**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра ТВ**

**отчЁт**

**по лабораторной работе № 1**

**по дисциплине «Интеллектуальный анализ данных»**

**Тема: Кластеризация k-средних**

|  |  |  |
| --- | --- | --- |
| Студенты гр. 9105 |  | Шаривзянов Д. Р. |
|  |  |  |
| Преподаватель |  | Поздеев А. А. |

Санкт-Петербург

2024

КЛАСТЕРИЗАЦИЯ K-СРЕДНИХ

**Цель работы:** моделирование алгоритма кластеризации k-средних.

1. Код программы

#ifndef CLUSTERING\_H

#define CLUSTERING\_H

#include <iostream>

#include <vector>

#include <opencv2\highgui\highgui.hpp>

#include <opencv2\imgproc\imgproc.hpp>

#include "logger.hpp"

using namespace std;

using namespace cv;

/\*\*

 \* @brief Calculates the Euclidean distance between two points

 \*

 \* @param point1 The first point

 \* @param point2 The second point

 \* @return The Euclidean distance between the two points

 \*/

double calcEuclidDist(const vector<double>& point1, const vector<double>& point2) {

    double distance = 0.0;

    // Calculate the squared difference between the corresponding coordinates of the two points

    for (size\_t i = 0; i < point1.size(); ++i)

        distance += pow(point1[i] - point2[i], 2);

    // Calculate the square root of the sum of the squared differences

    return sqrt(distance);

}

/\*\*

 \* @brief Initializes the centers of the clusters

 \*

 \* @param k The number of clusters

 \* @param dimensions The number of dimensions in the data

 \* @return A vector of vectors, where each inner vector represents a cluster center

 \*/

vector<vector<double>> initRandCenters(int k, int dimensions) {

    vector<vector<double>> centers;

    // Initialize k cluster centers with random values between 0 and 1

    for (int i = 0; i < k; ++i) {

        vector<double> center;

        for (int j = 0; j < dimensions; ++j)

            center.push\_back((rand() % 256) / 255.0); // For color values between 0 and 1

        centers.push\_back(center);

    }

    return centers;

}

/\*\*

 \* @brief Prints the centers of the clusters to the logger

 \*

 \* @param centers The centers of the clusters

 \*/

void printCenters(const vector<vector<double>>& centers) {

    for (size\_t i = 0; i < centers.size(); ++i) {

        // Create an output stream to build the string

        std::ostringstream center\_stream;

        // Add the center of the cluster to the stream

        center\_stream << "Center of cluster " << (i + 1) << ": [";

        // center\_stream << "             ├Center of cluster " << (i + 1) << ": [";

        // Add the coordinates of the center to the stream

        for (size\_t j = 0; j < centers[i].size(); ++j) {

            center\_stream << centers[i][j];

            // Add a comma if it's not the last coordinate

            if (j < centers[i].size() - 1)

                center\_stream << ", ";

        }

        // Close the bracket

        center\_stream << "]";

        // Log the string

        logger.info(center\_stream.str());

    }

}

/\*\*

 \* @brief Runs the K-means clustering algorithm on the given image

 \*

 \* @param src The input image

 \* @param dst The output image with the clustered colors

 \* @param k The number of clusters to form

 \* @param max\_iterations The maximum number of iterations to run the algorithm

 \*/

void segmentImageKMeans(Mat& src, Mat& dst, int k, int max\_iterations) {

    dst = Mat::zeros(src.size(), src.type());

    int channels = src.channels();

    // Convert the image to a vector of points

    vector<vector<double>> points;

    for (int i = 0; i < src.rows; ++i)

        for (int j = 0; j < src.cols; ++j) {

            vector<double> pixel;

            for (int c = 0; c < channels; ++c)

                pixel.push\_back(static\_cast<double>(src.at<Vec3b>(i, j)[c]) / 255.0);

            points.push\_back(pixel);

        }

    // Initialize the cluster centers

    vector<vector<double>> centers = initRandCenters(k, channels);

    // Run the K-means algorithm

    vector<int> labels(points.size());

    int iteration = 0;

    bool converged = false;

    while (!converged && iteration < max\_iterations) {

        vector<vector<double>> new\_centers(k, vector<double>(channels, 0.0));

        vector<int> counts(k, 0);

        // Assign each point to a cluster

        for (size\_t i = 0; i < points.size(); ++i) {

            double min\_distance = numeric\_limits<double>::max();

            int closest\_center = 0;

            for (int j = 0; j < k; ++j) {

                double distance = calcEuclidDist(points[i], centers[j]);

                if (distance < min\_distance) {

                    min\_distance = distance;

                    closest\_center = j;

                }

            }

            for (int c = 0; c < channels; ++c)

                new\_centers[closest\_center][c] += points[i][c];

            counts[closest\_center]++;

            labels[i] = closest\_center;

        }

        // Recalculate the cluster centers

        for (int i = 0; i < k; ++i)

            if (counts[i] != 0)

                for (int c = 0; c < channels; ++c)

                    new\_centers[i][c] /= counts[i];

        // Check for convergence

        if (new\_centers == centers)

            converged = true;

        else

            centers = new\_centers;

        // Print the cluster centers after each iteration

        logger.info("Iteration {}:", (iteration + 1));

        // logger.info("Iteration {}: ┐", iteration + 1);

        printCenters(centers);

        iteration++;

    }

    // Create a new image with the colors of the cluster centers

    for (size\_t i = 0; i < points.size(); ++i) {

        int x = i / src.cols;

        int y = i % src.cols;

        for (int c = 0; c < channels; ++c)

            dst.at<Vec3b>(x, y)[c] = static\_cast<uchar>(centers[labels[i]][c] \* 255);

    }

}

/\*\*

 \* @brief Runs the K-means clustering algorithm on the given image

 \*

 \* @param img\_bgr The input image in BGR format

 \* @param num\_clusters The number of clusters to form

 \* @param max\_iterations The maximum number of iterations to run the algorithm

 \*/

void lab1KMeans(Mat& img\_bgr) {

    // Initialize the logger

    init\_logger();

    logger.info("Lab 1: K-Means Clustering");

    // Display the original image

    imshow("image bgr", img\_bgr);

    waitKey(1);

    cout << "Enter the number of clusters: ";

    int num\_clusters;

    cin >> num\_clusters;

    if (num\_clusters <= 0) {

        logger.error("Invalid number of clusters: {}", num\_clusters);

        return;

    }

    // Run the K-means algorithm on the BGR image

    logger.info("Clustering BGR image with k = {}", num\_clusters);

    Mat img\_k\_bgr;

    segmentImageKMeans(img\_bgr, img\_k\_bgr, num\_clusters, 10);

    // Display the clustered image

    imshow("image clustered bgr", img\_k\_bgr);

    // spdlog::shutdown();

    // Wait for the user to press a key

    waitKey();

}

#endif CLUSTERING\_H

1. Исходное изображение



Рис 1. Исходное цветное изображение.

1. Кластеризованное изображение.



Рис 2. Маска из 6 цветов, соответствующих сформированным кластерам.

**Вывод:** в ходе работы был реализован алгоритм кластеризации K-средних. На сходимость центроидов необходимо порядка 10-30 итераций алгоритма, причём количество этих итераций сильно зависит от количества центроидов. Эта зависимость нелинейна.