Министерство образования Республики Беларусь

Учреждение образования «Полоцкий государственный университет имени Евфросинии Полоцкой»

Факультет информационных технологий

Кафедра технологий программирования

**Отчёт по лабораторной работе № 1 по курсу «Распределенные вычисления»**

«Реализация алгоритма кластеризации методом К-средних на CPU»

ВЫПОЛНИЛ студент группы 21-ИТ-1

Шиковец Е.А.

ПРОВЕРИЛ преподаватель

Адамовский Е.Р.

Полоцк, 2024 г.

Цель: Ознакомиться с алгоритмом кластеризации методом К-средних и реализовать данный алгоритм на CPU.

**ОПИСАНИЕ ПРОДЕЛАННОЙ РАБОТЫ**

Задание:

1. Реализовать консольное приложение (на языке программирования C++) для выполнения алгоритма K-средних согласно представленным вариантам в таблице.
2. Запустить реализованное консольное приложение, измерить встроенными средствами время выполнения алгоритма и его отдельных итераций на CPU.
3. Представить результаты измерений в графическом виде.

|  |  |  |
| --- | --- | --- |
| **Вариант** | **Количество элементов и атрибутов элемента** | **Количество кластеров** |
| 10 | 100 000 × 7 | 8 |

Листинг 1.1 – KMeans.cpp

#include <iostream>

#include <vector>

#include <cmath>

using namespace std;

// Класс для реализации алгоритма KMeans

class KMeans {

private:

vector<vector<double>> data; // Входные данные для

кластеризации

int k; // Количество кластеров

vector<vector<vector<double>>> clusters; // Результаты

кластеризации

public:

// Конструктор класса

KMeans(vector<vector<double>> inputData, int k) :

data(inputData), k(k) {}

// Метод для получения результирующих кластеров

vector<vector<vector<double>>> getClusters() {

return clusters;

}

// Метод для запуска алгоритма кластеризации

void run() {

// Инициализация центроидов случайными точками из данных

vector<vector<double>> centroids = initializeCentroids();

// Итеративный процесс перераспределения точек между

кластерами

for (int iter = 0; iter < 100; iter++) { // Максимальное

количество итераций

// Создание новых кластеров

vector<vector<vector<double>>> newClusters(k);

// Перераспределение точек между кластерами на основе

ближайшего центроида

for (const auto& point : data) {

// Находим индекс ближайшего центроида для данной

точки

int closestCentroidIndex =

findClosestCentroidIndex(point, centroids);

// Добавляем точку в соответствующий кластер

newClusters[closestCentroidIndex].push\_back(point);

}

// Пересчитываем центроиды

centroids = updateCentroids(newClusters);

// Если новые кластеры не отличаются от предыдущих,

завершаем процесс

if (areClustersEqual(clusters, newClusters)) {

clusters = newClusters;

break;

}

// Обновляем текущие кластеры

clusters = newClusters;

}

}

private:

// Метод для инициализации центроидов

vector<vector<double>> initializeCentroids() {

vector<vector<double>> centroids(k);

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

// Выбираем случайный индекс точки из входных данных

int randomIndex = rand() % data.size();

// Присваиваем центроиду координаты выбранной случайной

точки

centroids[i] = data[randomIndex];

}

return centroids;

}

// Метод для нахождения индекса ближайшего центроида для данной

точки

int findClosestCentroidIndex(const vector<double>& point, const

vector<vector<double>>& centroids) {

double minDistance = numeric\_limits<double>::max(); //

Инициализируем минимальное расстояние очень большим числом

int closestIndex = 0;

// Проходим по всем центроидам и находим ближайший к точке

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

double distance = calculateDistance(point,

centroids[i]); // Вычисляем расстояние между точкой и текущим

центроидом

// Если найдено более короткое расстояние, обновляем

ближайший центроид

if (distance < minDistance) {

minDistance = distance;

closestIndex = i;

}

}

return closestIndex;

}

// Метод для вычисления евклидова расстояния между двумя

точками

double calculateDistance(const vector<double>& point1, const

vector<double>& point2) {

double sum = 0;

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

// Вычисляем сумму квадратов разностей координат

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

}

// Корень из суммы квадратов - евклидово расстояние

return sqrt(sum);

}

// Метод для пересчета центроидов на основе новых кластеров

vector<vector<double>> updateCentroids(const

vector<vector<vector<double>>>& clusters) {

vector<vector<double>> centroids(k);

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

if (!clusters[i].empty()) {

// Если кластер не пуст, вычисляем его центроид

centroids[i] = calculateCentroid(clusters[i]);

}

else {

// Если кластер пуст, оставляем центроид пустым

centroids[i] = vector<double>(data[0].size());

}

}

return centroids;

}

// Метод для вычисления центроида для заданного кластера

vector<double> calculateCentroid(const vector<vector<double>>&

cluster) {

size\_t dimension = cluster[0].size();

vector<double> centroid(dimension, 0);

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

double sum = 0;

for (const auto& point : cluster) {

sum += point[i];

}

// Среднее значение координат точек кластера

centroid[i] = sum / cluster.size();

}

return centroid;

}

// Метод для проверки равенства двух наборов кластеров

bool areClustersEqual(const vector<vector<vector<double>>>&

clusters1, const vector<vector<vector<double>>>& clusters2) {

if (clusters1.size() != clusters2.size())

return false;

for (size\_t i = 0; i < clusters1.size(); i++) {

if (clusters1[i].size() != clusters2[i].size())

return false;

for (size\_t j = 0; j < clusters1[i].size(); j++) {

// Сравниваем точки внутри кластеров

if (clusters1[i][j] != clusters2[i][j])

return false;

}

}

return true;

}

};

Листинг 1.2 – lab1.cpp

#include <iostream>

#include <vector>

#include <cmath>

#include <string>

#include <chrono> // Для измерения времени выполнения

#include "matplotlibcpp.h" // Библиотека для визуализации

#include "KMeans.cpp" // Подключение файла с реализацией KMeans

using namespace std;

namespace plt = matplotlibcpp;

// Функция для генерации тестовых данных

vector<vector<double>> generateData(int rowCount, int columnCount)

{

vector<vector<double>> data(rowCount,

vector<double>(columnCount));

for (int i = 0; i < rowCount; i++) {

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

data[i][j] = static\_cast<double>(rand()) / RAND\_MAX \*

100; // Генерация случайных значений от 0 до 100

}

}

return data;

}

// Функция для визуализации кластеров

void plotClusters(const vector<vector<vector<double>>>& clusters) {

vector<string> colors = { "r", "g", "b", "c", "m", "y", "k" };

// Цвета для кластеров

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

vector<double> xs, ys;

for (const auto& point : clusters[i]) {

xs.push\_back(point[0]); // Координата x точки

ys.push\_back(point[1]); // Координата y точки

}

plt::plot(xs, ys, "o"); // Построение точек кластера на

графике

plt::named\_plot("Cluster " + to\_string(i + 1), xs, ys,

colors[i % colors.size()] + "o"); // Добавление подписи и цвета к

кластеру

}

plt::legend(); // Добавление легенды

plt::show(); // Отображение графика

}

int main() {

// Данные по варианту

vector<vector<double>> data = generateData(100000, 7); //

Генерация 100 000 точек в 7-мерном пространстве

int k = 8; // Количество кластеров

auto start = chrono::steady\_clock::now(); // Засекаем время

начала выполнения

KMeans kmeans(data, k); // Создание объекта KMeans

// Запуск алгоритма K-средних

kmeans.run();

auto end = chrono::steady\_clock::now(); // Засекаем время

окончания выполнения

// Вывод времени выполнения в консоль

auto duration = chrono::duration\_cast<chrono::milliseconds>(end

- start);

cout << "Time taken: " << duration.count() << " milliseconds"

<< endl;

// Получение результирующих кластеров

auto clusters = kmeans.getClusters();

// Отображение точек на графике

plotClusters(clusters);

return 0;

}

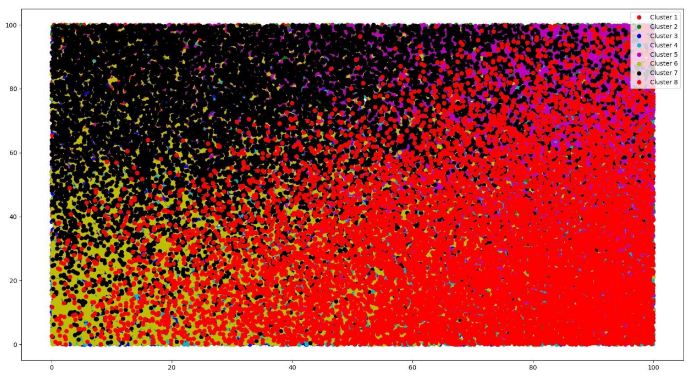


Рисунок 1.1 – Результаты в графическом представлени

**ВЫВОД**

В результате выполнения данной лабораторной работы я знакомился с алгоритмом кластеризации методом К-средних и реализовал данный алгоритм