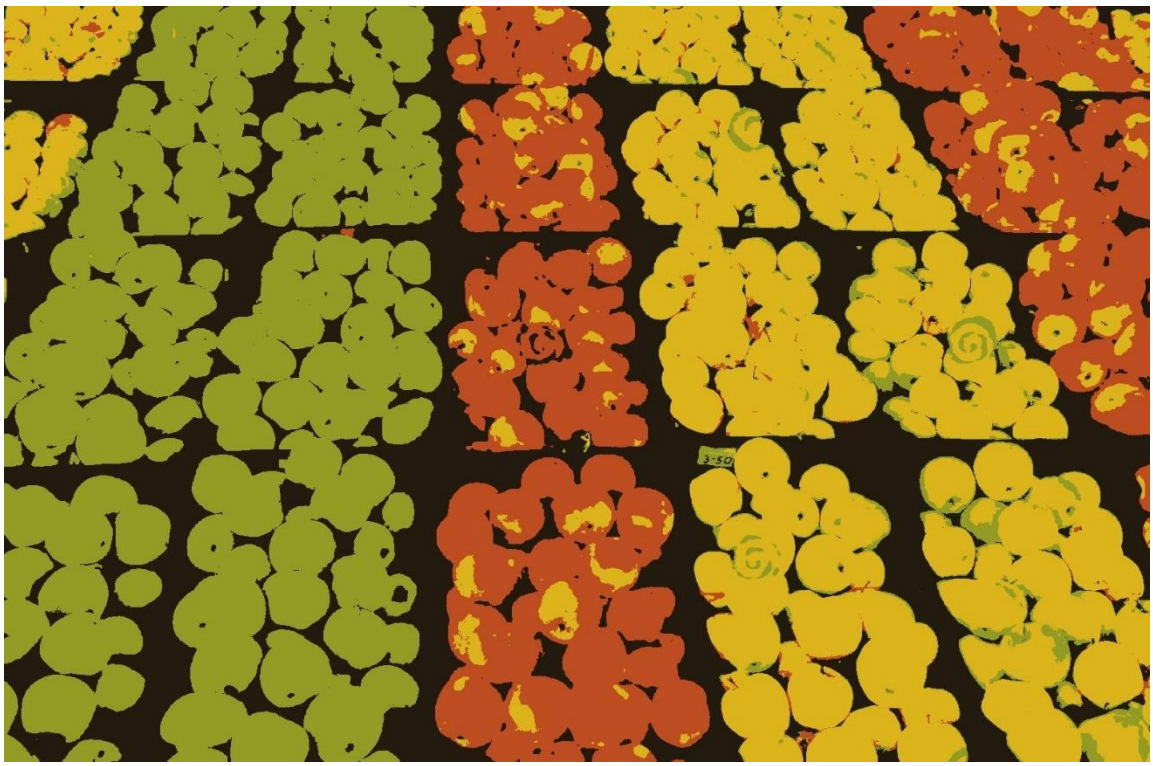

Figure 27: сегментация изображения методом k-средних.

В итоге два метода вернули схожие результаты работы.

Вообще говоря, данные методы совершенно разные и выбор того или иного должен осуществляться в зависимости от решаемой задачи.

4. Сегментация 3

В данном пункте будем проводить текстурную сегментацию.

Figure 28: исходное изображение.

На данном изображении присутствуют две текстуры: суша и вода.

Применим описанный выше алгоритм сегментации.

Listing 13. Текстурная сегментация на Python.

```
# Convert to grayscale
img_gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

# Calculate entropy and then normalize
ent = skimage.filters.rank.entropy(img_gray,
skimage.morphology.disk(13)).astype(np.float32)
ent_norm = (ent - np.min(ent)) / (np.max(ent) - np.min(ent))
```

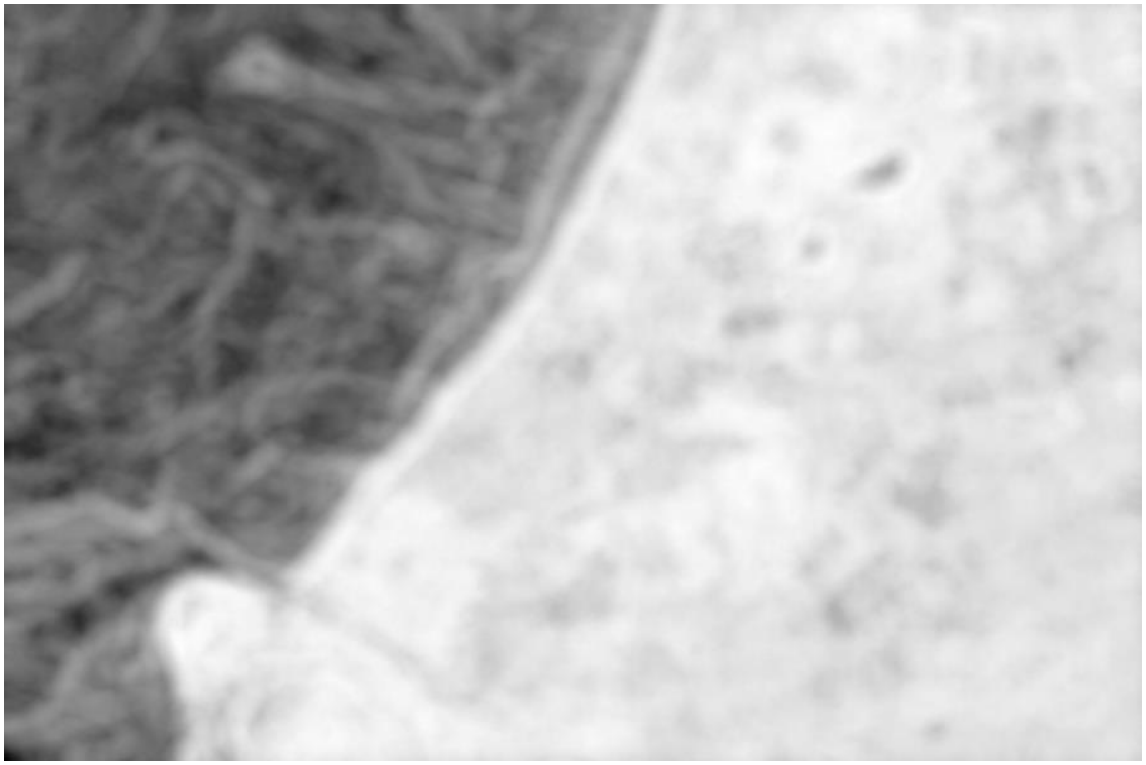

Figure 29: энтропия изображения.

```
# Binarize the resulting normalized array using the Otsu method
ret, bin_ent = cv.threshold(np.uint8(ent_norm * 255), 0, 255, cv.THRESH_OTSU)
```

Figure 30: бинаризация энтропии методом Оцу.

```
# 1. Remove small objects from binary image
img_fil = bwareaopen(bin_ent, 2000, conn=8)
# 2. Remove internal defects with closing operation
el = cv.getStructuringElement(cv.MORPH_ELLIPSE, (9, 9))
img_fil = cv.morphologyEx(img_fil, cv.MORPH_CLOSE, el)
# 3. Fill remaining large holes
img_fil = imfill(img_fil)
```

Figure 31: бинаризация энтропии после морфологических фильтров.

```
# Find shape contours and then draw them on a black background
contours, h = cv.findContours(img_fil, cv.RETR_TREE, cv.CHAIN_APPROX_NONE)
boundary = np.zeros_like(img_gray)
cv.drawContours(boundary, contours, -1, 255, 1)
```

Figure 32: граница раздела текстур.

```
# Apply the found border to the source image
segment_results = img_gray.copy()
segment_results[boundary != 0] = 255
```

Figure 33: граница текстур на исходном изображении.

```
# Excluding the water area from the source image and do the same steps to select the
remaining texture of the land
img_gray_2 = img_gray.copy()
img_gray_2[img_fil == 0] = 0
```

Figure 34: оставшаяся область суши.

Теперь проделаем те же шаги, чтобы сегментировать участок суши.

```
# Entropy
ent_2 = skimage.filters.rank.entropy(img_gray_2,
skimage.morphology.disk(17)).astype(np.float32)
ent_norm_2 = (ent_2 - ent_2.min()) / (ent_2.max() - ent_2.min())


# Binarization
ret_2, bin_ent_2 = cv.threshold(np.uint8(ent_norm_2 * 255), 0, 255, cv.THRESH_OTSU)


# Filtering
img_fil_2 = bwareaopen(bin_ent_2, 2000, conn=8)
img_fil_2 = cv.morphologyEx(img_fil_2, cv.MORPH_CLOSE, el)
img_fil_2 = imfill(img_fil_2)


# Select boundary
contours_2, h_2 = cv.findContours(img_fil_2, cv.RETR_TREE, cv.CHAIN_APPROX_NONE)
boundary_2 = np.zeros_like(img_gray_2)
cv.drawContours(boundary_2, contours_2, -1, 255, 1)


# Apply the found border to the source image
segment_results_2 = img_gray.copy()
segment_results_2[boundary_2 != 0] = 255


# Select textures of water and land basing on either of masks
texture_1 = img_gray.copy()
```

```
texture_1[img_fil_2 == 0] = 0
texture_2 = img_gray.copy()
texture_2[img_fil_2 != 0] = 0
```

Итого, получили сегментацию сначала относительно воды, затем относительно суши.

Сравним их.

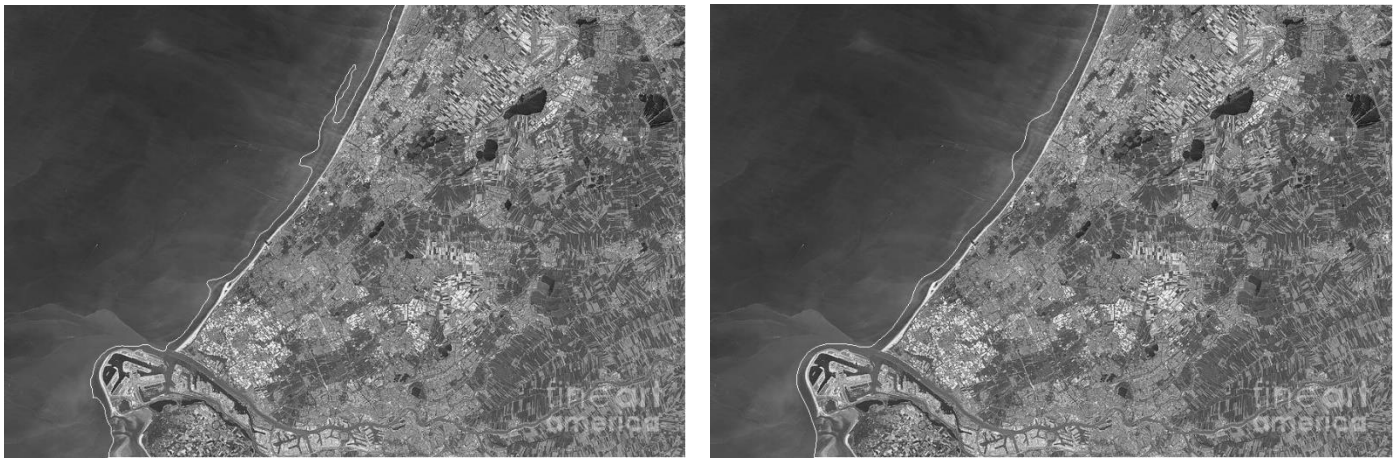

Figure 35: границы сегментаций (слева – относительно воды, справа – относительно суши).

Как результат сегментации будем использовать правую границу.

Figure 36: результат сегментации суши.

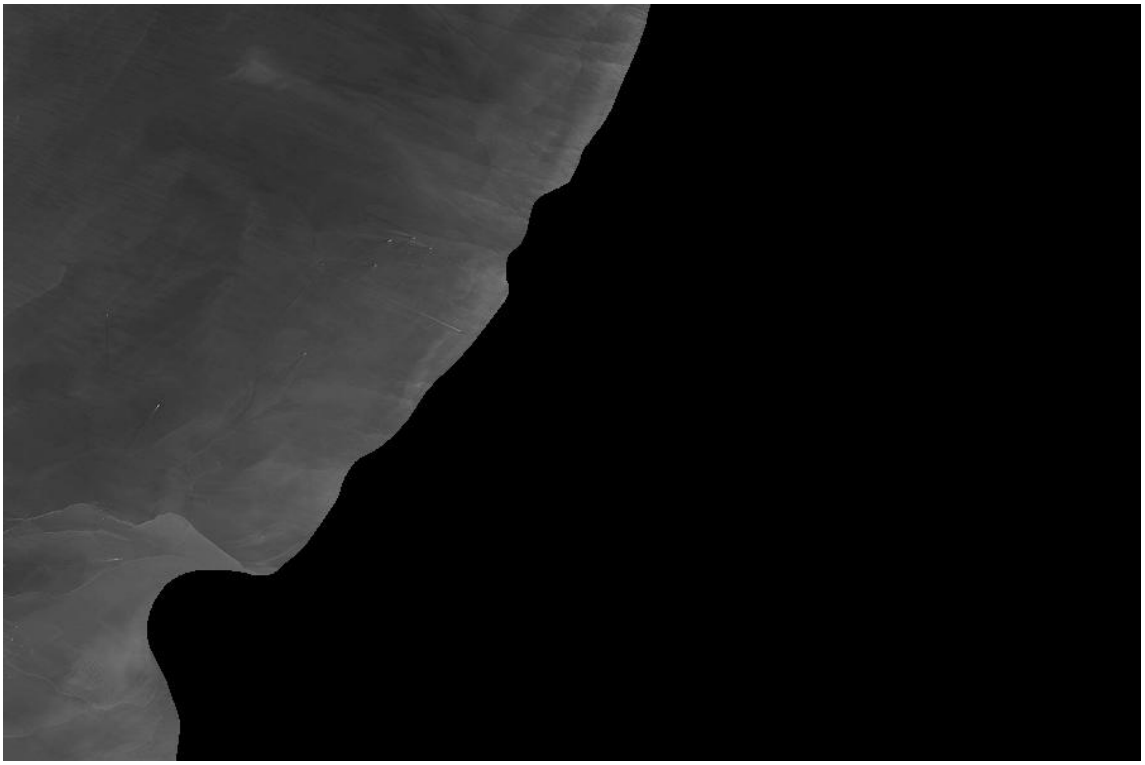

Figure 37: результат сегментации воды.

Теперь произведем оценку параметров текстур:

1. Среднее значение случайной величины.
2. Стандартное отклонение (квадратный корень из второго центрального момента).
3. R - относительная гладкость.
4. Характеристика симметрии гистограммы (третий центральный момент).
5. Однородность гистограммы.
6. Энтропия гистограммы.

Listing 14. Вычисление параметров текстур на Python.

```
# Calculate estimating parameters
# 1. Mean intensity
mean_1 = np.mean(img_gray[img_fil_2 != 0])
mean_2 = np.mean(img_gray[img_fil_2 == 0])
# 2. Standard deviation
std_1 = np.std(img_gray[img_fil_2 != 0])
std_2 = np.std(img_gray[img_fil_2 == 0])
# 3. R - relative smoothness
r_1 = 1 - 1 / (1 + np.power(std_1, 2))
r_2 = 1 - 1 / (1 + np.power(std_2, 2))
# 4. Third central moment
tcm_1 = scipy.stats.moment(img_gray[img_fil_2 != 0], moment=3, axis=0)
tcm_2 = scipy.stats.moment(img_gray[img_fil_2 == 0], moment=3, axis=0)
# 5. Histogram homogeneity
hist_1 = cv.calcHist([img_gray[img_fil_2 != 0]], [0], None, [256], (0, 256)) /
img_gray[img_fil_2 != 0].size
```

```
hist_2 = cv.calcHist([img_gray[img_fil_2 == 0]], [0], None, [256], (0, 256))/
img_gray[img_fil_2 == 0].size
hh_1 = np.sum(np.power(hist_1, 2))
hh_2 = np.sum(np.power(hist_2, 2))
# 6. Entropy
ent_1 = scipy.stats.entropy(img_gray[img_fil_2 != 0], base=2)
ent_2 = scipy.stats.entropy(img_gray[img_fil_2 == 0], base=2)
```

	Mean	*Std deviation*	*R*	*Third central moment*	*Uniformity measure*	*Entropy*
Texture 1 - Суша	121.433	40.865	0.999	43863.7	0.00733	18.325
Texture 2 - Вода	67.843	13.616	0.994	1919.503	0.025812	17.520

Выводы

Выполнена бинаризация чёрно-белого изображения по порогу, по двойному порогу, методом для нахождения порога Отсу, адаптивными Гауссовым и Медианным методами.

Обнаружено лицо на изображение с использованием трех различных методов на основе цвета кожи. Качество страдает.

Выполнена цветовая сегментация различных типов яблок и корзин в цветовом пространстве Lab по методу ближайших соседей и методом автоматического определения классов методом k-средних.

Выполнена текстурная сегментация фотографии на воду и сушу методом автоматической текстурной сегментации на основе яркостной сегментации энтропии изображения после применения морфологических фильтров. Также были вычислены параметры текстур и сведены в общую таблицу.