ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – София

Проект

по

„Програмни езици“

на тема

„Клъстерен анализ. K-meaning алгоритъм“

Изготвили: Николай Тониев Чобанов, Винсент Стефанов Кръстанов
факултетни № 121214189, 121214197

Проверил: доц. д-р инж. Аделина Алексиева-Петрова

Съдържание:

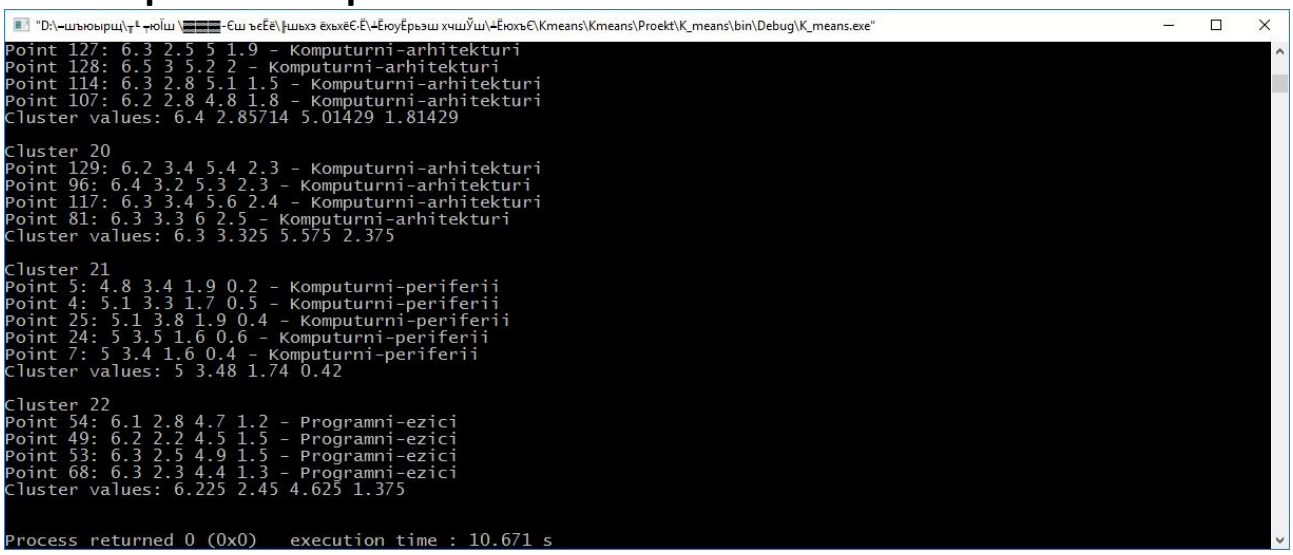
Николай Чобанов
47 гр., фак. № 121214189

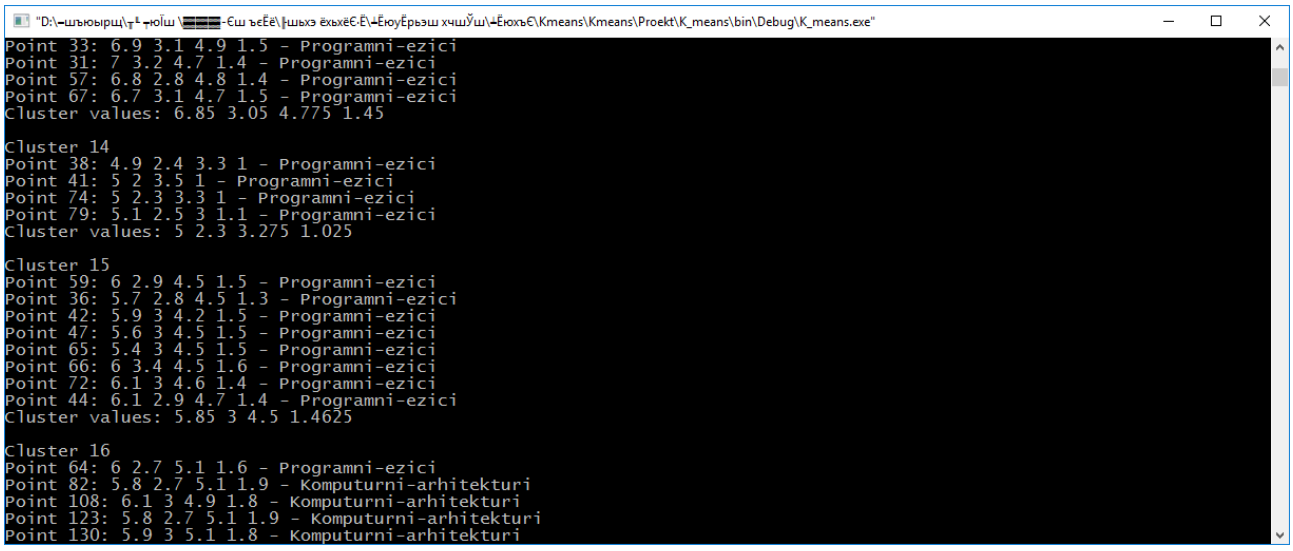
Винсент Кръстанов
44гр., фак. № 121214197

1. Цел на проекта.

Идеята да направим този проект е да се запознаем с това що е клъстерен анализ, да придобием знания относно основните алгоритми за клъстерен анализ, да намерим, този алгоритъм, който е най-интересен за нас и да направим собствена имплементация, която да следва основните принципи на избрания алгоритъм. Произволните данни които се подават към програмата се анализират от нея и се разпределят в групи. Елементите разпределени в дадени групи трябва да бъдат със сходни характеристики, кои да са следените характеристики, се дефинират чрез допълнителни настройки. За да може действията изпълнявани от програмата да бъдат по прозрачни за потребителя е разработено конзолно приложение, което чрез принтиращи функции държи потребителя информиран относно протичащите процеси.

2. За проекта накратко.





3. Определение за клъстерен анализ (Clustering).

Клъстерният анализ е класификация, чиято цел е да се оформят естествени групи въз основа на много признаци едновременно. Целта при клъстерния анализ е n на брой обекта да се групират в k (k>1) на брой групи, наречени клъстери, като се използват p (p>0) на брой признаци (променливи). Самият клъстерен анализ е събирателно понятие и съдържа много на брой различни клъстеризационни процедури. Едно важно деление на клъстеризационните процедури е в зависимост от това дали се задава предварително броят на клъстерите или не. При предварително зададен брой на клъстерите се използва метода K-Meaning (K-means) Cluster (клъстерен анализ на K-средните). А когато броят на клъстерите не е предварително определен си служим с Hierarchical Cluster Analysis или т. н. йерархичен клъстерен анализ. Голямото разнообразие на клъстеризационни процедури се поражда още от използваната метрика между различните обекти. По-известни метрики са: Евклидовото разстояние, Манхатъново разстояние, разстояние на Чебишев и др. Разнообразието се поражда и от използваните правила за създаване на клъстерите – за тяхното обединяване или разединяване.

Друго популярно определение за клъстерен анализ е че клъстерният анализът е многомерен статистически метод, предназначен за разпределяне на множество обекти едновременно по няколко критерия в сравнително малко на брой и относително хомогенни групи, наречени клъстери. Обектите в даден клъстер си приличат по между си (имат сходни значения по набора от критериите) и се различават съществено от обектите в другите клъстери. Като критерии за класифициране на единиците в клъстери се използват определен брой от класификационни променливи.

При използване на този метод се минава през следните етапи:

1) определят се класификационните критерии
2) избира се измерителя на дистанцията между единиците
3) избор на метод на клъстеризация може да бъде йерархична и нейерархична
   - Метод на най-близкият съсед
   - Метод на най-отдалечения съсед
   - Метод на междугрупово свързване
   - Метод на вътрешно групово свързване
4) определяне броя на клъстерите

При предварително зададен брой на клъстерите един много популярен метод е K-Means Cluster (Клъстерен анализ на К-средните). Когато броя на клъстерите не е предварително определен, се използва така наречените йерархични клъстеризационни процедури (Hierarchical Cluster).

4. Определение за метод K-means.

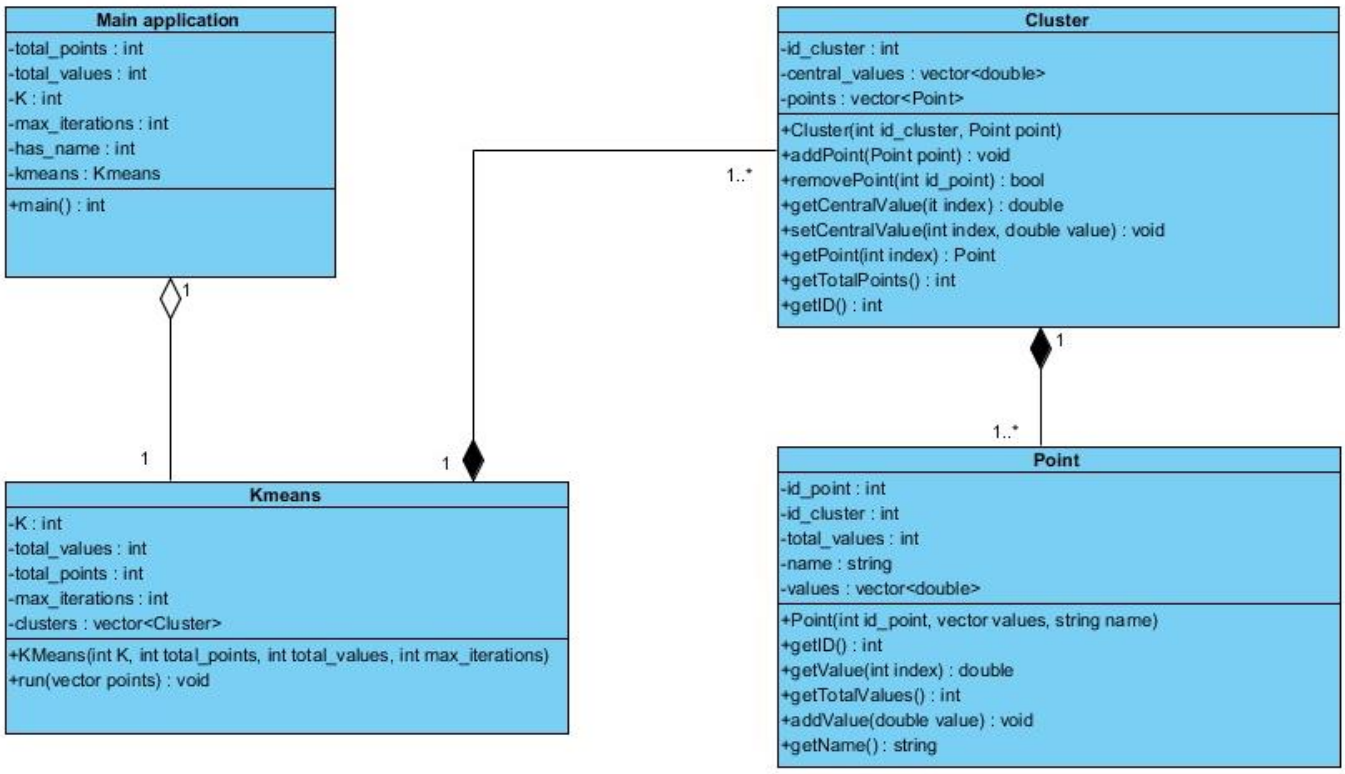
K-means е най-популярния метод за клъстеризация. Изобретен е през 1950-та година от математика Hugo Dyonizy Steinhaus. Особена популярност добива след работата на J. MacQueen. По аналогия с метода на основните компоненти, центъра на клъстъра се нарича главна точка, а самият метод се нарича метод на главната точка и е включена в общата теория на основните обекти на осигуряване на възможно най-доброто сближаване на данни.

5. Основна идея и логика на нашата имплементация на метода. UML диаграма отразяваща програмния код.

За коректна работа на алгоритъма и представяния на възможностите и функциите на обектно ориентираното програмиране сме използвали композиционни и агрегатни връзки между класове на C++. Класовете са сепарирани на смислов принцип. Всеки елемент, изпълняващ близки по значение и логически сходни функции е обособен в отделен клас. Член-променливите са обособени в Private секции, като тези, които е нужно да бъдат достъпвани от други класове имат get (гетъри) и set (сетъри) методи в Public секцията.

Причините за това, релациите между класовете да са само от тип композиция и агрегация са следните: Алгоритъмът е съставен от елементи, които задължително при всяка итерация си взаимодействат. Това е предпоставка за силна връзка между тях. Асоциативни връзки не биха били удачни, тъй като биха могли да развалят цялостта и коректната работа на алгоритъма.

Файловата структура на сорс кода е смислово обособена на хедър (hpp) файлове и имплементационни (cpp) файлове. Целта на това е да се раздели чистата имплементация от декларацията на функциите, член-променливите, конструктора.



6. Обяснение на конкретни функции, детайли и тънки моменти.

Представяне на данните:

С цел удобно и разбираемо представяне на работата на алгоритъма е избрано конзолно принтиране на резултатите чрез използване на вградените в библиотеката <iostream> функции cout, cin.

Примерните данни са готово конфигурирани на случаен принцип, за да демонстрират работата на алгоритъма във всякакви ситуации. Размерът на данните е така избран, че да бъде подходящ за работа с K-meaning.

K-meaning алгоритъма изисква предварителна настройка според изискванията на ползвателя. Тази настройка се прави в началото преди конфигурирането на данните, за да бъде постигнат желания от потребителя резултат. Въвеждането на данните става по горе описания начин – конзолно.

Алгоритъмът има способността сам да успява чрез анализ на данните да ги клъстеризира в отделни клъстери според вида им, както и нещата по които си приличат и се различават. Това го прави коренно различен от алгоротми за разделяне на информация, алгоритми за сортиране на данни и алгоритми за отсяване на определен тип информация.

Функция run(vector<Point> & points) на класа Kmeans:

Това е основната смислова функция, движеща алгоритъма. В нея се използват всички елементи, нужни за работата

на алгоритъма - масива от клъстерите, както и масива от принадлежащите точки на всеки от клъстерите. Във

функцията се извършват всички циклични операции, като чрез изпълнение на нужния брой итерации се определят
реалните центроиди на клъстерите.

Функция getIDNearestCenter(Point point) на класа Kmeans:
Това е функцията, вътрешна за класа Kmeans. Тя е спомагателна фукция, която чрез изпълнение на нужен брой
итерации връща ID стойността на най-близкия центроид на точката, подадена като аргумент на функцията.

Функция addPoint(Point point) на класа Cluster:
Тази функция се използва за добавяне на желаните от потребителя точки, както и за добавяне на
всяка нужна нова точка при наличието на нови стойности при поява на нова итерация в цикъл от алгоритъма.

Функция removePoint(int id_point) на класа Cluster:
Тази функция се използва за изтриване на точка от вектора с точки на даден клъстер. Тя е извиквана при всяка
нова итерация от цикъл на алгоритъма, като след като се доставят нови стойности за точките, то се добавят нови
точки с тях, а старите се изтриват от масива.

7. Прототипи на функциите.

A) Cluster.h

```
#ifndef CLUSTER_H_INCLUDED
#define CLUSTER_H_INCLUDED

#include <vector>

#include"Point.h"

using namespace std;

class Cluster
{
private:

   int id_cluster;
   vector<double> central_values;
   vector<Point> points;

public:

   Cluster(int id_cluster, Point point);

   void addPoint(Point point);

   bool removePoint(int id_point);

   double getCentralValue(int index);

   void setCentralValue(int index, double value);

   Point getPoint(int index);
```

```
        int getTotalPoints();

        int getID();
```

B) Kmeans.h

```
#ifndef KMEANS_H_INCLUDED
#define KMEANS_H_INCLUDED
#include <iostream>
#include <vector>
#include <math.h>
#include <algorithm>
#include"Cluster.h"
using namespace std;
class KMeans
{
private:

        int K; // number of clusters
        int total_values, total_points,
max_iterations;
        vector<Cluster> clusters;
```

C) Point.h

```
#ifndef POINT_H_INCLUDED
#define POINT_H_INCLUDED

#include <vector>
#include <stdlib.h>
#include <string>

using namespace std;

class Point
{
private:

        int id_point, id_cluster;
        vector<double> values;
        int total_values;
        string name;

public:
```

```
};

#endif
```

```
        // return ID of nearest center
(uses euclidean distance)
        int getIDNearestCenter(Point
point);

public:

        KMeans(int K, int total_points,
int total_values, int max_iterations);

        void run(vector<Point> & points);

};

#endif
```

```
        Point(int id_point,
vector<double>& values, string name =
"");

        int getID();

        void setCluster(int id_cluster);

        int getCluster();

        double getValue(int index);

        int getTotalValues();

        void addValue(double value);

        string getName();
};
```

8. Имплементация на функциите.

A) main.cpp

```
#include <iostream>
#include <vector>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <algorithm>

#include"Point.h"
#include"Cluster.h"
#include"Kmeans.h"

using namespace std;

int main(int argc, char *argv[])
{
	srand (time(NULL));

	int total_points, total_values, K, max_iterations, has_name;

	cin >> total_points >> total_values >> K >> max_iterations >> has_name;

	vector<Point> points;
	string point_name;

	for(int i = 0; i < total_points; i++)
	{
		vector<double> values;
```

B) Cluster.cpp

```
#include"Cluster.h"

Cluster::Cluster(int id_cluster, Point point)
	{
		this->id_cluster = id_cluster;
#endif
```

```
		for(int j = 0; j < total_values; j++)
		{
			double value;
			cin >> value;

		values.push_back(value);
		}

		if(has_name)
		{
			cin >> point_name;
			Point p(i, values, point_name);
			points.push_back(p);
		}
		else
		{
			Point p(i, values);
			points.push_back(p);
		}
	}

	KMeans kmeans(K, total_points, total_values, max_iterations);
	kmeans.run(points);

	return 0;
}
```

```
		int total_values = point.getTotalValues();

		for(int i = 0; i < total_values; i++)
```

```
        central_values.push_back(point.
getValue(i));

                points.push_back(point);
        }

        void        Cluster::addPoint(Point
point)
        {
                points.push_back(point);
        }

        bool      Cluster::removePoint(int
id_point)
        {
                int        total_points        =
points.size();

               for(int i = 0; i < total_points;
i++)
                {
                        if(points[i].getID()
== id_point)
                        {

        points.erase(points.begin() + i);
                                return true;
                        }
                }
                return false;
```

C) Kmeans.cpp

```
#include"Kmeans.h"

KMeans::KMeans(int        K,        int
total_points, int total_values, int
max_iterations)
        {
                this->K = K;
                this->total_points        =
total_points;
                this->total_values        =
total_values;
                this->max_iterations      =
max_iterations;
        }
        int
KMeans::getIDNearestCenter(Point
point)
        {
              double sum = 0.0, min_dist;
        }

        double
Cluster::getCentralValue(int index)
        {
                return
central_values[index];
        }

        void  Cluster::setCentralValue(int
index, double value)
        {
                central_values[index]     =
value;
        }

        Point Cluster::getPoint(int index)
        {
                return points[index];
        }

        int Cluster::getTotalPoints()
        {
                return points.size();
        }

        int Cluster::getID()
        {
                return id_cluster;
        }
```

```
            int id_cluster_center = 0;

            for(int i = 0; i < total_values; i++)
            {
                sum += pow(clusters[0].getCentralValue(i) - point.getValue(i), 2.0);
            }

            min_dist = sqrt(sum);

            for(int i = 1; i < K; i++)
            {
                double dist;
                sum = 0.0;

                for(int j = 0; j < total_values; j++)
                {
                    sum += pow(clusters[i].getCentralValue(j) - point.getValue(j), 2.0);
                }

                dist = sqrt(sum);

                if(dist < min_dist)
                {
                    min_dist = dist;
        id_cluster_center = i;
                }
            }

            return id_cluster_center;
        }
```

```
        void KMeans::run(vector<Point> & points)
        {
            if(K > total_points)
                return;

            vector<int> prohibited_indexes;

            // izbirane na K otchetlivi stoinosti za centura na klusterite
            for(int i = 0; i < K; i++)
            {
                while(true)
                {
                    int index_point = rand() % total_points;


        if(find(prohibited_indexes.begin(), prohibited_indexes.end(),

        index_point) == prohibited_indexes.end())
                    {

        prohibited_indexes.push_back(index_point);

        points[index_point].setCluster(i);
                        Cluster cluster(i, points[index_point]);

        clusters.push_back(cluster);
                        break;
                    }
                }
            }

            int iter = 1;

            while(true)
```

```
            {
                bool done = true;

                // asocirane na vsqka pochka blizka do centura
                for(int i = 0; i < total_points; i++)
                {
                    int id_old_cluster = points[i].getCluster();
                    int id_nearest_center = getIDNearestCenter(points[i]);


        if(id_old_cluster != id_nearest_center)
                    {

        if(id_old_cluster != -1)

        clusters[id_old_cluster].removePoint(points[i].getID());



        points[i].setCluster(id_nearest_center);

        clusters[id_nearest_center].addPoint(points[i]);
                        done = false;
                    }
                }

                // rekalkuclirane na centura na vseki kluster
                for(int i = 0; i < K; i++)
                {
                    for(int j = 0; j < total_values; j++)
                    {
                        int total_points_cluster = clusters[i].getTotalPoints();
                        double sum = 0.0;


        if(total_points_cluster > 0)
                        {

        for(int p = 0; p < total_points_cluster; p++)

        sum += clusters[i].getPoint(p).getValue(j);

        clusters[i].setCentralValue(j, sum / total_points_cluster);
                        }
                    }
                }

                if(done == true || iter >= max_iterations)
                {
                    cout << "Break in iteration " << iter << "\n\n";
                    break;
                }

                iter++;
            }

            // pokazvane na elementite na klustera
            for(int i = 0; i < K; i++)
            {
                int total_points_cluster = clusters[i].getTotalPoints();
```

```
                cout << "Cluster nomer: " << clusters[i].getID() + 1 << endl;
                for(int j = 0; j < total_points_cluster; j++)
                {
                        cout << "Tochka: " << clusters[i].getPoint(j).getID() + 1 << ": ";
                        for(int p = 0; p < total_values; p++)
                                cout << clusters[i].getPoint(j).getValue(p) << " ";

                        string point_name = clusters[i].getPoint(j).getName();
```

D) Point.cpp

```
#include"Point.h"

Point::Point(int id_point, vector<double>& values, string name)
{
        this->id_point = id_point;
        total_values = values.size();

        for(int i = 0; i < total_values; i++)

this->values.push_back(values[i]);

        this->name = name;
        id_cluster = -1;
}

int Point::getID()
{
        return id_point;
}
```

```
        if(point_name != "")
                cout << "- " << point_name;

                cout << endl;
        }

        cout << "Cluster stoinosti: ";

        for(int j = 0; j < total_values; j++)
                cout << clusters[i].getCentralValue(j) << " ";

                cout << "\n\n";
        }
}


void Point::setCluster(int id_cluster)
{
        this->id_cluster = id_cluster;
}

int Point::getCluster()
{
        return id_cluster;
}

double Point::getValue(int index)
{
        return values[index];
}

int Point::getTotalValues()
{
        return total_values;
}
```

```
void Point::addValue(double value)
{
	values.push_back(value);
}

string Point::getName()
{
	return name;
}
```

9. Заключение.

K-meaning алгоритъмът е известен с широката си употреба в клъстеризирането поради всеобхватната му работа с различен вид данни. Освен това алгоритъмът може да бъде имплементиран на различни програмни езици и да има еднаква функционалност със сходни ресурс параметри. Освен това алготитъмът силно се възползва от основните принципи на обектно-ориентираното програмиране – то му дава ясна смислова разграничимост на елементите, както и ефективно взаимодействие между тях. Чрез примерите и изготвеното конзолно приложение се показват базови действия, които да демонстрират клъстеризирането като цяло, както и клъстеризирането с допълнителни изисквания от страна на потребителя, вградени конкретно в K-meaning алгоритъма за клъстерен анализ.

10. Източници на информация.

https://ru.wikipedia.org/wiki/K-means
http://www.fmi-plovdiv.org/evlm/DBbg/database/studentbook/SPSS_CA_2.pdf
https://en.wikipedia.org/wiki/Cluster_analysis