第10单元　入侵检测

10.1　知识讲解

10.1.1　KDD Cup 1999数据集

KDD Cup 1999数据集是第三届知识发现与数据挖掘竞赛所使用的数据集。

源于模拟美国空军军事网络环境下的局域网连续九周原始TCP dump数据。

共包含大约700万个样本，分为训练集和测试集两部分。

每个样本包含41个特征，本单元使用其中的20个特征，前三个为标签型特征，其余为数值型特征：

序号	列号	特征	类型
1	2	协议类型	标签
2	3	服务类型	标签
3	4	状态标志	标签
4	5	源到目的的字节数	连续
5	6	目的到源的字节数	连续
6	11	登录失败次数	连续
7	14	是否获得root权限	二值
8	16	获得root权限的次数	连续
9	22	是否以guest身份登录	二值
10	23	两秒钟内连接同一台主机的次数	连续
11	24	两秒钟内连接同一个端口的次数	连续
12	27	REJ错误的比率	连续
13	29	目的端口相同的连接比率	连续
14	30	目的端口不同的连接比率	连续
15	33	目的地址相同，目的端口相同的连接次数	连续
16	34	目的地址相同，目的端口相同的连接比率	连续

17	35	目的地址相同，目的端口不同的连接比率	连续
18	36	目的地址相同，源端口相同的连接比率	连续
19	37	目的地址相同，目的端口相同，源地址不同的连接比率	连续
20	39	目的地址相同，目的端口相同，SYN错误的比率	连续

每个样本对应一个类别，正常或者攻击，攻击分为四种类型：

- 拒绝服务，如SYN洪泛
- 监视与探测，如端口扫描
- 非法获得超级用户权限，如缓冲区溢出
- 非法入侵，如密码猜测

10.1.2 K-Means聚类算法

1. 聚类简介

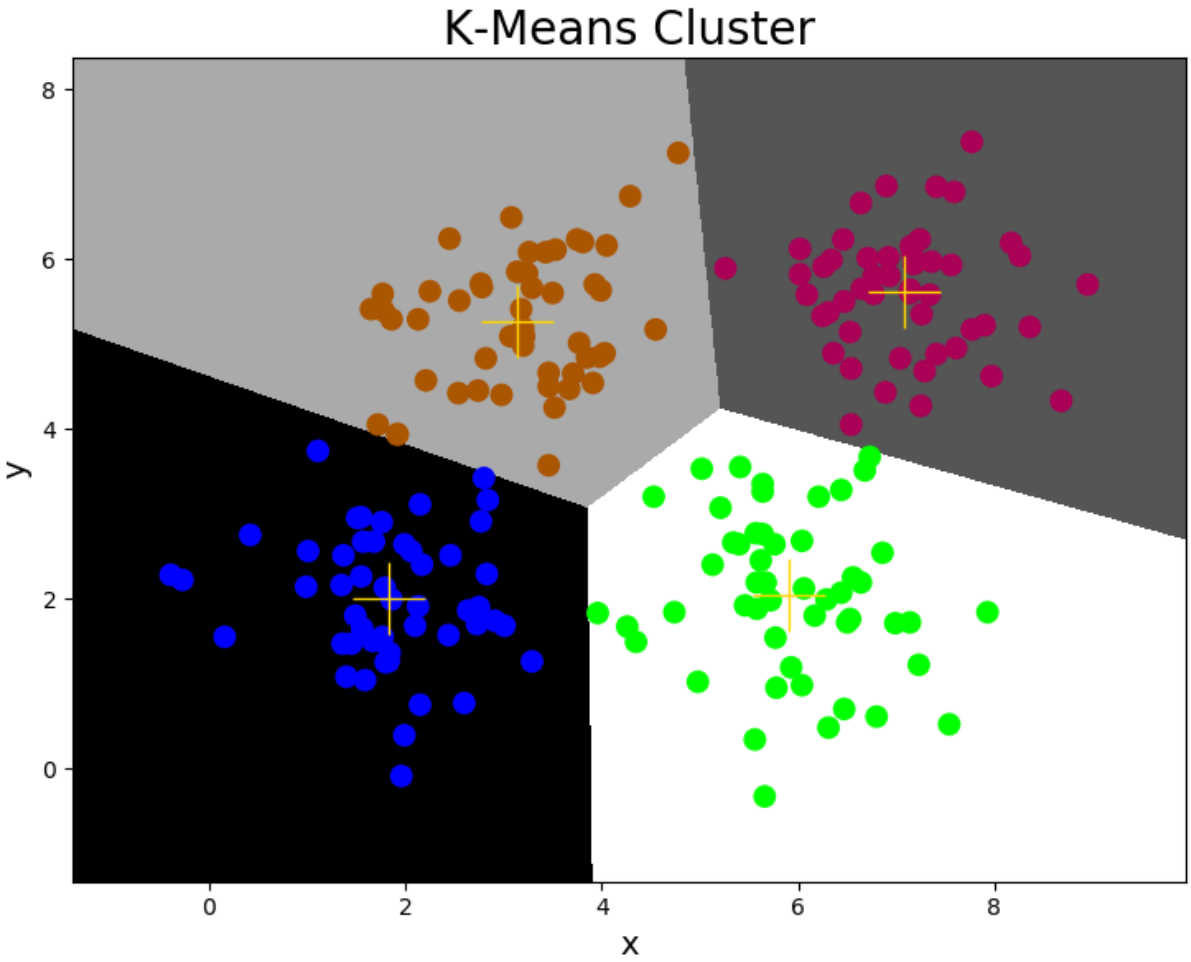
根据具体应用的不同，聚类算法可分为以下五类：

- 基于划分的聚类算法，如K-Means算法
- 基于层次的聚类算法，如凝聚层次算法
- 基于密度的聚类算法，如DBSCAN算法
- 基于网格的聚类算法，如CLIQUE算法
- 基于模型的聚类算法，如均值漂移算法

K-Means算法是一种基于划分的聚类方法，即在一个由n个样本组成的样本空间中构建k $(k < n)$个群落，其中每个群落即表示一个聚类。通过划分得到的聚类必须同时满足以下两个条件：

- 每个聚类至少包含一个样本
- 每个样本必须隶属于且仅隶属于一个聚类

如下图所示：



一般而言，判断划分效果优劣的基本原则是：

- 同一聚类中的样本尽可能地密集，即内密
- 不同聚类中的样本尽可能地疏远，即外疏

同时满足内密外疏的聚类划分就是好的聚类划分，而度量样本疏密的量化指标就是聚类划分的依据，即相似度，如曼哈顿距离、欧几里得距离、闵可夫斯基距离等。

2. 相似度计算

假设总样本空间由n个样本(记录、对象)组成，每个样本包含m个特征(字段、属性)，其中样本X和样本Y记作：

$$X = (x_1, x_2, \ldots, x_m)$$

$$Y = (y_1, y_2, \ldots, y_m)$$

则样本X和样本Y的相似度$D(X, Y)$可用如下距离表示：

1) 曼哈顿距离

$$D_{manhattan}(X,Y)=\sum_{i=1}^{m}\mid x_i-y_i\mid$$

2) 欧几里得距离

$$D_{euclidean}(X,Y)=\sqrt{\sum_{i=1}^{m}\mid x_i-y_i\mid^2}$$

3) 闵可夫斯基距离

$$D_{minkowski}(X,Y)=\sqrt[q]{\sum_{i=1}^{m}\mid x_i-y_i\mid^q}\qquad(q=1,2,\dots)$$

其中，q是一个大于或等于1的正整数：

- $q=1$的闵可夫斯基距离即曼哈顿距离
- $q=2$的闵可夫斯基距离即欧几里得距离

另外，考虑不同特征对相似度影响的差异，可通过特征权重加以体现，权重向量记作：

$$W=(w_1,w_2,\dots,w_m)$$

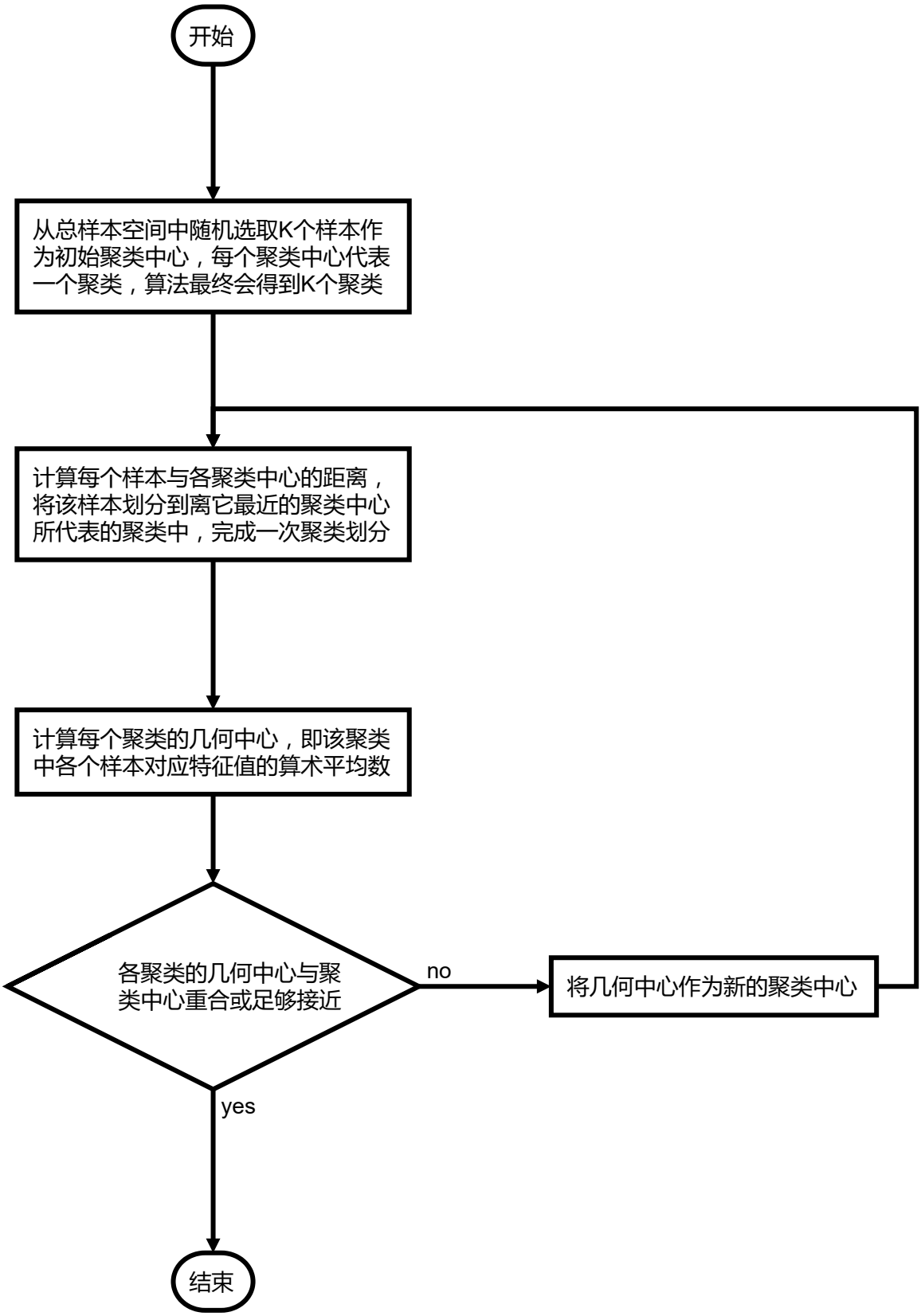
表示样本相似度的三种距离公式为：

$$D_{manhattan}(X,Y,W)=\sum_{i=1}^{m}w_i\mid x_i-y_i\mid$$

$$D_{euclidean}(X,Y,W)=\sqrt{\sum_{i=1}^{m}w_i\mid x_i-y_i\mid^2}$$

$$D_{minkowski}(X,Y,W)=\sqrt[q]{\sum_{i=1}^{m}w_i\mid x_i-y_i\mid^q}\qquad(q=1,2,\dots)$$

3. K-Means算法



10.1.3 K-Means聚类算法的缺陷与扩展

1. K-Means聚类算法的缺陷

KDD Cup 1999数据集包含正常样本和四类攻击样本，共计五个聚类，即$K = 5$。但仅将初始聚类中心的个数设置为5并不一定能获得足够理想的聚类效果。究其原因：

- 同一种类型的攻击，其相似度却可能很低
- 某些攻击样本与正常样本的相似度甚至高于与同类攻击样本的相似度

因此，对KDD Cup 1999数据集应用K-Means算法做聚类分析，无论K值如何选取，仅对数据做一次聚类都是远远不够的。

2. K-Means聚类算法的扩展

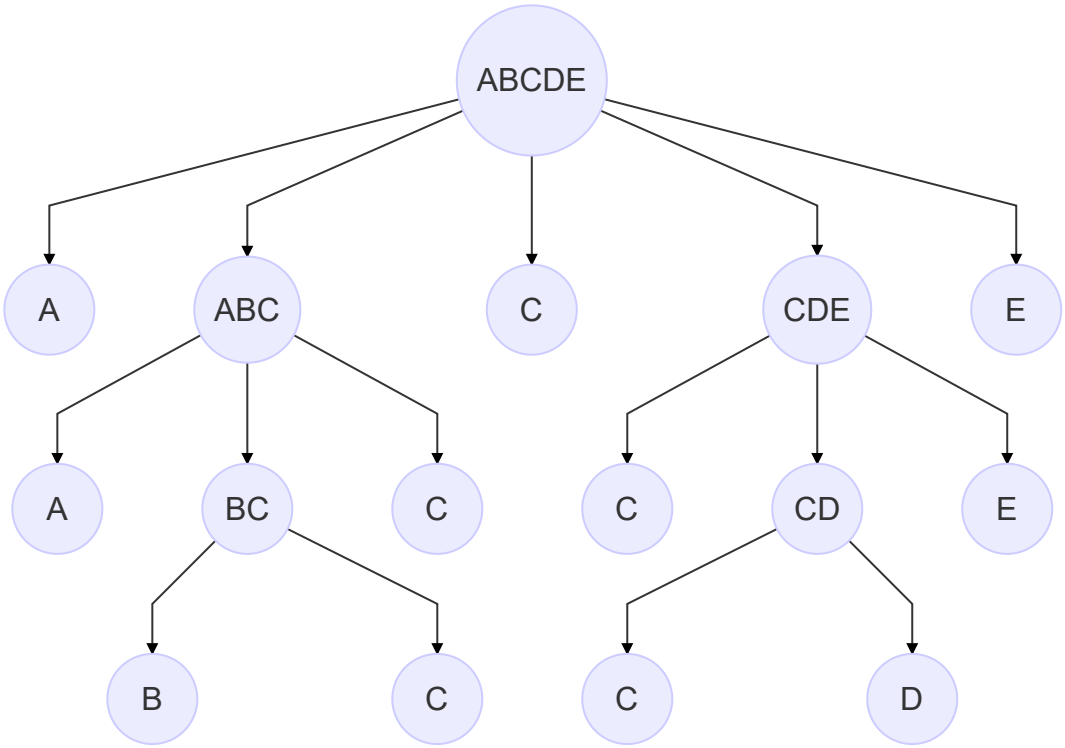
1) 聚类不纯度(Cluster Impurity)

理想情况下，一个聚类中应该只包含一种类别的样本，但事实上有可能包含多种类别的样本，即不够纯。聚类不纯度即表征了一个聚类不纯的程度。其计算过程如下：

- 统计一个聚类中各个类别的样本数
  - 类别1：10个样本
  - 类别2：100个样本
  - 类别3：1000个样本
- 在所有类别中找出样本数最多的类别作为该聚类的主类别
  - 类别3：1000个样本
- 聚类不纯度为非主类别样本数与主类别样本数的比率
  - $CI = \frac{10+100}{1000} = 0.11$

2) 聚类树(Cluster Tree)

利用K-Means算法将总样本空间划分为K个聚类，计算其中每个聚类的聚类不纯度，如果发现某个聚类的聚类不纯度高于事先给定的上限，则按照其所包含的类别数继续进行聚类，不断重复这个过程，形成一棵聚类树，该树的叶级聚类的聚类不纯度均不高于事先给定的上限。如下图所示：



10.1.4 聚类和分类算法的性能指标

为了评估一个聚类或分类算法的性能优劣，需要通过不同的指标加以量化。

1. 聚类算法的性能指标

纯粹的聚类算法属于无监督学习的一种，其用于模型测试的样本通常没有已知的类别标签，因此无法通过预测类别与实际类别之间的符合程度进行性能评估。

轮廓系数(Silhouette Score)可以在无需类别标签的前提下，精确地刻画出聚类本身内密外疏的程度。

对参与模型测试的每一个样本，其轮廓系数可表示为：

$$s = \frac{b - a}{max(a, b)}$$

其中，a代表该样本与其所在聚类中其它样本的平均距离；b则表示该样本与其最近的另一个聚类中所有样本的平均距离：

- 好的聚类，内部样本足够密集，聚类之间足够疏远，$a \ll b$，$s \to 1$
- 坏的聚类，内部样本过于疏远，聚类之间过于密集，$a \gg b$，$s \to -1$
- 聚类重叠，内部样本和聚类之间的疏密程度很接近，$a \approx b$，$s \to 0$

整个聚类的轮廓系数可用所有测试样本轮廓系数的算术平均数表示：

$$S = \overline{s}$$

2. 分类算法的性能指标

1) 混淆矩阵(Confusion Matrix)

对于类别标签已知的测试样本，通常用混淆矩阵表示预测类别与已知类别之间的符合程度。混淆矩阵中的每一行表示一个实际类别的样本数，每一列表示一个预测类别的样本数。如下所示：

	0	1	2	3	4
0	1566	4	16	0	4
1	200	4272	1	0	0
2	6	4	42	0	0
3	0	0	0	2	0
4	100	0	0	0	3

混淆矩阵的主对角线表示实际类别和预测类别一致的样本数。理想混淆矩阵中的所有非零元素，全部集中在主对角线上。换言之，混淆矩阵除主对角线以外的元素数值越小越好，最好都是0。

2) 查准率(Precision)

某个特定类别的查准率表示模型预测该类别的正确性，即模型预测该类别样本中正确样本的占比：

- $p_0 = \frac{1566}{1566+200+6+100} = 0.836538$
- $p_1 = \frac{4272}{4+4272+4} = 0.998131$
- $p_2 = \frac{42}{16+1+42} = 0.711864$
- $p_3 = \frac{2}{2} = 1$
- $p_4 = \frac{3}{4+3} = 0.428571$

分类模型的查准率可用各个类别查准率的算术平均数表示：

- $P = \frac{p_0+p_1+p_2+p_3+p_4}{5} = \frac{0.836538+0.998131+0.711864+1+0.428571}{5} = 0.795021$

3) 召回率(Recall)

某个特定类别的召回率表示模型预测该类别的完整性，即该类别样本中被模型正确预测的样本占比：

- $r_0 = \frac{1566}{1566+4+16+4} = 0.984906$
- $r_1 = \frac{4272}{200+4272+1} = 0.955064$
- $r_2 = \frac{42}{6+4+42} = 0.807692$
- $r_3 = \frac{2}{2} = 1$
- $r_4 = \frac{3}{100+3} = 0.029126$

分类模型的召回率可用各个类别召回率的算术平均数表示：

- $R = \frac{r_0+r_1+r_2+r_3+r_4}{5} = \frac{0.984906+0.955064+0.807692+1+0.029126}{5} = 0.755358$

4) F1得分(F1-Score)

某个特定类别的F1得分用两倍该类别查准率、召回率之积与它们的和的比值表示：

- $f_0 = 2 \times \frac{p_0 \times r_0}{p_0 + r_0} = 2 \times \frac{0.836538 \times 0.984906}{0.836538 + 0.984906} = 0.904679$
- $f_1 = 2 \times \frac{p_1 \times r_1}{p_1 + r_1} = 2 \times \frac{0.998131 \times 0.955064}{0.998131 + 0.955064} = 0.976122$
- $f_2 = 2 \times \frac{p_2 \times r_2}{p_2 + r_2} = 2 \times \frac{0.711864 \times 0.807692}{0.711864 + 0.807692} = 0.756757$
- $f_3 = 2 \times \frac{p_3 \times r_3}{p_3 + r_3} = 2 \times \frac{1 \times 1}{1 + 1} = 1$
- $f_4 = 2 \times \frac{p_4 \times r_4}{p_4 + r_4} = 2 \times \frac{0.428571 \times 0.029126}{0.428571 + 0.029126} = 0.054545$

分类模型的F1得分可用各个类别F1得分的算术平均数表示：

- $F = \frac{f_0 + f_1 + f_2 + f_3 + f_4}{5} = \frac{0.904679 + 0.976122 + 0.756757 + 1 + 0.054545}{5} = 0.738421$

5) 分类报告(Classification Report)

将以上各指标以分类报告的形式加以汇总：

Label	Precision	Recall	F1-Score
0	0.836538	0.984906	0.904679
1	0.998131	0.955064	0.976122
2	0.711864	0.807692	0.756757
3	1	1	1
4	0.428571	0.029126	0.054545
Average	**0.795021**	**0.755358**	**0.738421**

10.2 实训案例

10.2.1 训练并测试用于入侵检测的聚类模型

在Linux系统上编写预处理器和聚类器两个应用程序。

预处理器负责对KDD Cup 1999数据集做预处理。

聚类器使用K-Means算法对训练数据集进行多次聚类分析，形成聚类树模型。

聚类器利用训练得到的聚类树模型，对测试数据集中的每条记录进行类别预测。

10.2.2 程序清单

1. 声明Preprocessor类

```
// preprocessor.h
// 声明Preprocessor类

#pragma once

#include <string>
#include <vector>
using namespace std;

// 预处理器
class Preprocessor {
public:
    // 构造函数
    Preprocessor(const char* input, const char* output);

    // 预处理
    int preprocess(void) const;

private:
    // 编码协议类型
    double encodeProtocol(const string& protocol) const;
    // 编码服务类型
    double encodeService(const string& service) const;
    // 编码状态标志
    double encodeStatus(const string& status) const;
    // 编码攻击类型
    double encodeAttack(const string& attack) const;

    // 抽取特征值
    void extractFeatures(const vector<string>& vs,
        vector<double>& vd) const;

    string input;  // 输入文件
    string output; // 输出文件
};
```

2. 实现Preprocessor类

```cpp
// preprocessor.cpp
// 实现Preprocessor类

#include <stdlib.h>
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;

#include "preprocessor.h"

// 构造函数
Preprocessor::Preprocessor(const char* input, const char* output) :
    input(input), output(output) {}

// 预处理
int Preprocessor::preprocess(void) const {
    // 打开输入文件
    ifstream ifs(input.c_str());
    if (! ifs) {
        cerr << "Unable to open the file: " << input << endl;
        return -1;
    }

    // 打开输出文件
    ofstream ofs(output.c_str());
    if (! ofs) {
        cerr << "Unable to open the file: " << output << endl;
        ifs.close();
        return -1;
    }

    // 从输入文件中读取样本字符串
    string sample;
    for (size_t samples = 0; ifs >> sample; ++samples) {
        // 每个样本中的特征字符串向量
        vector<string> vs;

        // 解析每个样本中的特征字符串
        string::size_type begin = 0;
        for (string::size_type end = sample.find_first_of(",", 0);
            end != string::npos; end = sample.find_first_of(",", begin)) {
            vs.push_back(sample.substr(begin, end - begin));
            begin = end + 1;
        }
        vs.push_back(sample.substr(begin));

        // 包括类别标签在内共42个特征字符串
        if (vs.size() < 42) {
            cerr << "Ivdalid data in the file: " << input << endl;
            ofs.close();
            ifs.close();
            return -1;
        }
```

```
        // 每个样本中的特征浮点数向量
        vector<double> vd;

        // 抽取特征值
        extractFeatures(vs, vd);

        // 包括类别标签在内共21个特征浮点数
        if (vd.size() < 21) {
            cerr << "Ivdalid data in the file: " << input << endl;
            ofs.close();
            ifs.close();
            return -1;
        }

        // 将每个样本的特征浮点数写入输出文件
        typedef vector<double>::const_iterator CFTR;
        for (CFTR ftr = vd.begin(); ftr != vd.end(); ++ftr)
            ofs.write ((char*)&*ftr, sizeof(*ftr));

        cout << "Sample " << samples + 1 << ": " << vd.back() << endl;
    }

    // 无法读取输入文件
    if (! ifs.eof()) {
        cerr << "Unable to read the file: " << input << endl;
        ofs.close();
        ifs.close();
        return -1;
    }

    // 关闭输入输出文件
    ofs.close();
    ifs.close();

    return 0;
}

// 编码协议类型
double Preprocessor::encodeProtocol(const string& protocol) const {
    double code = 0;

    if (protocol == "icmp")
        code = 1;
    else
    if (protocol == "tcp")
        code = 2;
    else
    if (protocol == "udp")
        code = 3;

    return code;
}

// 编码服务类型
```

```
double Preprocessor::encodeService(const string& service) const {
    double code = 0;

    if (service == "domain_u")
        code = 1;
    else
    if (service == "ecr_i")
        code = 2;
    else
    if (service == "eco_i")
        code = 3;
    else
    if (service == "finger")
        code = 4;
    else
    if (service == "ftp_data")
        code = 5;
    else
    if (service == "ftp")
        code = 6;
    else
    if (service == "http")
        code = 7;
    else
    if (service == "hostnames")
        code = 8;
    else
    if (service == "imap4")
        code = 9;
    else
    if (service == "login")
        code = 10;
    else
    if (service == "mtp")
        code = 11;
    else
    if (service == "netstat")
        code = 12;
    else
    if (service == "other")
        code = 13;
    else
    if (service == "private")
        code = 14;
    else
    if (service == "smtp")
        code = 15;
    else
    if (service == "systat")
        code = 16;
    else
    if (service == "telnet")
        code = 17;
    else
    if (service == "time")
```

```
        code = 18;
    else
    if (service == "uucp")
        code = 19;

    return code;
}

// 编码状态标志
double Preprocessor::encodeStatus(const string& status) const {
    double code = 0;

    if (status == "REJ")
        code = 1;
    else
    if (status == "RSTO")
        code = 2;
    else
    if (status == "RSTR")
        code = 3;
    else
    if (status == "S0")
        code = 4;
    else
    if (status == "S3")
        code = 5;
    else
    if (status == "SF")
        code = 6;
    else
    if (status == "SH")
        code = 7;

    return code;
}

// 编码攻击类型
double Preprocessor::encodeAttack(const string& attack) const {
    double code = 0; // 正常

    if (attack == "smurf."           ||
        attack == "neptune."         ||
        attack == "back."            ||
        attack == "teardrop."        ||
        attack == "pod."             ||
        attack == "land.")
        code = 1; // 拒绝服务攻击
    else
    if (attack == "satan."           ||
        attack == "ipsweep."         ||
        attack == "portsweep."       ||
        attack == "nmap.")
        code = 2; // 监视与探测攻击
    else
    if (attack == "buffer_overflow." ||
```

```
        attack == "rootkit."          ||
        attack == "loadmodule."       ||
        attack == "perl.")
        code = 3; // 非法获得超级用户权限攻击
    else
    if (attack == "ftp_write."        ||
        attack == "guess_passwd."     ||
        attack == "imap."             ||
        attack == "multihop."         ||
        attack == "phf."              ||
        attack == "spy."              ||
        attack == "warezclient."      ||
        attack == "warezmaster.")
        code = 4; // 非法入侵攻击

    return code;
}

// 抽取特征值
void Preprocessor::extractFeatures(const vector<string>& vs,
    vector<double>& vd) const {
    // 协议类型
    vd.push_back(encodeProtocol(vs[1]));
    // 服务类型
    vd.push_back(encodeService(vs[2]));
    // 状态标志
    vd.push_back(encodeStatus(vs[3]));

    // 源到目的的字节数
    vd.push_back(atof(vs[4].c_str()));
    // 目的到源的字节数
    vd.push_back(atof(vs[5].c_str()));
    // 登录失败次数
    vd.push_back(atof(vs[10].c_str()));
    // 是否获得root权限
    vd.push_back(atof(vs[13].c_str()));
    // 获得root权限的次数
    vd.push_back(atof(vs[15].c_str()));
    // 是否以guest身份登录
    vd.push_back(atof(vs[21].c_str()));
    // 两秒钟内连接同一台主机的次数
    vd.push_back(atof(vs[22].c_str()));
    // 两秒钟内连接同一个端口的次数
    vd.push_back(atof(vs[23].c_str()));
    // REJ错误的比率
    vd.push_back(atof(vs[26].c_str()));
    // 目的端口相同的连接比率
    vd.push_back(atof(vs[28].c_str()));
    // 目的端口不同的连接比率
    vd.push_back(atof(vs[29].c_str()));
    // 目的地址相同，目的端口相同的连接次数
    vd.push_back(atof(vs[32].c_str()));
    // 目的地址相同，目的端口相同的连接比率
    vd.push_back(atof(vs[33].c_str()));
    // 目的地址相同，目的端口不同的连接比率
```

```
    vd.push_back(atof(vs[34].c_str()));
    // 目的地址相同，源端口相同的连接比率
    vd.push_back(atof(vs[35].c_str()));
    // 目的地址相同，目的端口相同，源地址不同的连接比率
    vd.push_back(atof(vs[36].c_str()));
    // 目的地址相同，目的端口相同，SYN错误的比率
    vd.push_back(atof(vs[38].c_str()));

    // 攻击类型
    vd.push_back(encodeAttack(vs[41]));
}
```

3. 测试Preprocessor类

```
// preprocessor_test.cpp
// 测试Preprocessor类

#include <stdlib.h>
#include <iostream>
using namespace std;

#include "preprocessor.h"

#define TRAINING_INPUT  "../data/kddcup.data_10_percent"
#define TRAINING_OUTPUT "../data/training.dat"
#define TESTING_INPUT   "../data/corrected"
#define TESTING_OUTPUT  "../data/testing.dat"

int main(void) {
    // 训练集预处理器
    Preprocessor training(TRAINING_INPUT, TRAINING_OUTPUT);
    // 预处理训练数据
    if (training.preprocess() == -1)
        return EXIT_FAILURE;

    cout << "-------------------------------------" << endl;

    // 测试集预处理器
    Preprocessor testing(TESTING_INPUT, TESTING_OUTPUT);
    // 预处理测试数据
    if (testing.preprocess() == -1)
        return EXIT_FAILURE;

    return EXIT_SUCCESS;
}
```

4. 测试Preprocessor类构建脚本

```
# preprocessor_test.mak
# 测试Preprocessor类构建脚本

PROJ   = preprocessor_test
OBJS   = preprocessor_test.o preprocessor.o
CXX    = g++
LINK   = g++
RM     = rm -rf
CFLAGS = -c -g -Wall -I.

$(PROJ): $(OBJS)
    $(LINK) $^ -o $@

.cpp.o:
    $(CXX) $(CFLAGS) $^

clean:
    $(RM) $(PROJ) $(OBJS)
```

5. 声明KMeans类

```
// kmeans.h
// 声明KMeans类

#pragma once

#include <vector>
#include <list>
#include <map>
#include <set>
using namespace std;

// K-Means模型
class KMeans {
public:
    // 构造函数
    KMeans(size_t features = 0);

    // 训练
    int train(const char* filename);
    // 打印
    friend ostream& operator<< (ostream& os, const KMeans& model);

    // 测试
    int test(const char* filename,
        vector<pair<double, double> >& labels) const;

    // 混淆矩阵
    static vector<vector<size_t> > confusionMatrix(
        const vector<pair<double, double> >& labels);
    // 分类报告
    static vector<vector<double> > classificationReport(
        const vector<vector<size_t> >& cm);

private:
    // 构造函数
    KMeans(size_t level, const list<vector<double>*>& samples);

    // 随机无重数列
    set<size_t> random(size_t lower, size_t upper, size_t number) const;
    // 欧几里得距离
    double euclid(const vector<double>& x,
        const vector<double>& y) const;
    // 聚类几何中心
    vector<double> mean(const pair<vector<double>,
        list<vector<double>*> >& cluster) const;

    // 初始化聚类中心
    void initCenters(void);
    // 聚类划分
    void cluster(void);
    // 更新聚类中心
    bool updateCenters(void);
    // 聚类不纯度
    double impurity(const pair<vector<double>,
```

```
        list<vector<double>*> >& cluster) const;

    // 测试
    pair<double, double> test(size_t index,
        const vector<double>& sample) const;

    // 读取训练集
    int readTraining(const char* filename);
    // 训练
    void train(void);

    // 读取测试集
    int readTesting(const char* filename,
        list<vector<double> >& testing) const;
    // 测试
    void test(const list<vector<double> >& testing,
        vector<pair<double, double> >& labels) const;

    static const double MAXEU; // 最大中心距离
    static const double MAXCI; // 最大聚类不纯度
    static const size_t MAXIT; // 最大迭代次数
    static const size_t MAXLV; // 最大模型层级
    static const double COVER; // 测试集覆盖率

    size_t                      level;    // 模型层
    size_t                      features; // 特征数
    list<vector<double> >       training; // 训练集
    list<vector<double>*>       samples;  // 样本集
    list<pair<vector<double>,
        list<vector<double>*> > > clusters; // 聚类集
    map<double, KMeans>         models;   // 子模型

    typedef list<vector<double>*>::const_iterator CSAM; // 样本迭代器
    typedef list<pair<vector<double>,
        list<vector<double>*> > >::const_iterator CCLS; // 聚类迭代器
    typedef list<pair<vector<double>,
        list<vector<double>*> > >::iterator       CLS;  // 聚类迭代器
    typedef map<double, KMeans>::const_iterator   CMOD; // 模型迭代器
};
```

6. 实现KMeans类

```cpp
// kmeans.cpp
// 实现KMeans类

#include <stdlib.h>
#include <algorithm>
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;

#include "kmeans.h"

const double KMeans::MAXEU = 1e-3; // 最大中心距离
const double KMeans::MAXCI = 1e-3; // 最大聚类不纯度
const size_t KMeans::MAXIT = 1000; // 最大迭代次数
const size_t KMeans::MAXLV = 20;   // 最大模型层级
const double KMeans::COVER = 0.02; // 测试集覆盖率

// 构造函数
KMeans::KMeans(size_t features /* = 0 */) :
    level(1), features(features) {}

// 训练
int KMeans::train(const char* filename) {
    // 读取训练集
    if (readTraining(filename) == -1)
        return -1;

    // 训练
    train();

    return 0;
}

// 打印
ostream& operator<< (ostream& os, const KMeans& model) {
    string indent;
    for (size_t i = 0; i < model.level - 1; ++i)
        indent += "   ";

    for (KMeans::CCLS cls = model.clusters.begin();
        cls != model.clusters.end(); ++cls) {

        os << indent << '[' << setw(2) << model.level << "] Cluster " <<
            cls->first.back() << ": " << cls->second.size() <<
            endl;

        KMeans::CMOD mod = model.models.find(cls->first.back());
        if (mod != model.models.end())
            os << mod->second;
    }

    return os;
}
```

```
// 测试
int KMeans::test(const char* filename,
    vector<pair<double, double> >& labels) const {
    // 读取测试集
    list<vector<double> > testing;
    if (readTesting(filename, testing) == -1)
        return -1;

    // 测试
    test(testing, labels);

    return 0;
}

// 混淆矩阵
vector<vector<size_t> > KMeans::confusionMatrix(
    const vector<pair<double, double> >& labels) {
    // 混淆矩阵维数
    size_t dim = 0;
    typedef vector<pair<double, double> >::const_iterator CLAB;
    for (CLAB lab = labels.begin(); lab != labels.end(); ++lab) {
        if (dim < lab->first)
            dim = lab->first;
        if (dim < lab->second)
            dim = lab->second;
    }
    ++dim;

    // 填充混淆矩阵
    vector<vector<size_t> > cm(dim, vector<size_t>(dim));
    for (CLAB lab = labels.begin(); lab != labels.end(); ++lab)
        ++cm[lab->first][lab->second];

    return cm;
}

// 分类报告
vector<vector<double> > KMeans::classificationReport(
    const vector<vector<size_t> >& cm) {
    size_t dim = cm.size();

    // 混淆矩阵各行各列分别求和
    vector<double> rs(dim), cs(dim);
    for (size_t i = 0; i < dim; ++i)
        for (size_t j = 0; j < dim; ++j) {
            rs[i] += cm[i][j];
            cs[j] += cm[i][j];
        }

    // 填充分类报告
    vector<vector<double> > cr(dim + 1, vector<double>(3));
    for (size_t i = 0; i < dim; ++i) {
        // 查准率
        cr[i][0] = cs[i] ? cm[i][i] / cs[i] : 1;
```

```
        // 召回率
        cr[i][1] = rs[i] ? cm[i][i] / rs[i] : 1;
        // F1得分
        if (cr[i][0] || cr[i][1])
            cr[i][2] = 2 * cr[i][0] * cr[i][1] / (cr[i][0] + cr[i][1]);

        // 累加查准率、召回率和F1得分
        for (size_t j = 0; j < 3; ++j)
            cr[dim][j] += cr[i][j];
    }

    // 平均查准率、召回率和F1得分
    for (size_t i = 0; i < 3; ++i)
        cr[dim][i] /= dim;

    return cr;
}

// 构造函数
KMeans::KMeans(size_t level, const list<vector<double>*>& samples) :
    level(level), features(samples.front()->size() - 1),
    samples(samples) {
    // 训练
    train();
}

// 随机无重数列
set<size_t> KMeans::random(size_t lower, size_t upper,
    size_t number) const {
    srand(time(NULL));
    set<size_t> sequence;

    while (sequence.size() < number)
        sequence.insert(rand() % (upper - lower) + lower);

    return sequence;
}

// 欧几里得距离
double KMeans::euclid(const vector<double>& x,
    const vector<double>& y) const {
    double d = 0;

    for (size_t i = 0; i < features; ++i)
        d += (x[i] - y[i]) * (x[i] - y[i]);

    return d;
}

// 聚类几何中心
vector<double> KMeans::mean(const pair<vector<double>,
    list<vector<double>*> >& cluster) const {
    if (! cluster.second.size())
        return cluster.first;
```

```
    vector<double> m;

    for (size_t i = 0; i < features; ++i) {
        double s = 0;
        for (CSAM sam = cluster.second.begin();
            sam != cluster.second.end(); ++sam)
            s += (**sam)[i];
        m.push_back(s / cluster.second.size());
    }

    m.push_back(cluster.first.back());

    return m;
}

// 初始化聚类中心
void KMeans::initCenters(void) {
    cout << '[' << setw(2) << level << "] Initializing centers ..." <<
        endl;

    // 生成类别集合
    set<double> labels;
    for (CSAM sam = samples.begin(); sam != samples.end(); ++sam)
        labels.insert((**sam).back());

    // 随机无重数列
    set<size_t> indices = random(0, samples.size(), labels.size());

    // 初始聚类中心
    size_t index = 0;
    double label = 0;
    for (CSAM sam = samples.begin(); sam != samples.end();
        ++sam, ++index)
        if (indices.find(index) != indices.end()) {
            clusters.push_back(make_pair(**sam,
                list<vector<double>*>()));
            clusters.back().first.back() = label++;
        }

    cout << '[' << setw(2) << level << "] Initializing centers OK!" <<
        endl;
}

// 聚类划分
void KMeans::cluster(void) {
    cout << '[' << setw(2) << level << "] Clustering ..." << endl;

    // 计算每个样本
    for (CSAM sam = samples.begin(); sam != samples.end(); ++sam) {
        CLS nearest = clusters.begin();
        double minimum = euclid(**sam, nearest->first);
        // 与各聚类中心的距离
        for (CLS cls = clusters.begin(); cls != clusters.end(); ++cls)
            if (minimum > euclid(**sam, cls->first))
                nearest = cls;
```

```
        // 将该样本划分到离它最近的聚类中心所代表的聚类中
        nearest->second.push_back(*sam);
    }

    for (CCLS cls = clusters.begin(); cls != clusters.end(); ++cls)
        cout << '[' << setw(2) << level << "] Cluster " <<
            cls->first.back() << ": " << cls->second.size() << endl;

    cout << '[' << setw(2) << level << "] Clustering OK!" << endl;
}

// 更新聚类中心
bool KMeans::updateCenters(void) {
    cout << '[' << setw(2) << level << "] Updating centers ..." << endl;

    bool updated = false;

    for (CLS cls = clusters.begin(); cls != clusters.end(); ++cls) {
        // 聚类几何中心
        vector<double> center = mean(*cls);
        // 若几何中心与其聚类中心不重合
        if (euclid(center, cls->first) > MAXEU) {
            // 将几何中心作为新的聚类中心
            cls->first = center;
            cout << '[' << setw(2) << level << "] Center "
                << cls->first.back() << endl;
            updated = true;
        }
    }

    if (updated)
        for (CLS cls = clusters.begin(); cls != clusters.end(); ++cls)
            cls->second.clear();

    cout << '[' << setw(2) << level << "] Updating centers OK!" << endl;
    return updated;
}

// 聚类不纯度
double KMeans::impurity(const pair<vector<double>,
    list<vector<double>*> >& cluster) const {
    // 类别大小
    map<double, double> sizes;
    for (CSAM sam = cluster.second.begin();
        sam != cluster.second.end(); ++sam)
        ++sizes[(**sam).back()];

    if (sizes.empty())
        return 0;

    // 总样本数和主类别样本数
    double total = 0, maximum = sizes.begin()->second;
    typedef map<double, double>::const_iterator CSIZ;
    for (CSIZ siz = sizes.begin(); siz != sizes.end(); ++siz) {
        total += siz->second;
```

```
        if (maximum < siz->second)
            maximum = siz->second;
    }

    return (total - maximum) / maximum;
}

// 测试
pair<double, double> KMeans::test(size_t index,
    const vector<double>& sample) const {
    // 寻找最近聚类
    CCLS nearest = clusters.begin();
    double minimum = euclid(sample, nearest->first);
    for (CCLS cls = clusters.begin(); cls != clusters.end(); ++cls)
        if (! cls->second.empty() &&
            minimum > euclid(sample, cls->first))
            nearest = cls;

    // 最近聚类有子模型
    CMOD mod = models.find(nearest->first.back());
    if (mod != models.end())
        return mod->second.test(index, sample);

    // 类别大小
    map<double, double> sizes;
    for (CSAM sam = nearest->second.begin();
        sam != nearest->second.end(); ++sam)
        ++sizes[(**sam).back()];

    // 主类别
    typedef map<double, double>::const_iterator CSIZ;
    CSIZ main = sizes.begin();
    for (CSIZ siz = sizes.begin(); siz != sizes.end(); ++siz)
        if (main->second < siz->second)
            main = siz;

    double rlabel = sample.back();
    double plabel = main->first;

    cout << '[' << setw(2) << level << "] Sample " <<
        index + 1 << ": " << rlabel << " -> " << plabel << endl;
    return make_pair(rlabel, plabel);
}

// 读取训练集
int KMeans::readTraining(const char* filename) {
    cout << '[' << setw(2) << level << "] Reading training ..." << endl;

    // 打开训练集文件
    ifstream ifs(filename);
    if (! ifs) {
        cerr << "Unable to open the file: " << filename << endl;
        return -1;
    }
```

```
    // 读取训练集文件
    double sample[features+1];
    while (ifs.read ((char*)sample, sizeof(sample))) {
        training.push_back(vector<double>(sample,
            sample + sizeof(sample) / sizeof(sample[0])));
        samples.push_back(&training.back());
    }

    // 无法读取训练集文件
    if (! ifs.eof ()) {
        cerr << "Unable to read the file: " << filename << endl;
        ifs.close();
        return -1;
    }

    // 关闭训练集文件
    ifs.close();

    cout << '[' << setw(2) << level << "] Reading training OK!" << endl;
    return 0;
}

// 训练
void KMeans::train(void) {
    cout << '[' << setw(2) << level << "] Training ..." << endl;

    // 初始化聚类中心
    initCenters();

    for (size_t i = 0; i < MAXIT; ++i) {
        // 聚类划分
        cluster();

        // 更新聚类中心
        if (! updateCenters())
            break;
    }

    // 检查每个聚类的聚类不纯度
    for (CCLS cls = clusters.begin(); cls != clusters.end(); ++cls) {
        double ci = impurity(*cls);
        cout << '[' << setw(2) << level << "] Impurity " <<
            cls->first.back() << ": " << ci << endl;

        // 聚类不纯度过高
        if (ci > MAXCI && level < MAXLV)
            // 在子模型中继续聚类
            models[cls->first.back()] = KMeans(level + 1, cls->second);
    }

    cout << '[' << setw(2) << level << "] Training OK!" << endl;
}

// 读取测试集
int KMeans::readTesting(const char* filename,
```

```
    list<vector<double> >& testing) const {
    cout << '[' << setw(2) << level << "] Reading testing ..." << endl;

    // 打开测试集文件
    ifstream ifs(filename);
    if (! ifs) {
        cerr << "Unable to open the file: " << filename << endl;
        return -1;
    }

    // 读取测试集文件
    double sample[features+1];
    while (ifs.read ((char*)sample, sizeof(sample)))
        testing.push_back(vector<double>(sample,
            sample + sizeof(sample) / sizeof(sample[0])));

    // 无法读取测试集文件
    if (! ifs.eof ()) {
        cerr << "Unable to read the file: " << filename << endl;
        ifs.close();
        return -1;
    }

    // 关闭测试集文件
    ifs.close();

    cout << '[' << setw(2) << level << "] Reading testing OK!" << endl;
    return 0;
}

// 测试
void KMeans::test(const list<vector<double> >& testing,
    vector<pair<double, double> >& labels) const {
    cout << '[' << setw(2) << level << "] Testing ..." << endl;

    // 随机无重数列
    set<size_t> indices = random(0, testing.size(),
        testing.size() * COVER);

    // 遍历测试样本
    size_t index = 0;
    typedef list<vector<double> >::const_iterator CTST;
    for (CTST tst = testing.begin(); tst != testing.end();
        ++tst, ++index)
        if (indices.find(index) != indices.end())
            labels.push_back(test(index, *tst));

    cout << '[' << setw(2) << level << "] Testing OK!" << endl;
}
```

7. 测试KMeans类

```cpp
// kmeans_test.cpp
// 测试KMeans类

#include <stdlib.h>
#include <iostream>
#include <iomanip>
using namespace std;

#include "kmeans.h"

#define TRAINING_FILE "../data/training.dat"
#define TESTING_FILE  "../data/testing.dat"

int main(int argc, char* argv[]) {
    // K-Means模型
    KMeans model(20);

    // 训练
    if (model.train(TRAINING_FILE) == -1)
        return EXIT_FAILURE;

    cout << "----------------------------------" << endl;

    // 打印
    cout << model;

    cout << "----------------------------------" << endl;

    // 测试
    vector<pair<double, double> > labels;
    if (model.test(TESTING_FILE, labels) == -1)
        return EXIT_FAILURE;

    cout << "----------------------------------" << endl;

    // 混淆矩阵
    vector<vector<size_t> > cm = KMeans::confusionMatrix(labels);
    typedef vector<vector<size_t> >::const_iterator CCMR;
    typedef vector<size_t>::const_iterator CCMC;
    for (CCMR row = cm.begin(); row != cm.end(); ++row) {
        for (CCMC col = row->begin(); col != row->end(); ++col)
            cout << setw(6) << *col << ' ';
        cout << endl;
    }

    cout << "----------------------------------" << endl;

    // 分类报告
    cout << " Precision      Recall    F1-Score" << endl;
    cout << "----------------------------------" << endl;
    vector<vector<double> > cr = KMeans::classificationReport(cm);
    typedef vector<vector<double> >::const_iterator CCRR;
    typedef vector<double>::const_iterator CCRC;
    for (CCRR row = cr.begin(); row != cr.end(); ++row) {
```

```
        for (CCRC col = row->begin(); col != row->end(); ++col)
            cout << fixed << setprecision(6) << setw(10) << *col << "  ";
        cout << endl;
        if (row == cr.end() - 2)
            cout << "---------------------------------" << endl;
    }

    return EXIT_SUCCESS;
}
```

8. 测试KMeans类构建脚本

```
# kmeans_test.mak
# 测试KMeans类构建脚本

PROJ   = kmeans_test
OBJS   = kmeans_test.o kmeans.o
CXX    = g++
LINK   = g++
RM     = rm -rf
CFLAGS = -c -g -Wall -I.

$(PROJ): $(OBJS)
    $(LINK) $^ -o $@

.cpp.o:
    $(CXX) $(CFLAGS) $^

clean:
    $(RM) $(PROJ) $(OBJS)
```

10.3 扩展提高

10.3.1 聚类不纯度对入侵检测模型性能的影响

聚类不纯度是一个聚类中干扰数据占主类别数据的比例：

- 聚类不纯度过高，意味着不同类别的数据混入一个聚类，降低入侵检测模型的准确性
- 聚类不纯度过低，延长聚类树生成的收敛过程，增加入侵检测模型的资源消耗
- 选择合理的聚类不纯度标准有助于建立高效准确的入侵检测模型

聚类层次不同对聚类不纯度的上限也不同，层次越深上限越高。

10.3.2 常用入侵检测工具

1. Snort

Snort最早可以追溯到1998年由Martin Roesch编写的一款开源入侵检测工具。迄今为止，Snort已经发展为一个跨平台，能够提供实时流量分析和记录网络数据包的入侵检测与防御系统。

Snort具有三种工作模式：

- 嗅探器：读取网络上的数据包，以连续数据流的形式显示在终端上
- 记录器：捕获网络上的数据包，以纯文本文件的形式记录在硬盘上
- 入侵检测：根据可定制的规则，检测网络上的数据包，对符合规则者执行相应的操作

通过如下命令可以启动Snort的入侵检测工作模式：

```
$ snort -d -h 192.168.1.0/24 -l ./log -c snort.conf
```

其中，snort.conf为描述规则的配置文件。Snort会将捕获到的数据包与配置文件中的每条规则逐一匹配，一旦发现其与某条规则匹配，即执行该规则所描述的操作，同时将符合规则的数据包以文本文件形式保存在-l选项指定的目录中。

Snort规则包括两个逻辑部分：

- 规则头：匹配条件，如动作、协议、源和目的地址、源和目的端口等
- 规则选项：执行操作，如检查内容、收集、报警、丢弃等

例如：

```
alert tcp any any -> 192.168.1.0/24 111 (content: "|00 01 86 a5|"; msg: "mountd access")
\_____________________________________/ \_____________________________________________/
              Rule Header                                Rule Options
```

规则头中可以包含多个条件，它们是逻辑与的关系，而配置文件中的多条规则则是逻辑或的关系。

2. PSAD

端口扫描攻击探测器(The Port Scan Attack Detector, PSAD)通过对防火墙日志的分析，检测包括端口扫描、拒绝服务攻击等在内的各种可疑流量，并通过自动制定防火墙规则，实现有针对性的防御。

PSAD可以和Snort密切配合，通过Snort检测各种后门程序、拒绝服务工具，以及高级端口扫描器的探测行为。PSAD能够检测出Snort规则集中描述的各种攻击行为。

PSAD能够通过对TCP SYN数据包的分析，提取发起端口扫描的远程主机指纹。

PSAD还提供数据输出接口。用户可将PSAD输出的数据导入诸如AfterGlow或GnuPlotd等软件中，绘制各种图表，进一步探查发起各种网络攻击的源头。
