НИУ ИТМО

Факультет систем управления и робототехники

Лабораторная работа №4
по дисциплине «Техническое зрение»

Выполнили: Гридусов Д.Д

Преподаватель: Шаветов С.В.

Санкт-Петербург
2024 г.

Содержание

1 Введение

BREAKING NEWS: в методичке нет листингов для C++, и (представим что) меня забанили в документации OpenCV (за просрочку подавляющего большинства лабораторных работ по тех.зрению). Поэтому я не могу знать, что бинаризацию и сегментацию изображения можно сделать через встроенные методы библиотеки opencv, придется писать все самому (в том числе и свою реализацию K-means). Веселье.

Цель работы: освоение основных способов сегментации изображений на семантические области.

2 Бинаризация

Как и обещал, буду делать ручками. Возьмем картинку с малиной:

Листинг 1.1 Бинаризация

```
cv::Mat binary_threshold(cv::Mat img, int threshold)
{
    // converting threshold and image to double for easier binarization process
    double t = threshold / 255.0;
    auto depth = img.depth();
    if (depth == CV_8U)
    {
        img.convertTo(img, CV_32F, 1.0 / 255.0);
    }

    for (int i = 0; i < img.rows; i++)
    {
        float *row_ptr = img.ptr<float>(i);
        for (int j = 0; j < img.cols; j++)
        {
            if (row_ptr[j] <= t)
            {
                row_ptr[j] = 0;
            }
            else
            {
```

```
                row_ptr[j] = 1;
            }
        }
    }
    if (depth == CV_8U)
    {
        img.convertTo(img, CV_8U, 255);
    }

    return img;
}
```

Листинг 1.2 Бинаризация по двойному порогу

```
cv::Mat double_binary_threshold(cv::Mat img, int t1, int t2)
    {
        double threshold1 = t1 / 255.0, threshold2 = t2 / 255.0;
        auto depth = img.depth();
        if (depth == CV_8U)
        {
            img.convertTo(img, CV_32F, 1.0 / 255.0);
        }
        for (int i = 0; i < img.rows; i++)
        {
            float *row_ptr = img.ptr<float>(i);
            for (int j = 0; j < img.cols; j++)
            {
                if ((threshold1 < row_ptr[j] <= threshold2))
                {
                    row_ptr[j] = 0;
                }
                else
                {
                    row_ptr[j] = 1;
                }
            }
        }
        if (depth == CV_8U)
        {
            img.convertTo(img, CV_8U, 255);
        }
        return img;
    }
```

Попробуем задавать порог бинаризации по-умному, а не подбором.

Листинг 1.3 Вычисление порога через поиск минимальной и максимальной интенсивности

```
double calc_median_threshold(cv::Mat img)
{
    auto depth = img.depth();
    if (depth == CV_8U)
    {
        img.convertTo(img, CV_32F, 1.0 / 255.0);
    }
```

```
    double min, max;
    cv::minMaxLoc(img, &min, &max);
    double threshold = (max - min) / 2;
    threshold *= 255; // converting to [0, 255] for the implemented threshlding methods
    return threshold;
}
```

Листинг 1.4 Вычисление порога через поиск градиента

```
double calc_gradient_threshold(cv::Mat img)
{
    double t = 0;
    double grad = 0;

    auto depth = img.depth();
    if (depth == CV_8U)
    {
        img.convertTo(img, CV_32F, 1.0 / 255.0);
    }

    for (int i = 1; i < img.rows - 1; i++)
    {
        for (int j = 1; j < img.cols + 1; j++)
        {
            double x_gradient = std::fabs(img.at<float>(i, j + 1) - img.at<float>(i, j - 1));
            double y_gradient = std::fabs(img.at<float>(i + 1, j) - img.at<float>(i - 1, j));
            double g = std::max(x_gradient, y_gradient);
            t += g * img.at<float>(i, j);
            grad += g;
        }
    }
    t /= grad;
    t *= 255; // converting to [0, 255] for the implemented binariztion methods
    return t;
}
```

Самый умный (из представленных здесь) способ - метод Отсу. Как будет видно далее именно он даст наиболее качественный результат.

Листинг 1.5 Вычисление оптимального порога методом Отсу

```
int *calc_hist_classic(cv::Mat img)
{
    // calc hist, bins from [0, 256]
    int histSize = 256;
    int *hist = new int[histSize];
    for (int t = 0; t < histSize; t++)
    {
        hist[t] = 0;
    }
    for (int i = 0; i < img.rows; i++)
    {
        for (int j = 0; j < img.cols; j++)
        {
            uchar intensity = img.at<uchar>(i, j);
```

```
            hist[int(intensity)]++;
        }
    }
    return hist;
}

std::vector<double> calc_prob_dist(int *hist, int img_size)
{
    std::vector<double> prob_dist;
    int hist_size = sizeof(hist) / sizeof(int);
    for (int i = 0; i < 256; i++)
    {
        prob_dist.push_back(hist[i] * 1.0 / img_size);
    }

    return prob_dist;
}

double calc_prob(std::vector<double> prob_dist, int from, int to)
{
    double prob = 0;
    for (int i = from; i <= to; i++)
    {
        prob += prob_dist[i];
    }

    return prob;
}

double calc_mat_exp(std::vector<double> prob_dist, int from, int to)
{
    double mat_exp = 0;
    for (int i = from; i <= to; i++)
    {
        mat_exp += i * prob_dist[i];
    }

    return mat_exp;
}

double calc_threshold_otsu(cv::Mat img)
{
    double min, L, t = 0;
    int img_size = img.cols * img.rows;
    cv::minMaxLoc(img, &min, &L);

    int *hist = calc_hist_classic(img);
    std::vector<double> prob_dist = calc_prob_dist(hist, img_size);
    double threshold = 0, maxSigma = -1;
    double w1, w2, mu1, mu2, disp;
    for (int k = 1; k < L; k++)
    {
        w1 = 0, w2 = 0, mu1 = 0, mu2 = 0, disp = 0;
```

```
        w1 = calc_prob(prob_dist, 0, k);
        w2 = 1 - w1;
        mu1 = calc_mat_exp(prob_dist, 0, k) / w1;
        mu2 = calc_mat_exp(prob_dist, k + 1, L) / w2;

        disp = w1 * w2 * (mu1 - mu2) * (mu1 - mu2);

        if (disp > maxSigma)
        {
            maxSigma = disp;
            threshold = k;
        }
    }

    return threshold;
}
```

Результаты:

Рис. 1: Бинаризация с одним порогом

Рис. 2: Бинаризация с двойным порогом

Рис. 3: Бинаризация по среднему значению интенсивности

Рис. 4: Бинаризация с порогом, найденным через градиент

Рис. 5: Бинаризация с помошью метода Отсу

3 Сегментация по цвету кожи

Самый, пожалуй, простой этап работы. Подход достаточно простой, и сработал не очень хорошо, пропустив часть кожи на лбу. Хотя и сильно лишнего алгоритм не выбрал.

Листинг 2.1 Сегментация по цвету кожи

```
cv::Mat skin_tone_segmentation(cv::Mat img){
    for (int row = 0; row < img.rows; row++){
        cv::Vec3b* ptr = img.ptr<cv::Vec3b>(row);
        for (int col = 0; col < img.cols; col++){
            uchar B = ptr[col][0], G = ptr[col][1], R = ptr[col][2];
            double r = 1.0 * R/(R+G+B), g = 1.0 * G/(R+G+B), b = 1.0 *B/(R+G+B);
            if (((r / g) > 1.185) && ((r * b)/(std::pow((r + g + b), 2)) > 0.107) && ((r *
                g)/(std::pow((r + g + b), 2)) > 0.112)){
                ptr[col] = cv::Vec3b(0, 255, 0);
            }
        }
    }
    return img;
}
```

Рис. 6: Исходное изображение

Рис. 7: Результат сегментации

4 Сегментация k-means

Мое любимое.
Для начала нам нужна структура, которая будет хранить информацию о пикселе изображения (можно реально использовать struct, но я написал честный класс, отличия все-таки минимальные). Пиксель будет характеризоваться id-шником кластера, к которому он принадлежи, и вектором вещественных чисел, состоящих из характеристик о пикселе, полученных следующим образом: переводим изображение в пространство CIE LAB, и для каждого пикселя изображение создаем объект класса Pixel, передавая в конструктор в том числе img_lab.at<cv::Vec3b>(i, j)
Листинг 4.1 Класс Pixel

```
class Pixel
{
private:
    int id;
    int clusterID;
    std::vector<double> lab;

public:
    Pixel(){}
    Pixel(int id, cv::Vec3b Lab){
        this->id = id;
        this->clusterID = 0;
        for (int i = 0; i < 3; i++){
            lab.push_back(Lab[i] * 1.0/ 255.0);
        }
    }

    int getId() { return id; }

    int getClusterID() { return clusterID; }

    double getVal(int pos) { return lab[pos]; }

    void setVal(int pos, double val){
        this->lab[pos] = val;
    }

    void setCluster(int clusterID) { this->clusterID = clusterID; }
};
```

Теперь напишем класс, описывающий кластер. Полями будут id-кластера, текущий центроид в кластере и общее количество пикселей, принадлежащих к данному кластеру.
Листинг 4.2 Класс Cluster

```
class Cluster
{
private:
    int clusterID;
    Pixel centroid;
    std::vector<Pixel> pixels;

public:
    Cluster(int clusterID, Pixel centroid){
```

```
        this->clusterID = clusterID;
        this->centroid = centroid;
    }
    Pixel getCentroid(){
        return this->centroid;
    }

    int getClusterID(){
        return clusterID;
    }

    Pixel getPixel(int pos) { return pixels[pos]; }

    void addPixel(Pixel p){
        p.setCluster(clusterID);
        pixels.push_back(p);
    }

    bool removePixel(int pixelID){
        for (int i = 0; i < pixels.size(); i++){
            if (pixels[i].getId() == pixelID){
                pixels.erase(pixels.begin() + i);
                return true;
            }
        }

        return false;
    }

    int getSize() { return pixels.size(); }

    void clearCluster(){pixels.clear();}

    void moveCentroid(int pos, double val){
        this->centroid.setVal(pos, val);
    }
};
```

Подготовительная работа для реализации алгоритма k-means сделана, переходим к написанию центрального класса этой части работы.

Листинг 4.3 Класс Kmeans

```
class Kmeans
{
private:
    int K, iters, total_points; // amount of clusters, maximum iterations, amount of pixels
    std::vector<Cluster> clusters;
    void clearClusters();
    int getNearestClusterId(Pixel pixel);
public:
    Kmeans(int K, int iters, int total_points);
    std::vector<int> fit(std::vector<Pixel> &all_pixels);
};
```

Что в классе из содержательного: есть гиперпараметры - количество кластеров разбиения, максимальное количество итераций (чтобы алгоритм не засиживался при очень маленьких, но ненулевых сдвигах центроид внутри кластеров и останавливал работу), и общее количество пикселей на изображении.
Также есть важный метод **getNearestClusterId**: он будет для каждого пикселя считать расстояние (используем евклидову метрику) до центроида каждого кластера и возвращать id ближайщего кластера. Этот id будет присваиваться пикселю в качестве значения поля clusterID.
И ключевой метод **fit** - в нем заключена основная работа:

- случайным образом инициализируются центры кластеров (это кстати неплохо так влияет на погрешность, которую дает k-means, и есть даже более продвинутые версии алгоритма, исправляющие эту проблему)
- все пиксели приписываются к ближайшему для них кластеру
- центры кластеров пересчитываются как среднее арифметическое признаков векторов (а в нашем случае векторов *lab* пикселей), вошедших в данный кластер
- когда центры кластеров перестают сдвигаться (или при достижении максимального количества итераций) алгоритм прекращает работу и возвращает вектор из лэйблов для каждого пикселя, на основе которого можно собрать сегментированное изображение

Листинг 4.4 Реализация алгоритма K-means

```
Kmeans::Kmeans(int K, int iters, int total_points){
    this->K = K;
    this->iters = iters;
    this->total_points = total_points;
}

void Kmeans::clearClusters(){
    for (Cluster cluster : this->clusters){
        cluster.clearCluster();
    }
}

int Kmeans::getNearestClusterId(Pixel pixel)
{
    // for each centroid measure euclidean distance(pixel, centroid) and return the nearest
        centroid
    double min_dist = DBL_MAX;
    int nearestCLusterID = 0;
    for (int i = 0; i < K; i++){
        Pixel current_centroid = clusters[i].getCentroid();
        double dist = std::sqrt( std::pow(current_centroid.getVal(0) - pixel.getVal(0), 2) +
            std::pow(current_centroid.getVal(1) - pixel.getVal(1), 2)
            +std::pow(current_centroid.getVal(2) - pixel.getVal(2), 2) );

        if (dist < min_dist){
            min_dist = dist;
            nearestCLusterID = clusters[i].getClusterID();
        }
    }
```

```
    return nearestCLusterID;
}

std::vector<int> Kmeans::fit(std::vector<Pixel> &all_pixels){
    std::vector<int> labels;

    // initializing clusters as random pixels of the image
    std::vector<int> used_pointIds; // vector with indexes of pixels that were assigned as
        clusters
    for (int i = 1; i <= K; i++)
    {
        while (true)
        {

            int index = std::rand() % this->total_points; //random index from 0 to
                total_points-1
            // if index is not in used_pointIds - it is a new cluster
            if (std::find(used_pointIds.begin(), used_pointIds.end(), index) ==
                used_pointIds.end())
            {
                used_pointIds.push_back(index);
                all_pixels[index].setCluster(i);
                Cluster cluster(i, all_pixels[index]);
                clusters.push_back(cluster);
                break;
            }
        }
    }
    std::cout << "Clusters initialized, K = " << clusters.size() << std::endl
              << std::endl;

    std::cout << "Starting clasterization..." << std::endl;
    int iter = 1;
    while (true)
    {
        std::cout << "Iteration - " << iter << "/" << iters << std::endl;
        bool done = true;

// Add all points to their nearest cluster
#pragma omp parallel for reduction(&& : done) num_threads(16)
        for (int i = 0; i < total_points; i++)
        {
            int currentClusterId = all_pixels[i].getClusterID();
            int nearestClusterId = getNearestClusterId(all_pixels[i]);

            if (currentClusterId != nearestClusterId)
            {
                all_pixels[i].setCluster(nearestClusterId);
                done = false;
            }
        }

        // clear all existing clusters
```

```
        clearClusters();

        // reassign points to their new clusters
        for (int i = 0; i < total_points; i++)
        {
            // cluster index is ID-1
            clusters[all_pixels[i].getClusterID() - 1].addPixel(all_pixels[i]);
        }

        // Recalculating the center of each cluster
        for (int i = 0; i < K; i++)
        {
            int ClusterSize = clusters[i].getSize();

            for (int j = 0; j < 3; j++)
            {
                double sum = 0.0;
                if (ClusterSize > 0)
                {
#pragma omp parallel for reduction(+ : sum) num_threads(16)
                    for (int p = 0; p < ClusterSize; p++)
                    {
                        sum += clusters[i].getPixel(p).getVal(j);
                    }
                    clusters[i].moveCentroid(j, sum * 1.0 / ClusterSize);
                }
            }
        }

        if (done || iter >= iters)
        {
            std::cout << "Clustering completed in iteration : " << iter << std::endl
                      << std::endl;
            break;
        }
        iter++;
    }

    for (int i = 0; i < total_points; i++)
    {
        labels.push_back(all_pixels[i].getClusterID());
    }

    return labels;

}
```

Метод, которым я собирал сегментированное изображение - это метод за который лично мне стыдно, но доработать (хоть это и вообще несложно) пока нет ресурсов - лабу надо сдать. И, если оправдываться по полной - у меня нет вычислительной мощности, чтобы сегментировать больше чем на 4 кластера. В общем, внизу листинг текущей сборки сегментированного изображения, метод поправлю позже (только не бейте).

Листинг 4.5 Предобработка изображения и получение сегментированной картинки

```
    std::vector<Pixel> prepareImg(cv::Mat img){
    // converting img to CIE LAB color space
    // and flattening to a vector of Pixel objects
    cv::Mat img_Lab;
    cv::cvtColor(img, img_Lab, cv::COLOR_BGR2Lab);
    std::vector<Pixel> imgData;
    int id_counter = 0;
    for (int y = 0; y < img.rows; y++) {
        for (int x = 0; x < img.cols; x++) {
            Pixel pixel{id_counter, img_Lab.at<cv::Vec3b>(y, x)};
            id_counter++;
            imgData.push_back(pixel);
        }
    }

    return imgData;
}

int main(){

    std::string ORIGINAL_DIR = "/home/den/CV_labs/Lab4/img/original/";
    std::string RES_DIR = "/home/den/CV_labs/Lab4/img/outputs/segmentation3/";
    cv::Mat img = cv::imread(ORIGINAL_DIR + "flower.jpeg");

    std::vector<Pixel> imgData = prepareImg(img);
    int K = 4, iters = 100, pixels_amount = img.rows * img.cols;


    Kmeans model{K, iters, pixels_amount};
    std::vector<int> labels = model.fit(imgData);

    int height = img.rows;
    int width = img.cols;

    cv::Mat outputImage(img.rows, img.cols, CV_8UC3, cv::Scalar(0, 0, 0));
    for (int i = 0; i < labels.size(); i++) {
        int spatialRow = i / img.cols;
        int spatialCol = i % img.cols;
        switch (labels[i] % 3) {
        case 0:
            outputImage.at<cv::Vec3b>(spatialRow, spatialCol)[0] = 255;
            outputImage.at<cv::Vec3b>(spatialRow, spatialCol)[1] = 0;
            outputImage.at<cv::Vec3b>(spatialRow, spatialCol)[2] = 0;
            break;
        case 1:
            outputImage.at<cv::Vec3b>(spatialRow, spatialCol)[0] = 0;
            outputImage.at<cv::Vec3b>(spatialRow, spatialCol)[1] = 255;
            outputImage.at<cv::Vec3b>(spatialRow, spatialCol)[2] = 0;
            break;
        case 2:
            outputImage.at<cv::Vec3b>(spatialRow, spatialCol)[0] = 0;
            outputImage.at<cv::Vec3b>(spatialRow, spatialCol)[1] = 0;
```

```
            outputImage.at<cv::Vec3b>(spatialRow, spatialCol)[2] = 255;
            break;
        default:
            outputImage.at<cv::Vec3b>(spatialRow, spatialCol)[0] = 255;
            outputImage.at<cv::Vec3b>(spatialRow, spatialCol)[1] = 255;
            outputImage.at<cv::Vec3b>(spatialRow, spatialCol)[2] = 255;
            break;
        }
    }

    cv::imwrite(RES_DIR + "flower_4_segments.jpeg", outputImage);

    return 0;
}
```