一、 什么是离群点分析

1、什么是离群点？

在样本空间中，与其他样本点的一般行为或特征不一致的点，我们称为离群点。

2、离群点产生的原因？

第一， 计算的误差或者操作的错误所致，比如：某人的年龄 -999 岁，这就是明显由误操作所导致的离群点；
第二， 数据本身的可变性或弹性所致，比如：一个公司中 CEO 的工资肯定是明显高于其他普通员工的工资，于是 CEO 变成为了由于数据本身可变性所导致的离群点。

3、为什么要对离群点进行检测？

“一个人的噪声也许是其他的信号”。换句话说，这些离群点也许正是用户感兴趣的，比如在欺诈检测领域，那些与正常数据行为不一致的离群点，往往预示着欺诈行为，因此成为执法者所关注的。

4、离群点检测遇到的困难？

第一， 在时间序列样本中发现离群点一般比较困难，因为这些离群点可能会隐藏在趋势、季节性或者其他变化中；
第二， 对于维度为非数值型的样本，在检测过程中需要多加考虑，比如对维度进行预处理等；
第三， 针对多维数据，离群点的异常特征可能是多维度的组合，而不是单一维度就能体现的。

二、 几类离群点检测方法

1、基于统计分布的离群点检测

这类检测方法假设样本空间中所有数据符合某个分布或者数据模型，然后根据模型采用不和谐校验（ discordancy test ）识别离群点。不和谐校验过程中需要样本空间数据集的参数知识（ eg: 假设的数据分布），分布的参数知识（ eg: 期望和方差）以及期望的离
群点数目。

不和谐校验分两个过程：工作假设和备选假设

工作假设指的是如果某样本点的某个统计量相对于数据分布的是显著性概率充分小，那么我们则认为该样本点是不和谐的，工作假设被拒绝，此时备用假设被采用，它声明该样本点来自于另一个分布模型。

如果某个样本点不符合工作假设，那么我们认为它是离群点。如果它符合备选假设，我们认为它是符合某一备选假设分布的离群点。

基于统计分布的离群点检测的缺点：

基于统计分布的离群点检测的缺点：

第一， 在于绝大多数不和谐校验是针对单个维度的，不适合多维度空间；
第二， 需要预先知道样本空间中数据集的分布特征，而这部分知识很可能是在检测前无法获得的。

2、基于距离的离群点检测

基于距离的离群点检测指的是，如果样本空间 D 中至少有 N 个样本点与对象 O 的距离大于 dmin, 那么称对象 O 是以 { 至少 N 个样本点 } 和 dmin 为参数的基于距离的离群点。

其实可以证明，在大多数情况下，如果对象 O 是根据基于统计的离群点检测方法发现出的离群点，那么肯定存在对应的 N 和 dmin，是它也成为基于距离的离群点。

Eg: 假设标准正态分布，如果离均值偏差 3 或更大的对象认为是离群点，根据正态曲线概率密度函数， P （ |x-3|<=dmin ） <1-N/ 总点数，即 P （ 3-dim=<x<=3+dmin ） <1-N/ 总点数，假设 dmin=0.13, 则该 dmin 领域表示 [2.87,3.13] 的范围，假设总点数 =10000, N=12.

基于距离的离群点检测的缺点：

要求数据分布均匀，当数据分布非均匀时，基于距离的离群点检测将遇到困难。

3、基于密度的局部离群点检测

什么是局部离群点？

一个对象如果是局部离群点，那么相对于它的局部领域，它是远离的。

不同于前面的方法，基于密度的局部离群点检测不将离群点看做一种二元性质，即不简单用 Yes or No 来断定一个点是否是离群点，而是用一个权值来评估它的离群度。

它是局部的，意思是该程度依赖于对象相对于其领域的孤立情况。这种方法可以同时检测出全局离群点和局部离群点。

通过基于密度的局部离群点检测就能在样本空间数据分布不均匀的情况下也可以准确发现离群点。

4、基于偏差的离群点检测

基于偏差的离群点检测，它通过检查一组对象的主要特征来识别离群点，“偏差”这种特征的点我们认为是离群点。

通常有两种技术：

第一， 顺序异常技术
第二， 采用 OLAP 数据立方体技术

三、 基于密度的局部离群点检测

前面介绍了什么叫做基于密度的局部离群点检测，以及它的优势。现在详细介绍下它的一些概念。

1、 对象 p 的第 k 距离

对于正整数 k, 对象 p 的第 k 距离可记作 k-distance(p)。

在样本空间中，存在对象 o，它与对象 p 之间的距离记作 d(p,o)。如果满足以下两个条件，我们则认为 k-distance(p)= d(p,o)

1） 在样本空间中，至少存在 k 个对象 q, 使得 d(p,q)<= d(p,o);
2） 在样本空间中，至多存在 k-1 个对象 q, 使得 d(p,q)<d(p,o)

换句话说，满足这两个标准的 k-distance(p) 其实就是计算样本空间中其他对象与对象 p 之间的距离，然后找到第 k 大的那个距离，即为 k-distance(p)。显而易见，如果使用 k-distance(p) 来量化对象 p 的局部空间区域范围，那么对于对象密度较大的区域， k-distance(p) 值较小，而对象密度较小的区域， k-distance(p) 值较大。

2、 对象 p 的第 k 距离领域（ k-distance neighborhood of an object p ）

已知对象 p 的第 k 距离，那么，与对象 p 之间距离小于等于 k-distance(p) 的对象集合称为对象 p 的第 k 距离领域，记作： $N_{kdis(p)}(p)$

该领域其实是以 p 为中心， k-distance(p) 为半径的区域内所有对象的集合（不包括 P 本身）。由于可能同时存在多个第 k 距离的数据，因此该集合至少包括 k 个对象。

可以想象，离群度较大的对象 $N_{kdis(p)}(p)$ 范围往往比较大，而离群度小的对象 $N_{kdis(p)}(p)$ 范围往往比较小。对于同一个类簇中的对象来说，它们涵盖的区域面积大致相当。

3、 对象 p 相对于对象 o 的可达距离

可达距离 $reachdis_k(p,o)=max\{k\text{-distance}(o),d(p,o)\}$, 即 k-distance(o) 和 d(p,o) 值较大的那个。

4、 局部可达密度是基于 p 的 k 最近邻点的平均可达密度的倒数。如下

$Lrd_k(p)=|N_k(p)|/\sum_{o\in Nk(p)} reachdis_k(p,o)$ （$|N_k(p)|$表示对象 p 的第 k 距离领域的对象个数）

可以发现，如果对象 p 的离群度较小，那么对于同一类簇的数据对象 $reachdis_k(p,o)$ 取 k-distance(o) 可能性较大，因此它们的 $Lrd_k(p)$ 值波动性较小；而如果对象 p 的利群度较大，那么 $reachdis_k(p,o)$ 取 d(p,o) 的可能性较大，对于同一类簇的数据对象，它们的 $Lrd_k(p)$ 值波动性也比较大，并且 $Lrd_k(p)$ 值较小。

5、 局部离群点因子（ LOF ）

$$\sum \frac{lrd_{MinPts}(o)}{lrd_{MinPts}(p)}$$

$$LOF_{MinPts}(p) = \frac{\sum_{o \in N_{MinPts}(p)} \frac{lrd_{MinPts}(o)}{lrd_{MinPts}(p)}}{|N_{MinPts}(p)|},$$

它代表了 p 为离群点的程度。如果对象 p 的离群程度较大，则它 k 领域中大多数是离对象 p 较远且处于某一个类簇的数据对象，那么这些数据对象的 lrd 应该是偏大，而对象 p 本身的 lrd 是偏小，最后所得的 LOF 值也是偏大。反之，如果对象 p 的离群程度较小，对象 o 的 lrd 和对象 p 的 lrd 相似，最后所得的 lof 值应该接近 1.

四、 算法实现

算法：基于密度的局部离群点检测（ lof 算法）

输入：样本集合 D ，正整数 K （用于计算第 K 距离）

输出：各样本点的局部离群点因子

过程： 1 ）计算每个对象与其他对象的欧几里得距离

2 ）对欧几里得距离进行排序，计算第 k 距离以及第 K 领域

3 ）计算每个对象的可达密度

4 ）计算每个对象的局部离群点因子

5 ）对每个点的局部离群点因子进行排序，输出。

源码：

```
package com.lof;

import java.util.ArrayList;
import java.util.List;

public class Node {
   private String nodeName; // 样本点名
   private double[] dimensioin; // 样本点的维度
   private double kDistance; // k-距离
   private List<Node> kNeighbor=new ArrayList<Node>();// k-领域
   private double distance; //到给定点的欧几里得距离
   private double reachDensity;// 可达密度
   private double reachDis;// 可达距离


   private double lof;//局部离群因子

   public Node(){

   }
```

```
public Node(String nodeName,double[] dimensioin){
    this.nodeName=nodeName;
    this.dimensioin=dimensioin;
}

public String getNodeName() {
    return nodeName;
}

public void setNodeName(String nodeName) {
    this.nodeName = nodeName;
}

public double[] getDimensioin() {
    return dimensioin;
}

public void setDimensioin(double[] dimensioin) {
    this.dimensioin = dimensioin;
}

public double getkDistance() {
    return kDistance;
}

public void setkDistance(double kDistance) {
    this.kDistance = kDistance;
}

public List<Node> getkNeighbor() {
    return kNeighbor;
}

public void setkNeighbor(List<Node> kNeighbor) {
    this.kNeighbor = kNeighbor;
}


public double getDistance() {
    return distance;
}

public void setDistance(double distance) {
    this.distance = distance;
}
```

```
    public double getReachDensity() {
        return reachDensity;
    }

    public void setReachDensity(double reachDensity) {
        this.reachDensity = reachDensity;
    }

    public double getReachDis() {
        return reachDis;
    }

    public void setReachDis(double reachDis) {
        this.reachDis = reachDis;
    }

    public double getLof() {
        return lof;
    }

    public void setLof(double lof) {
        this.lof = lof;
    }

}
package com.lof;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;


public class OutlierNodeDetect {
    private static int MIN_PTS=5;

    //1.找到给定点与其他点的欧几里得距离
    //2.对欧几里得距离进行排序，找到前5位的点，并同时记下k距离
    //3.计算每个点的可达密度
    //4.计算每个点的局部离群点因子
    //5.对每个点的局部离群点因子进行排序，输出。
    public List<Node> getOutlierNode(List<Node> allNodes){

        List<Node> kdAndKnList=getKDAndKN(allNodes);
        calReachDis(kdAndKnList);
```

```
    calReachDensity(kdAndKnList);
    calLof(kdAndKnList);
    Collections.sort(kdAndKnList, new LofComparator());

    return kdAndKnList;
}


private void calLof(List<Node> kdAndKnList){
    for(Node node:kdAndKnList){
        List<Node> tempNodes=node.getkNeighbor();
        double sum=0.0;
        for(Node tempNode:tempNodes){
            double rd=getRD(tempNode.getNodeName(),kdAndKnList);
            sum=rd/node.getReachDensity()+sum;
        }
        sum=sum/(double)MIN_PTS;
        node.setLof(sum);
    }
}


private void calReachDensity(List<Node> kdAndKnList){
    for(Node node:kdAndKnList){
        List<Node> tempNodes=node.getkNeighbor();
        double sum=0.0;
        double rd=0.0;
        for(Node tempNode:tempNodes){
            sum=tempNode.getReachDis()+sum;
        }
        rd=(double)MIN_PTS/sum;
        node.setReachDensity(rd);
    }
}


private void calReachDis(List<Node> kdAndKnList){
    for(Node node:kdAndKnList){
        List<Node> tempNodes=node.getkNeighbor();
        for(Node tempNode:tempNodes){
            double kDis=getKDis(tempNode.getNodeName(),kdAndKnList);
            if(kDis<tempNode.getDistance()){
                tempNode.setReachDis(tempNode.getDistance());
            }else{
                tempNode.setReachDis(kDis);
            }
```

```
            }
        }
    }

    private double getKDis(String nodeName,List<Node> nodeList){
        double kDis=0;
        for(Node node:nodeList){
            if(nodeName.trim().equals(node.getNodeName().trim())){
                kDis=node.getkDistance();
                break;
            }
        }
        return kDis;

    }


    private double getRD(String nodeName,List<Node> nodeList){
        double kDis=0;
        for(Node node:nodeList){
            if(nodeName.trim().equals(node.getNodeName().trim())){
                kDis=node.getReachDensity();
                break;
            }
        }
        return kDis;

    }


    private List<Node> getKDAndKN(List<Node> allNodes){
       List<Node> kdAndKnList=new ArrayList<Node>();
       for(int i=0;i<allNodes.size();i++){
          List<Node> tempNodeList=new ArrayList<Node>();
          Node nodeA=new
Node(allNodes.get(i).getNodeName(),allNodes.get(i).getDimensioin());
          for(int j=0;j<allNodes.size();j++){
             Node nodeB=new
Node(allNodes.get(j).getNodeName(),allNodes.get(j).getDimensioin());
             double tempDis=getDis(nodeA,nodeB);
             nodeB.setDistance(tempDis);
             tempNodeList.add(nodeB);
          }

          //对tempNodeList进行排序
          Collections.sort(tempNodeList, new DistComparator());
```

```
        Collections.sort(tempNodeList, new DistComparator());
        for(int k=1;k<MIN_PTS;k++){
            nodeA.getkNeighbor().add(tempNodeList.get(k));
            if(k==MIN_PTS-1){
                nodeA.setkDistance(tempNodeList.get(k).getDistance());
            }
        }
        kdAndKnList.add(nodeA);
    }

    return kdAndKnList;
}


private double getDis(Node A,Node B){
    double dis=0.0;
    double[] dimA=A.getDimensioin();
    double[] dimB=B.getDimensioin();
    if (dimA.length == dimB.length) {
        for (int i = 0; i < dimA.length; i++) {
            double temp = Math.pow(dimA[i] - dimB[i], 2);
            dis = dis + temp;
        }
        dis=Math.pow(dis, 0.5);
    }
    return dis;
}

class DistComparator implements Comparator<Node>{
    public int compare(Node A, Node B){
        return A.getDistance()-B.getDistance()<0?-1:1;
    }
}

class LofComparator implements Comparator<Node>{
    public int compare(Node A, Node B){
        return A.getLof()-B.getLof()<0?-1:1;
    }
}

public static void main(String[] args){
    ArrayList<Node> dpoints = new ArrayList<Node>();

    double[] a={2,3};
    double[] b={2,4};
    double[] c={1,4};
    double[] d={1,3};
```

```
double[] d={1,3};
double[] e={2,2};
double[] f={3,2};

double[] g={8,7};
double[] h={8,6};
double[] i={7,7};
double[] j={7,6};
double[] k={8,5};

double[] l={100,2};//孤立点


double[] m={8,20};
double[] n={8,19};
double[] o={7,18};
double[] p={7,17};
double[] q={8,21};

dpoints.add(new Node("a",a));
dpoints.add(new Node("b",b));
dpoints.add(new Node("c",c));
dpoints.add(new Node("d",d));
dpoints.add(new Node("e",e));
dpoints.add(new Node("f",f));

dpoints.add(new Node("g",g));
dpoints.add(new Node("h",h));
dpoints.add(new Node("i",i));
dpoints.add(new Node("j",j));
dpoints.add(new Node("k",k));

dpoints.add(new Node("l",l));

dpoints.add(new Node("m",m));
dpoints.add(new Node("n",n));
dpoints.add(new Node("o",o));
dpoints.add(new Node("p",p));
dpoints.add(new Node("q",q));

OutlierNodeDetect lof=new OutlierNodeDetect();

List<Node> nodeList=lof.getOutlierNode(dpoints);

for(Node node:nodeList){
   System.out.println(node.getNodeName()+"  "+node.getLof());
```

```
    }

  }
}
```

测试结果：

q

0.7459309435620392
p 0.7459309435620392
e 0.7485293162241347
k 0.7518479734971145
i 0.7518479734971146
c 0.7693717709826069
b 0.7693717709826069
g 0.7836550344036045
o 0.8175878600290553
m 0.8175878600290553
a 0.827181166228103
d 0.8497518729207414
f 0.8588773305030418
j 0.8625820667657609
h 0.8625820667657609
n 0.8866630038097529
l 39.309353884068194