K-均值聚类算法（k-means）的C++实现

K-均值聚类算法（k-means ）的C++实现

k-均值聚类算法(k-means)主要用于解决 **距离空间** 上目标点集的自动分类问题。

本篇博文目的在于

1. 阐述 **k-means** 聚类算法的数学模型
2. 利用C++改写并封装K-Means算法，接口数据向MATLAB中kmeans函数看齐。

k-means算法的数学原理

k-means聚类算法的问题描述如下：

假设我们的研究对象可以一一映射到 *dimension* 维欧式空间中的一个点 **P** 。现在我们需要把空间中相近的目标点圈起来，看作是一类对象，一共需要分为 k 类。

先来介绍k-means算法涉及到的数据结构：

```
struct kmeans
{
    double clusterCenter[k][dimension];
    int clusterAssignment[targetCount];
}
```

clusterCenter[m] 代表第m类对象的类中心，为 *dimension* 维的向量，共有k个这样的聚类中心。
clusterAssignment[m] 则记录了目标集合中第m个目标点所属的类。

k-means算法是一个迭代算法，迭代的数学公式如下：

$$d(P_m, clusterCenter_i(index)) = min\{d(P_m, clusterCenter_i(j)), j = 1, 2, ..., k\}$$
$$clusterAssignment_{i+1}(m) = index$$
$$clusterCenter_{i+1}(n) = mean(\{P_j | clusterAssignment_{i+1}(j) = n\})$$

其中 d 为定义在目标所在空间上的距离函数。i为迭代次数，m,n,j为下标变量。

用一般语言对迭代过程进行描述：对于任一目标点 **P**，设距离其最近的聚类中心为类 **C** 中心，把目标点 **P** 归于类别 **C** 。完成所有目标点的归类后，将同一类的目标点求其质心，作为该类的新的聚类中心。重复上述操作，直至所有目标点的分类不再发生改变。

k-means算法的c++实现

对于k-means算法的c++代码已经比较常见，本文附上的C++代码则更关注于将k-means算法进行封装，提高算法模块的独立性，便于再次开发，向MATLAB中的kmeans函数看齐。

本节所提供的c++代码主要在CSDN博主—— **忆之独秀** 的博文代码基础上进行修改与封装。原文请见链接：
忆之独秀的CSDN博客

引用博主的伪代码对算法实现核心进行描述：

```
创建k个点作为起始质心(经常是随机选择)
当任意一个点的簇分配结果发生改变时
    对数据集中的每个数据点
        对每个质心
            计算质心与数据点之间的距离
        将数据点分配到距其最近的簇
    对每一个簇，计算簇中所有点的均值并将均值作为质心
```

以下给出封装后的KMEANS算法类中常用的几个变量与函数的解释。

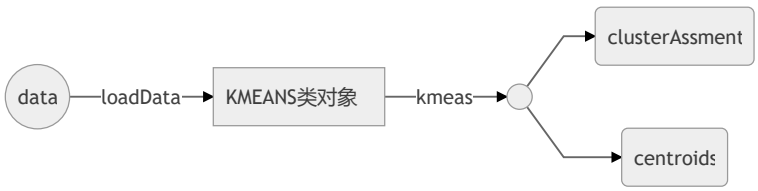
```
template<typename T>
    //cluster centers
    vector< vector<T> > centroids;
    //mark which cluster the data belong to
    vector<tNode>  clusterAssment;

    //construct function
    KMEANS::KMEANS(void);
    //load data into dataSet
    void KMEANS::loadData(vector< vector<T> > data);
    //running the kmeans algorithm
    void KMEANS::kmeans(int clusterCount);
```

成员变量 *centroids* 为聚类中心，规模为 *k * dimension* ， *centroids[m]* 代表第m类对象点的聚类中心。
成员变量 *clusterAssment* 则记录了各个目标点所属类。 *clusterAssment[m].minIndex* 为第m目标点所属类别，
clusterAssment[m].minDist 为第m目标点与其所属类中心的距离。

KMEANS 类是k-means聚类算法本身，构造函数中不需要给定任何参数。
loadData 函数将待聚类数据 *data* 送入算法类中。其中需要说明： *data* 为 *vector< vector < T > >* 型数据。类似于matlab-kmeans函数，
data[m] 为一个 *vector< T >* 型数据，代表着一个目标点的坐标。可以认为，data为 *particalCount * dimension* 的矩阵。
kmeans 函数则是运行kmeans聚类算法，运行结果存入 *centroids* 与 *clusterAssment* 中。

KMEANS模块的流如下图所示：



对应的函数调用：

```
KMEANS<double> YKmeans;
YKmeans.loadData(data);
YKmeans.kmeans();
//use YKmeans.Assment and YKmeans.centroids
```

以下附上KMEANS类的头文件与对应的例程代码。

测试范例：main.cpp

```
#include "YKmeans.h"
#include <vector>
#include <iostream>
#include <math.h>
using namespace std;
```

```
const int pointsCount = 9;
const int clusterCount = 2;

int main()
{
    //构造待聚类数据集
    vector< vector<double> > data;
    vector<double> points[pointsCount];
    for (int i = 0; i < pointsCount; i++)
    {
        points[i].push_back(i);
        points[i].push_back(i*i);
        points[i].push_back(sqrt(i));
        data.push_back(points[i]);
    }

    //构建聚类算法
    KMEANS<double> kmeans;
    //数据加载入算法
    kmeans.loadData(data);
    //运行k均值聚类算法
    kmeans.kmeans(clusterCount);

    //输出聚类后各点所属类情况
    for (int i = 0; i < pointsCount; i++)
        cout << kmeans.clusterAssment[i].minIndex << endl;
    //输出类中心
    cout << endl << endl;
    for (int i = 0; i < clusterCount; i++)
    {
        for (int j = 0; j < 3; j++)
            cout << kmeans.centroids[i][j] << ',' << '\t' ;
        cout << endl;
    }

    return(0);
}
```

KMEANS类对应的头文件：YKmeans.h

```
#ifndef _YKMEANS_H_
#define _YKMEANS_H_

#include <cstdlib>  //for rand()
#include <vector>   //for vector<>
#include <time.h>   //for srand
#include <limits.h> //for INT_MIN INT_MAX
using namespace std;

template<typename T>
class KMEANS
{
protected:
    //colLen:the dimension of vector;rowLen:the number of vectors
    int colLen, rowLen;
    //count to be clustered
    int k;
    //mark the min and max value of a array
    typedef struct MinMax
    {
        T Min;
```

```
        T Max;
        MinMax(T min, T max) :Min(min), Max(max) {}
    }tMinMax;
    //distance function
    //reload this function if necessary
    double (*distEclud)(vector<T> &v1, vector<T> &v2);

    //get the min and max value in idx-dimension of dataSet
    tMinMax getMinMax(int idx)
    {
        T min, max;
        dataSet[0].at(idx) > dataSet[1].at(idx) ? (max = dataSet[0].at(idx), min = dataSet[1].at(idx)) : (max = dat
aSet[1].at(idx), min = dataSet[0].at(idx));

        for (int i = 2; i < rowLen; i++)
        {
            if (dataSet[i].at(idx) < min)   min = dataSet[i].at(idx);
            else if (dataSet[i].at(idx) > max) max = dataSet[i].at(idx);
            else continue;
        }

        tMinMax tminmax(min, max);
        return tminmax;
    }
    //generate clusterCount centers randomly
    void randCent(int clusterCount)
    {
        this->k = clusterCount;
        //init centroids
        centroids.clear();
        vector<T> vec(colLen, 0);
        for (int i = 0; i < k; i++)
            centroids.push_back(vec);

        //set values by column
        srand(time(NULL));
        for (int j = 0; j < colLen; j++)
        {
            tMinMax tminmax = getMinMax(j);
            T rangeIdx = tminmax.Max - tminmax.Min;
            for (int i = 0; i < k; i++)
            {
                /* generate float data between 0 and 1 */
                centroids[i].at(j) = tminmax.Min + rangeIdx * (rand() / (double)RAND_MAX);
            }
        }
    }
    //default distance function ,defined as dis = (x-y)'*(x-y)
    static double defaultDistEclud(vector<T> &v1, vector<T> &v2)
    {
        double sum = 0;
        int size = v1.size();
        for (int i = 0; i < size; i++)
        {
            sum += (v1[i] - v2[i])*(v1[i] - v2[i]);
        }
        return sum;
    }

public:
    typedef struct Node
    {
```

```
    int minIndex; //the index of each node
    double minDist;
    Node(int idx, double dist) :minIndex(idx), minDist(dist) {}
}tNode;

KMEANS(void)
{
    k = 0;
    colLen = 0;
    rowLen = 0;
    distEclud = defaultDistEclud;
}
~KMEANS(void){}
//data to be clustered
vector< vector<T> > dataSet;
//cluster centers
vector< vector<T> > centroids;
//mark which cluster the data belong to
vector<tNode>  clusterAssment;

//Load data into dataSet
void loadData(vector< vector<T> > data)
{
    this->dataSet = data; //kmeans do not change the original data;
    this->rowLen = data.capacity();
    this->colLen = data.at(0).capacity();
}
//running the kmeans algorithm
void kmeans(int clusterCount)
{
    this->k = clusterCount;

    //initial clusterAssment
    this->clusterAssment.clear();
    tNode node(-1, -1);
    for (int i = 0; i < rowLen; i++)
        clusterAssment.push_back(node);

    //initial cluster center
    this->randCent(clusterCount);

    bool clusterChanged = true;
    //the termination condition can also be the loops less than some number such as 1000
    while (clusterChanged)
    {
        clusterChanged = false;
        for (int i = 0; i < rowLen; i++)
        {
            int minIndex = -1;
            double minDist = INT_MAX;
            for (int j = 0; j < k; j++)
            {
                double distJI = distEclud(centroids[j], dataSet[i]);
                if (distJI < minDist)
                {
                    minDist = distJI;
                    minIndex = j;
                }
            }
            if (clusterAssment[i].minIndex != minIndex)
            {
                clusterChanged = true;
```

```
                clusterAssment[i].minIndex = minIndex;
                clusterAssment[i].minDist = minDist;
            }
        }

        //step two : update the centroids
        for (int cent = 0; cent < k; cent++)
        {
            vector<T> vec(colLen, 0);
            int cnt = 0;
            for (int i = 0; i < rowLen; i++)
            {
                if (clusterAssment[i].minIndex == cent)
                {
                    ++cnt;
                    //sum of two vectors
                    for (int j = 0; j < colLen; j++)
                    {
                        vec[j] += dataSet[i].at(j);
                    }
                }
            }

            //mean of the vector and update the centroids[cent]
            for (int i = 0; i < colLen; i++)
            {
                if (cnt != 0)   vec[i] /= cnt;
                centroids[cent].at(i) = vec[i];
            }
        }
    }
};

#endif // _YKMEANS_H_
```

后注

在本文给出的KMEANS类中给出了一个默认的距离函数:

$$defaultDis(\vec{x},\vec{y}) = (\vec{x}-\vec{y}) \bullet (\vec{x}-\vec{y}) = ||\vec{x}-\vec{y}||^2$$

需要注意，定义在空间中的距离函数不一定是经典的二范数的形式。距离函数可以由用户给定，本文附上的KMEANS类代码也留出了用户指定距离函数的接口。用户只需要重载KMEANS类，将用户指定的距离函数送入 *KMEANS::distEclud* 距离函数指针即可。

代码部分参考来源：
忆之独秀的CSDN博客