异常探测

异常，即“异于寻常”。异常探测（又称为离群点检测）就是找出其行为很不同于预期对象的一个检测过程。这些对象被称为异常点或者离群点。

学者 Hawkins 定义异常值是一个与其他值偏差很大的观察值，这使得我们怀疑它是由一个不同的机制产生的。

可以看出，尽管大家对于异常的定义有所区别，但是学术界对于异常有一个共识：异常是不同的。此外，许多算法也假设异常是少数的。

异常点检测在生产生活中有着许多具体的应用，比如信用卡欺诈，工业设备故障检测，网络安全检测等。本章首先介绍异常探测领域非常基础的知识，让读者能够了解异常探测的基本方法。然后我们将介绍几种非常具有代表性的异常探测算法，它们分别具有不同的思想，能够让读者在学习算法的同时有所启发。

基础知识

基本统计方法介绍

一般来说，进行异常点检测的方法有很多，最常见的就是基于统计学的方法。

基于正态分布的一元离群点检测方法

假设有 n 个点（$x_1, \cdots, x_n$），那么这 n 个点的均值μ和方差σ都可以计算出来。

$$\mu = \sum_{i=1}^{n} x_i \,/\mathrm{n}$$

$$\sigma^2 = \sum_{i=1}^{n} (x_i - \mu)^2 \,/\mathrm{n}$$

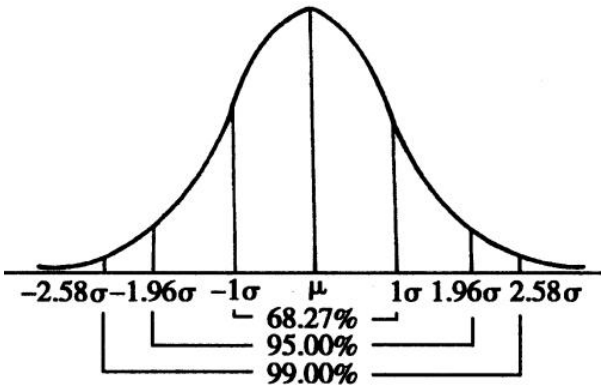


图 1 正态分布曲线

在正态分布的假设下，区域$\mu \mp 3\sigma$ 包含了 99.7%的数据，如果某个值和均值μ的距离超过了3σ，那么这个值就被视为异常，这是应用了异常“少而不同”的特点。

工业上常用的“3 西格玛原则”和“6 西格玛原则”就是基于这个思想。

多元离群点的检测方法

涉及两个或者两个以上变量的数据称为多元数据，很多一元离群点的检测方法都可以扩展到高维空间中，从而处理多元数据。我们介绍一种基于一元正态分布的离群点检测方法

设 n 维的数据集合形如$x_i = \{x_{i1}, x_{i2}, \cdots, x_{in}\}, \mathrm{i} \in \{1, \cdots, \mathrm{m}\}$,

那么每个维度的均值和方差都可计算出来，对于$\mathrm{j} \in \{1, \cdots, \mathrm{n}\}$:

$$\mu_j = \sum_{i=1}^{m} x_{ij} \,/\mathrm{m}$$

$$\sigma_j^2 = \sum_{i=1}^{m} (x_{ij} - \mu_j)^2 \,/\mathrm{m}$$

假设数据分布为正态分布的情况下，对于一个新数据 x,可以计算其概率：

$$\mathrm{p(x)} = \prod_{j=1}^{n} p\left(x_j; \mu_j, \sigma_j^2\right) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_j} \exp\left(-\frac{(x_j - \mu_j)^2}{2\sigma_j^2}\right)$$

这个概率值的大小就代表 x 的异常程度。

使用 Mahalanobis 距离检测多元离群点

对一个多为数据集合 D，假设μ是均值向量，那么对于 D 中的其他对象 x，x 到μ的 Mahalanobis 距离为：

$$\mathrm{MDist(x, \mu)} = \sqrt{(x - \mu)^T S^{-1} (x - \mu)}$$

其中 S 是协方差矩阵。

在这里 MDist(x, μ)是一个数值，我们可以对这个数值进行排序，如果数值过大就认为 x 是一个离群点。

我们用一个程序来介绍这个方法：我们要找到一群孩子当中身体发育不良的孩子。

```
#coding:utf-8
from numpy import float64
from matplotlib import pyplot as plt
import pandas as pd
import numpy as np
from scipy.spatial import distance
from pandas import Series
Height = np.array([164, 167, 168, 169, 169, 170, 170, 170, 171, 172,
172, 173, 173, 175, 176, 178], dtype=float64)
Weight = np.array([54,  57,  58,  60,  61,  60,  61,  62,  62,  64,
62,  62,  64,  56,  66,  70], dtype=float64)
hw = {'Height': Height, 'Weight': Weight}
hw = pd.DataFrame(hw)
n_outliers = 2#我们设置有两个异常值
```

```
#计算每个样本的马氏距离，并且从大到小排序，越大则越有可能是离群点,返回其位置
m_dist_order = Series([float(distance.mahalanobis(hw.iloc[i],
hw.mean(), np.mat(hw.cov().as_matrix()).I) ** 2)
                       for i in
range(len(hw))]).sort_values(ascending=False).index.tolist()
is_outlier = [False, ] * 16 #返回长度为16的全FALSE的列表
for i in range(n_outliers):# n_outliers = 2,找出马氏距离最大的两个样本，
标记为True,为离群点
    is_outlier[m_dist_order[i]] = True
color = ['g', 'black']
pch = [1 if is_outlier[i] == True else 0 for i in
range(len(is_outlier))]
cValue = [color[is_outlier[i]] for i in range(len(is_outlier))]
fig = plt.figure()
plt.title('Scatter Plot')
plt.xlabel('Height cm')
plt.ylabel('Weight kg')
plt.scatter(hw['Height'], hw['Weight'], s=40, c=cValue)
plt.show()
```

运行结果如下图：

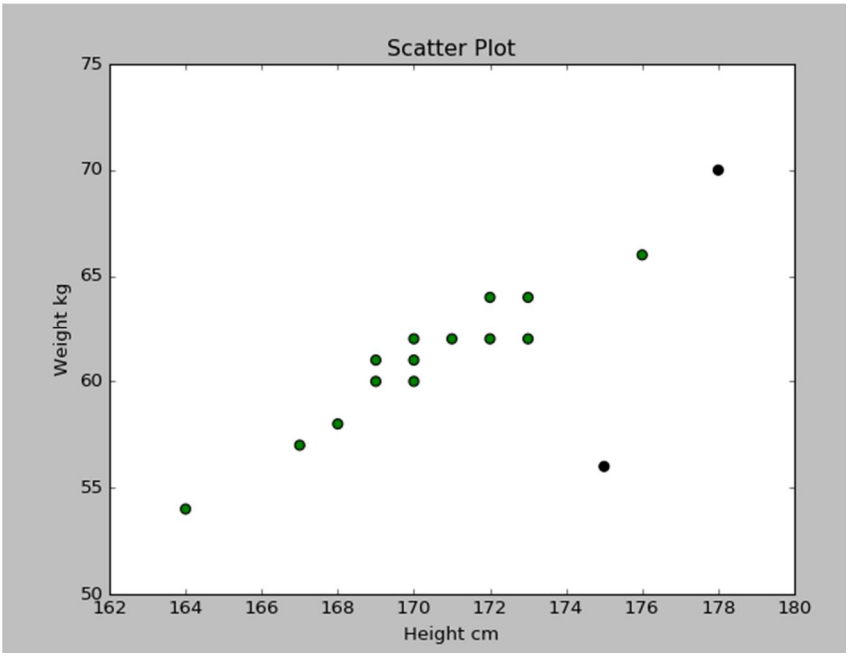


图 2 我们发现有两个孩子不太健康

需要注意的是：马氏距离仅仅把变量之间的关系作为线性关系处理。例如上面的身高和体重的关系，按常识，身高和体重必然存在线性关系，所以马氏距离能很好的检测到异常值，但是如果是非线性关系马氏距离就不适用了。

箱线图异常检测

箱线图是一种用作显示一组数据分散情况资料的统计图。因形状如箱子而得名。箱线图有五个要素：

（1）中位数

（2）上四分位数 Q1：四分位数的求法，是将序列平均分成四份。假设数据大小为 n,一般使用(n+1)/4 来计算。

（3）下四分位数 Q3：与上四分位数计算相似，一般用(1+n)/4*3 来计算。

（4）内限：对于上边界：Q3+1.5IQR（其中 IQR = Q3 - Q1）与剔除异常值后的极大值两者取最小；对于下边界：是 Q1-1.5IQR 与剔除异常值后的极小值两者取最大。

（5）外限：外限的计算与内限相似，只不过 IQR 前的系数变为 3。

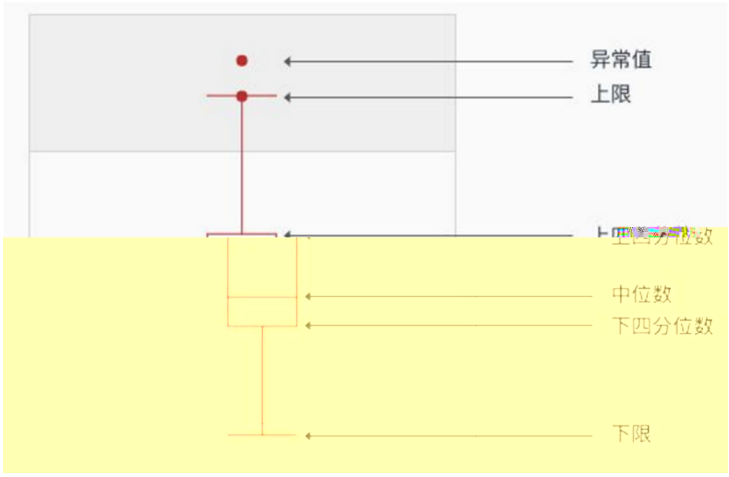


图 3：使用箱线图进行异常探测

常用的异常探测策略为：对于一个数据 x：

最小估计值为：Q1 - k(Q3 - Q1)，最大估计值为：Q3 + k(Q3 - Q1)。

K = 1.5 时认为是中度异常，k=3 时认为是极度异常。

异常探测评价指标

假设我们应用刚刚讲过的多元离群点的检测方法进行异常探测，我们通过 p(x)来判断新样本是否是异常，那么如何衡量这种方法是否达到性能要求呢？

我们通常使用精度 precision，召回率 recall 和 F 度量来衡量。

$$\text{precision} = \frac{TP}{TP + FP}$$

$$\text{recall} = \frac{TP}{TP + FN} = \frac{TP}{P}$$

$$\text{F} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

在异常探测问题中，通常认为 TP 是探测成功的异常元组个数，FP 是误报的异常（实际是正常，称为假正）元组个数；TN 是正确判定为正常的元组个数，FN 是没有被判定为异常的元组个数。

异常探测与分类问题的区别

既然在一些异常探测问题中，我们有正常的数据，又有异常的数据，为什么不直接使用分类算法呢？这是因为异常检测和监督学习有着不同的适用范围。

异常检测适用范围

（1）异常样本非常少，但正常样本很多。
（2）异常类型很多，算法很难从正常的数据样例里学习到异常的特征。
（3）未来的异常数据和我们训练样例里已知的异常数据根本不一样，即没见过的异常样例。

如果满足这三个条件中的任何一个，都需要考虑使用异常检测算法，而不是监督学习相关的算法。

监督学习算法适用于有大量的正向样本，也有大量的负向样本，有足够的正向样本让算法来学习其特征，未来新出现的正向数据可能和训练样例里的某个正向样本类似。由此可见异常检测和监督学习相关算法的适用范围是不一样的。

异常探测算法分类

我们已经介绍完基于统计的异常检测算法，这类方法往往通过概率计算判定异常。

异常探测方法还有很多其他的类别，我们简单的介绍一下：

（1） 基于统计学的方法。这是一种基于模型的方法，假定数据是由某个随机模型产生的，如果某个数据概率很低，那么它是异常的概率就很大。
（2） 基于邻近性的方法。这类方法的核心思想在于，如果一个数据离它最近的邻居都很远，那么它很可能是异常。因此 KNN 算法可以应用到这类异常探测问题中。
（3） 基于聚类的方法。假定正常数据的数量很大，很稠密，而异常值则很小，很稀疏。从而通过聚类，可以找到簇之外的离群点，判定为异常。因此许多聚类算法，如 kmeans,DBscan 都可以用于异常检测。

对于每个类别，还可以细分为多个方法，我们在此不再赘述。

对于这些方法中的每一种，数据的性质可以是监督的，半监督的或无监督的。

•在监督情况下，分类标签对于一组训练数据是已知的，所有的基于比较和距离的方法都与这种有标签的训练数据有关。

•在无监督的情况下，没有标签是已知的，所以距离和比较是在整个数据集上进行的。

•在半监督问题中，标签对于某些数据是已知的，但对于大多数其他数据来说并不知道。例如，我们已知某些类别的恶意软件，半监督学习算法可以试图确定哪些其他可疑的恶意软件属于同一类别。算法通常分多个阶段进行，在早期阶段给无标签的数据分配暂定的标签。

无监督异常检测算法应该符合以下特征：

1.正常行为必须动态定义。没有已有的训练数据集或者参考数据集可以定义

异常。

2.即使数据分布未知，也必须有效地检测到异常值。

隔离森林

大多数现有的基于模型的异常检测方法都会构建正常数据的概要(描述正常数据“长”成什么样子)，然后将不符合概要的实例标记为异常。本节介绍的隔离森林(iForest)则从“隔离”的角度进行异常探测。

隔离森林具有线性时间复杂度，低内存，探测结果良好的特点，引起了工业界的广泛关注。隔离森林对异常做如下假定：

（1） 异常是少数数据。

（2） 异常数据与正常数据有明显的区别，或者说异常数据与正常数据相比，具有非常不同的属性值。

也就是说异常是'少而不同'的，这使得异常数据比正常数据更容易受到“隔离”影响。隔离森林使用树结构来有效隔离每个实例。利用隔离森林，使得异常易于分离，异常点更接近树的根部;而正常点被隔离在树的较深的一端。

这个树的隔离特性构成了隔离森林算法的基础，我们把这棵树叫做隔离树或者 iTree。隔离森林(iForest)在给定的数据集建立一个 iTree 集合，然后异常数据就是在全部的 iTree 上平均路径较短的数据。

隔离森林算法有两个变量：

（1） iTree 的个数。

（2） 子采样大小。

经过实验证明，隔离森林只需要较小的子采样大小即可实现高效率的检测性能。介绍完隔离森林的基本思想和特点，我们接下来将详细介绍隔离森林算法的相关细节。

隔离与隔离树 iTree

术语“隔离”意味着“将一个实例与其余实例分离”。由于异常”少而不同”，因此他们更容易受到隔离的影响。在一个随机树中，重复递归地划分全部的数据实例，直到所有数据实例都被隔离。这种随机划分会导致异常数据具有明显更短的路径，这是因为：

（1） 异常数据越少，导致划分次数也越少，使得异常数据在树结构中具有更短的路径。

（2） 由于异常“不同”，具有与众不同的数据特征值，使得异常更容易在早期划分数据时就被分离，从而路径更短。

因此，当一个随机树构成的树林为某些特定点产生了更短的路径长度，那么这个特定点很可能是异常的。

为了证明异常在随机划分下更容易被隔离的想法，论文中给出了一个图示来说明这一基本特性。

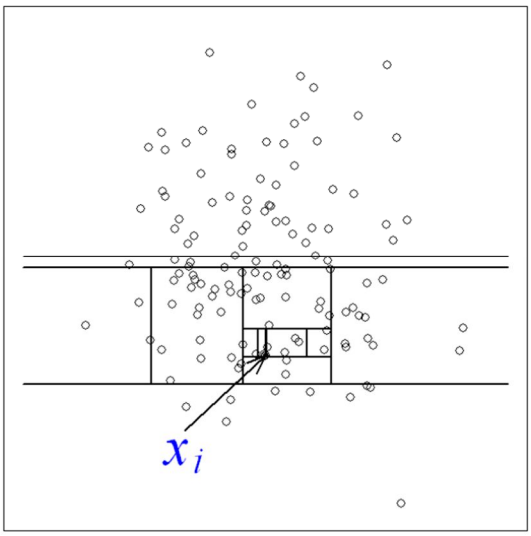


图 4(a)隔离 xi

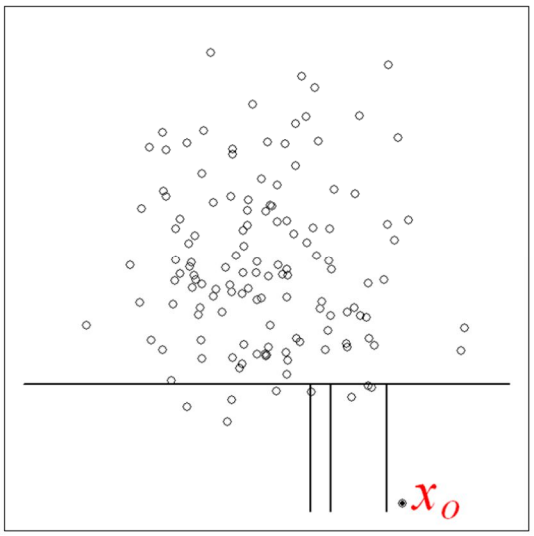


图 4(b)隔离 x0

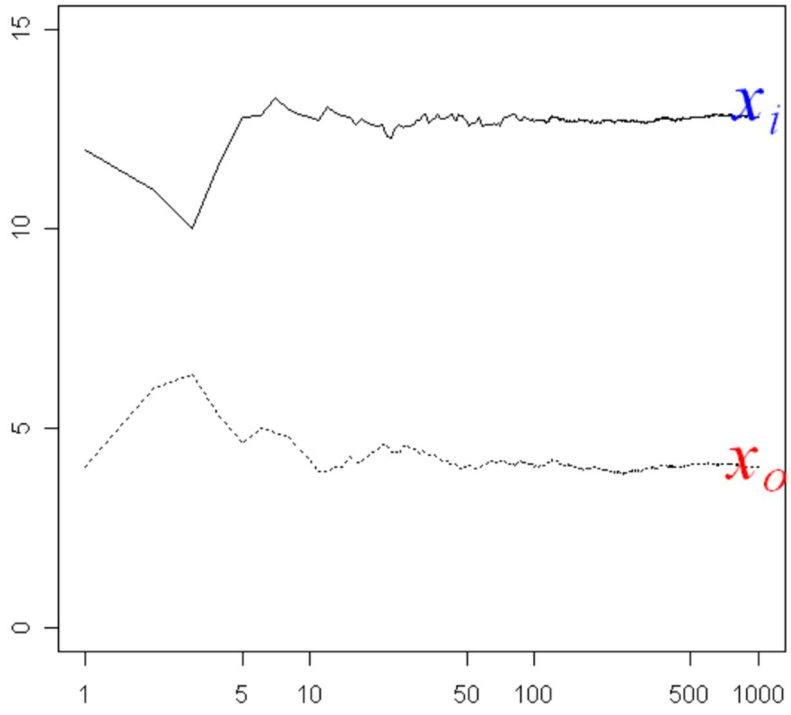


图 4(c)平均路径长度收敛

异常更容易受到隔离的影响，因此具有较短的路径长度：给定高斯分布(135 个点)，图 4(a)中正常点 xi 需要十二次随机划分才能被隔离; 而图 4(b)中异常 x0 只需要 4 次划分就被隔离。

我们观察到，正常点 xi 通常需要更多的划分才能被隔离。同时，异常点 xo 只需要较少的划分就能被隔离。在这个例子中，划分是通过随机选择一个属性，然后随机选择该属性最大值和最小值之间的分割值来产生的。由于递归划分可以由树结构表示，隔离一个数据点所需的划分数量等于从根节点到终止节点的路径长度。在这个例子中，xi 的路径长度明显大于 xo 的路径长度。

由于每个划分是随机生成的，因此每棵树都是由不同的数据划分集合生成的。我们在多个树上求取平均路径长度以找到预期的路径长度。图 4(c)显示当树木数

量增加时，xo 和 xi 的平均路径长度的变化情况（横坐标为树木的个数，纵坐标为平均路径长度）。当使用 1000 棵树时，xo 和 xi 的平均路径长度分别收敛到 4.02 和 12.82。可以看出异常数据的路径长度小于正常的路径长度。

定义：隔离树。设 T 是隔离树的一个节点。T 是没有孩子的外部节点，或者是一个具有测试条件的内部节点，并且有两个子节点(Tl，Tr)。 一个测试条件由属性 q 和分割值 p 组成，使得测试条件 q <p 将数据点分成 Tl 和 Tr。

给定来某个分布的 n 个实例的数据 X = {x1，...，xn}构成的样本，以构建隔离树(iTree)，我们通过随机选择一个属性 q 和一个拆分值 p 来递归地划分 X，直到满足如下某个条件时停止：

（1） 树达到高度限制

（2） | X | = 1

（3） X 中的所有数据具有相同的值。

iTree 是一个合适的二叉树，树中的每个节点都有零个或两个子节点。假设所有实例都是不同的，那么每个实例都可以在 iTree 完全生长时与外部节点隔离开来，这种情况下外部节点的数量是 n，内部节点的数量是 n-1； iTrees 节点的总数是 2n - 1; 因此内存要求是有界的而且仅与 n 成线性增长。

异常检测的任务是提供一个异常排名，从而反映了异常的程度。 因此，检测异常的一种方法是根据路径长度对数据点的异常分数进行排序; 异常点就是排名靠前的数据点。定义路径长度和异常评分如下：

定义：路径长度。点 x 的路径长度 h(x)由从根节点遍历到 iTree 的某个终止的外部节点访问过的边数决定。

异常评分：从 h(x)得出这样一个分数的困难在于当 iTree 的最大可能高度随着 n 增长时，平均高度就会按照 log n 的速度依次增长。**对** h(x)进行标准化时，会导致树高没有界限，从而不能直接比较。

由于 iTrees 与二叉搜索树(BST)具有相同的结构，外部终端节点的 h(x)的平均值估计就与 BST 中的不成功搜索次数相同。我们借用 BST 分析来估计 iTree 的平均路径长度。对于一个 n 个实例的数据集， BST 中不成功搜索的平均路径长度为：

$$c(n) = 2H(n - 1) - (2(n - 1)/n),$$

其中 H(i)是谐波数并且可以通过通过 ln(i)+ 0.5772156649(欧拉常数)来估计。由于 c(n)是给定 n 的 h(x)的平均值，我们用它来规范 h(x)。该实例 x 的异常评分 s 被定义为：

$$\mathrm{S(x, n)} = 2^{-\frac{E(h(x))}{c(n)}}$$

其中 E(h(x))为一组隔离树的 h(x)的平均值。$\mathrm{S(x, n)}$有如下特性：

当 E(h(x)) → c(n), s → 0.5;

当 E(h(x)) → 0, s → 1;

当 E(h(x)) → n - 1, s → 0.

s 对 h(x)是单调的。图 5 说明了在 E(h(x))和 s 之间的这种关系，并且应用以下条件，其中当 0 <s≤1 时，0 <h(x)≤n-1。使用异常评分 s，我们可以做出以下评估：

(1)如果实例返回分数非常接近 1，那么它们绝对是异常的。

(2)如果实例返回的分数远小于 0.5，那么他们被认为是正常。

(3)如果所有的实例返回 s≈0.5，那么整个样本实际上没有任何明显的异常。

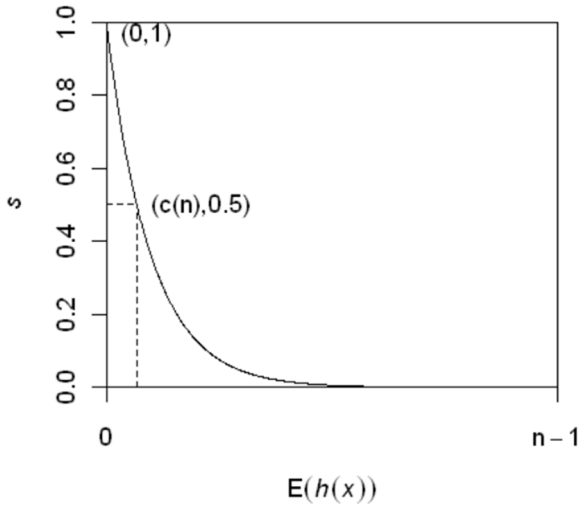


图 5　E(h(x))和 s 之间的关系

异常分数的轮廓可以通过一组隔离树对一组样本进行打分得到。图 6 显示了这种轮廓的一个例子，允许用户使用可视化技术来识别实例空间中的异常。使用轮廓，我们可以清楚地识别出有三个点，它们的 s≥0.6，这是潜在的异常。

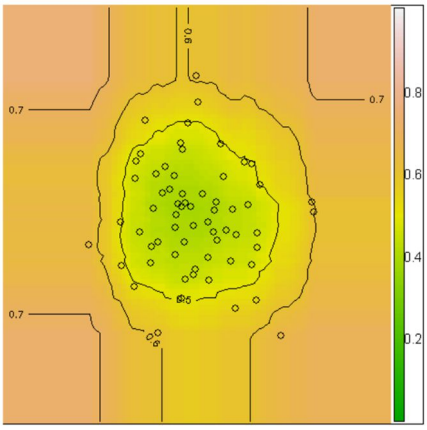


图 6 异常轮廓实例。图 6 绘制出了 iForest 的异常评分等高线，并给出了等高线为 s =0.5,0.6,0.7 的情况。潜在的异常可以确定为 s≥0.6 的点。

隔离树的特点

本小节介绍 iTree 的特性。作为一种集成学习方法，隔离森林有如下特点：

(1)将异常识别为具有较短路径长度的点。

(2)具有多个作为“专家”的树，以针对不同的异常。

由于 iForest 不需要隔离所有正常情况——也就是大部分的训练数据，iForest 能够很好地使用采样策略构建模型。

与现有的方法不同，隔离的方法在采样量很小的情况下仍然能够有良好的表现。因此，隔离森林采用子采样算法。

图 7(a)显示了一个具体的数据集。数据集有两个异常集群，位于靠近一大群正常点的地方。在异常集群周围存在着部分干扰点，而且异常集群比正常集群更

密集。图 7(b)显示了 128 个原始数据实例的子样本。 这些异常在子样本中可以清楚地识别出来。在两个异常集群周围的正常实例已经被清除，异常集群的大小变得更小，这使得它们更容易被区分。

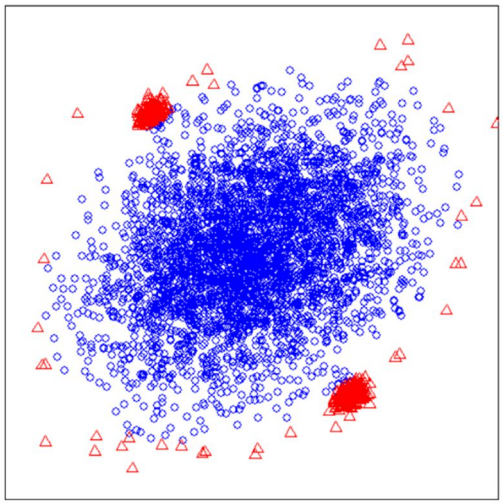


图 7(a) 原始数据集(4096 个样本)

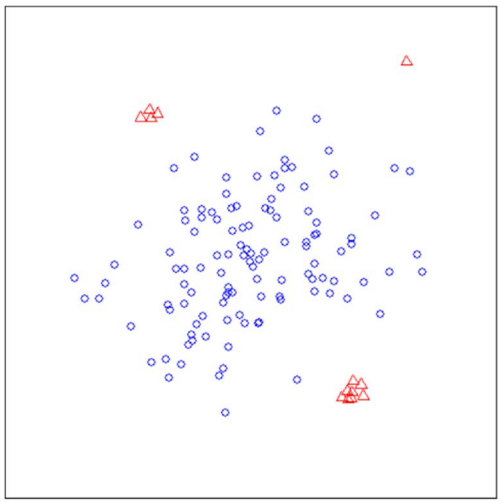


图 7(b) 采样数据集

算法实现

使用 iForest 进行异常检测是一个两阶段过程。第一阶段(训练阶段)使用训练集的子样本构建隔离树。第二个(测试)阶段将测试实例通过隔离树来获取每个实例的异常分数。

在训练阶段，隔离树通过递归地划分给定训练集来构建，直到全部的实例都被隔离或者到达特定的树高而停止。

请注意，树高度限制 l 由子采样大小 ψ 自动设置：l = ceiling(log2 ψ)，这大约是树高大小的平均值。生长树木达到平均树高的原理就是这样，我们只对具有比平均路径长度短的数据点感兴趣，因为这些点更可能是异常。

Algorithm : *iForest*(*X, t, ψ*)
Inputs: *X* – 输入数据, *t* – 树的个数, *ψ* – 子采样大小 **Output**: 一组 *iTrees* 1: **Initialize** *Forest* 2: 设置高度限制 *l* = *ceiling*(log2 *ψ*) 3: **for** *i* = 1 to *t* **do** 4: *X0* ← *sample*(*X, ψ*) 5: *Forest* ← *Forest* ∪ *iTree*(*X0,* 0, *l*) 6: **end for** 7: **return** *Forest*

iForest 算法有两个输入参数。它们是子采样大小 ψ 和树的个数 t。

我们在下面提供了一个指南，可以为两个参数选择合适的值。

(1) 子采样大小 ψ 控制训练数据大小。我们发现，当 ψ 增加到所需值时，iForest 可以进行可靠地检测，并且不需要进一步增加 ψ，因为它增加了处理时

间和内存大小，而且并没有带来检测性能的任何增益。通过实际经验，我们发现将 ψ 设置为 28 或 256 通常可以提供足够的详细信息，以便在广泛的数据范围内执行异常检测。除非另有说明，否则我们使用 ψ = 256 作为我们的实验的默认值。

(2)树的数量 t。我们发现，路径长度通常在 t = 100 之前收敛。因此除非另有说明，一般在实验中使用 t = 100 作为默认值。

Algorithm : *iTree*(*X*,*e*,*l*)
Inputs: *X* – 输入数据， e- 当前树高， *l* – 高度限制 **Output:** 一棵 iTree 树 1: **if** $e \geq l$ or $\|X\| \leq 1$ **then** 2: return *exNode{Size* ← $\|X\|$*}* 3: **else** 4: 令 *Q* 为 X 中的一组属性 5:随机选择一个属性 $q \in Q$ 6: 对于属性 q,随机在 max 和 min 之间选一个划分点 p。 7: *Xl* ← *filter*(*X*,*q* < *p*) 8: *Xr* ← *filter*(*X*,*q* ≥ *p*) 9: return *inNode{Left* ← *iTree*(*Xl*,*e* + 1,*l*), 10: *Right* ← *iTree*(*Xr*,*e* + 1,*l*), 11: *SplitAtt* ← *q*, 12: *SplitV alue* ← *p}* 13: **end if**

Algorithm : *PathLength*(*x*,*T*,*e*)
Inputs : *x* ： 一个实例， *T* – 一个 iTree, *e* –当前路径长度，首次调用时初始化为 0 **Output**: x 的路径长度 1: **if** *T* 是一个外部节点 **then** 2: return *e* + *c*(*T.size*) *{ c*(*n*) = 2*H*(*n* - 1) - (2(*n* - 1)/*n*) *}* 3: **end if** 4: *a* ← *T.splitAtt* 5: **if** *xa* < *T.splitValue* **then** 6: return *PathLength*(*x*,*T.left*,*e* + 1) 7: **else** *{xa* ≥ *T.splitValue}* 8: return *PathLength*(*x*,*T.right*,*e* + 1) 9: **end if**

训练过程结束时，返回所有的树木并准备好进行异常探测。iForest 算法训练的复杂度为 $O(t\psi \log \psi)$。

应用实例

在 python sklearn 库中已有写好的隔离森林程序供我们使用。函数原型如下：

IsolationForest(n_estimators=100, max_samples='auto', contamination=0.1, max_features=1.0, bootstrap=False, n_jobs=1, random_state=None, verbose=0)

各个参数的说明如下：

n_estimators	树的个数
max_samples	每棵树的最大采样大小
contamination	数据集中异常数据所占的比例
max_features	采样时用到的最大特征数
bootstrap	采样时是否有放回
n_jobs	并行作业的数目
random_state	用于实现采样的随机性
verbose	控制树构建过程中的冗余

表 1 隔离森林函数原型所用参数

我们给出一个应用算法的实例（程序来自 sklearn 官网）：

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import IsolationForest
rng = np.random.RandomState(42)
# 生成训练数据
X = 0.3 * rng.randn(100, 2)
X_train = np.r_[X + 2, X - 2]
# 生成一些新颖值
X = 0.3 * rng.randn(20, 2)
X_test = np.r_[X + 2, X - 2]
# 生成异常值
X_outliers = rng.uniform(low=-4, high=4, size=(20, 2))
# 训练模型
clf = IsolationForest(max_samples=100, random_state=rng)
clf.fit(X_train)
y_pred_train = clf.predict(X_train)
y_pred_test = clf.predict(X_test)
y_pred_outliers = clf.predict(X_outliers)
# 绘制图形
xx, yy = np.meshgrid(np.linspace(-5, 5, 50), np.linspace(-5, 5, 50))
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.title("IsolationForest")
```

```
plt.contourf(xx, yy, Z, cmap=plt.cm.Blues_r)
b1 = plt.scatter(X_train[:, 0], X_train[:, 1], c='white',
                 s=20, edgecolor='k')
b2 = plt.scatter(X_test[:, 0], X_test[:, 1], c='green',
                 s=20, edgecolor='k')
c = plt.scatter(X_outliers[:, 0], X_outliers[:, 1], c='red',
                s=20, edgecolor='k')
plt.axis('tight')
plt.xlim((-5, 5))
plt.ylim((-5, 5))
plt.legend([b1, b2, c],
           ["training observations",
            "new regular observations", "new abnormal observations"],
           loc="upper left")
plt.show()
```

生成的结果如图所示：

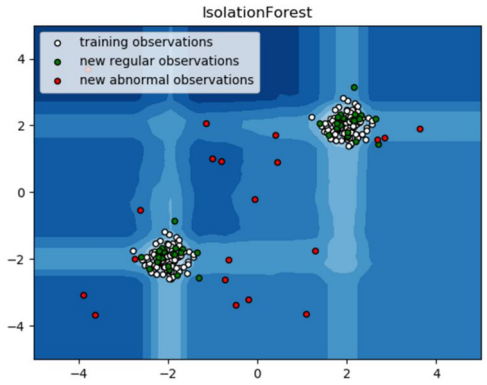


图 8 隔离森林算法程序实例

小结

iForest 广泛的应用于异常数据挖掘问题，如网络安全中的攻击检测和流量异常分析，金融机构欺诈行为检测等。算法对内存要求很低，且处理速度很快，其时间复杂度也是线性的。可以很好的处理高维数据和大数据，并且也可以作为在线异常检测。

通过本节的介绍，相信读者能对这种新颖的异常检测算法有更全面的认识。

LOF：局部异常因子算法

局部异常因子算法是异常领域十分经典的算法。LOF 算法从一个全新的角度对待异常检测问题：现实中的一些异常是与其邻居比较疏远的点，或者说与邻居的密度不同。

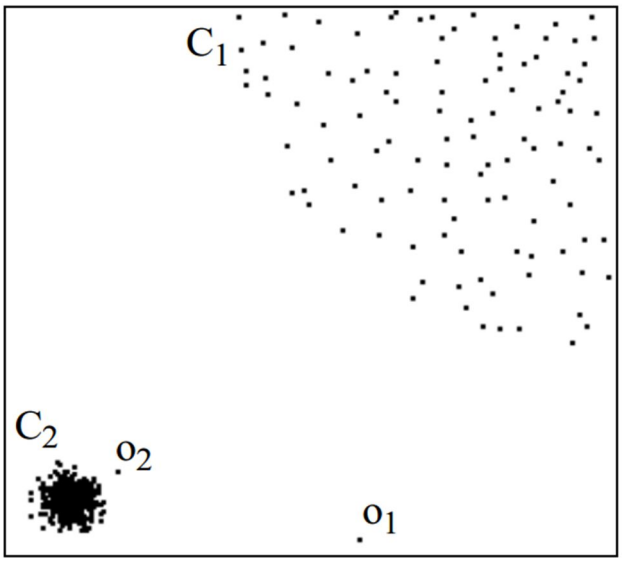


图 9 一个二维数据点实例

如图 9 所示，对于 C1 集合中的点，整体间距，密度，分散情况较为均匀一致，可以认为这些点属于同一个簇；对于 C2 集合的点，同样可认为是一簇。o1、o2 点相对孤立，可以认为是异常点或离散点。现在的问题是，如何实现算法的通用性，可以满足对 C1 和 C2 这种密度分散情况迥异的集合的异常点识别。LOF 可以解决这一问题。

基本定义

下面我们给出一系列的基本定义和概念：

定义 1：Hawkins-异常

异常值是一个与其他值偏差很大的观察值，这使得我们怀疑它是由一个不同的机制产生的。

定义 2：DB(pct, dmin)-异常

数据集 D 中的对象 p 是 DB(pct，dmin)-异常当且仅当与 p 距离大于 dmin 的对象的百分比为 pct，即集合{q∈D|d(p，q)≤dmin}的大小小于或等于 D 的大小乘以(100 - pct)％。

例如，数据集中与 p 距离 > 0.1 的对象所占的百分比大于 60%，则称 p 是 DB(0.6，0.1)异常。

定义 3：d(o,p)

两点 o 和 p 之间的距离.

定义 4：对象 p 的 k 距离

对于任意正整数 k，p 的 k 距离表示为 *k-distance*(*p*)，被定义为在 p 和一个对

象 o∈D 之间的距离，满足：

(1)在集合中至少有不包括 p 在内的 k 个点 o’ ∈D\{p}, 满足 d(p,o’,)≤d(p,o) ；

(2) 在集合中最多有不包括 p 在内的 k−1 个点 o’ ∈D\{p},，满足 d(p,o,)<d(p,o) ；

定义 5：对象 p 的 k 距离邻域

给定 p 的 k 距离，p 的 k 距离邻域包含与 p 的距离不大于 k 距离的每个对象，也就是 Nk-distance(p)(p) = { q ∈ D\{p} | d(p, q) ≤ k-distance(p) }.

这些对象 q 被称为 p 的 k 最近邻居。

定义 6：对象 p 的对象的可达距离

设 k 是一个自然数。对象 p 关于对象 o 的可达距离被定义为

reach-distk(p，o)= max {k-distance(o)，d(p，o)}

图 10 说明了 k = 4 时可达距离的概念。直观地讲，如果对象 p 远离 o(例如图中的 p2)，那么两者之间的可达距离仅仅是它们的实际距离。然而，如果它们“足够”地靠近(例如，图中的 p1)，则实际距离被 o 的 k 距离替代。这样做的原因是，所有的 d(p，o)的统计波动可以显着减小。这个平滑效果可以通过参数 k 来控制。k 的值越大，在同一邻域内对象的可达距离越相似。

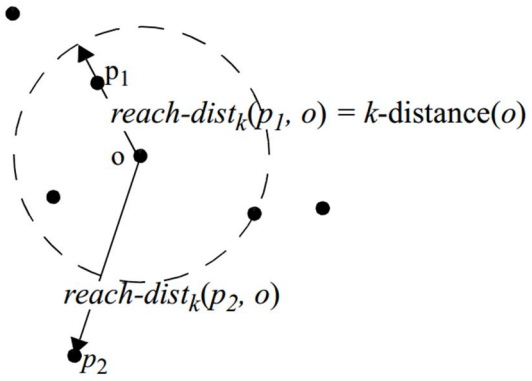


图 10　K=4 时 reach-dist(p1,o) 和 reach-dist(p2,o)

定义 7：一个对象 p 的局部可达密度

P 的局部可达密度定义为：

$$lrd_{MinPts}(\mathrm{p}) = 1/\left(\frac{\sum_{o\in N_{MinPts(p)}} reach-dist_{MinPts}(p,o)}{|N_{MinPts}(p)|}\right)$$

直观上，对象 p 的局部可达密度是基于 p 的 MinPts 邻居的平均可达距离的倒数。

这个值的含义可以这样理解，首先这代表一个密度，密度越高，我们认为越可能属于同一簇，密度越低，越可能是离群点。如果 p 和周围邻域点是同一簇，导致可达距离之和较小，密度值较高；如果 p 和周围邻居点较远，那么可达距离可能都会取较大值，导致密度较小，越可能是离群点。

定义 8: 对象 p 的局部异常因子

对象 p 的局部异常因子定义为：

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

对象 p 的局部异常因子刻画了 p 的异常程度。如果这个比值越接近 1，说明

p 的邻域点密度差不多，p 越可能和邻域同属一簇；如果这个比值越小于 1，说明 p 的密度高于其邻域点密度，p 为密集点；如果这个比值越大于 1，说明 p 的密度小于其邻域点密度，p 越可能是异常点。

异常检测

基于以上定义，LOF 通过比较每个点 p 和其邻域点的密度来判断该点是否为异常点：点 p 的密度越低，越可能被认定是异常点。

而点的密度，是通过点之间的距离来计算的，点之间距离越远，密度越低，距离越近，密度越高。因为 LOF 的密度通过点的 k 邻域来计算，而不是全局计算，因此称为局部异常因子。

应用实例

在 python sklearn 库中已有写好的 LOF 程序包供我们使用。

我们给出一个算法应用实例（程序来自 sklearn 官网）：

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import LocalOutlierFactor
np.random.seed(42)
# 生成训练数据
X = 0.3 * np.random.randn(100, 2)
# 生成新颖值
X_outliers = np.random.uniform(low=-4, high=4, size=(20, 2))
X = np.r_[X + 2, X - 2, X_outliers]
# 构建模型
clf = LocalOutlierFactor(n_neighbors=20)
y_pred = clf.fit_predict(X)
y_pred_outliers = y_pred[200:]
# 绘制边界
xx, yy = np.meshgrid(np.linspace(-5, 5, 50), np.linspace(-5, 5, 50))
Z = clf._decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.title("Local Outlier Factor (LOF)")
```

```
plt.contourf(xx, yy, Z, cmap=plt.cm.Blues_r)
a = plt.scatter(X[:200, 0], X[:200, 1], c='white',edgecolor='k',
s=20)
b = plt.scatter(X[200:, 0], X[200:, 1], c='red',edgecolor='k', s=
20)
plt.axis('tight')
plt.xlim((-5, 5))
plt.ylim((-5, 5))
plt.legend([a, b], ["normal observations", "abnormal observations"],loc
="upper left")
plt.show()
```

程序运行完得到 11 的结果。

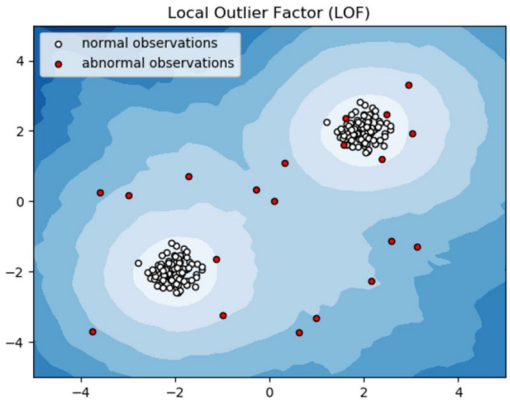


图 11 程序运行结果

小结

可以看出，局部异常因子提供了“基于密度”的思想去探测异常，其基本思想认为，如果一个点与其邻居相差较多，则视为异常。

LOF 算法适用于基于不同密度的数据，此时可以得出每个对象的 LOF 值，来判断其是否为一个异常点。但此方法也具有一定的局限性，并不能适用于所有的场景，此外，它的计算也相对较为复杂。

One-class svm

本节要介绍的 one-class svm 也是异常探测领域十分经典的算法，它的思想能够代表了一类异常算法：用正常数据刻画正常数据的轮廓，那么不符合这个轮廓的数据就会被视为异常。

举一个网络安全的实例：在网络安全问题中，各种网络入侵花样百出，如果要针对这些异常（入侵事件）建立学习模型会面临两个问题：

（1） 异常样本有限（入侵事件的个数不是很大）。

（2） 未来的异常更可能是新的，从来没有见过的异常。

这样依赖异常样本的方法就会很被动，所以许多异常探测算法提出刻画正常数据的轮廓进行异常探测。

基本原理

一类分类（one-class）问题是为了找到一个超平面而制定的，这个超平面能够将所需的一部分训练模式从特征空间的源 F 中分离出来。

这个超平面不能在原始特征空间中被找到，因此我们需要一个映射函数：

$\Phi : F \rightarrow F'$，将 F 映射到核空间 F'。当使用高斯核函数时，可以证明我们总能够找到这种超平面。

高斯核函数可以表示为：

$$K(x, y) = \Phi(x) \cdot \Phi(y) = \exp(-\gamma||x - y||^2)$$

问题就可以形式化的表示为：

$$min_{w,\xi,\rho}(\frac{1}{2}||w||^2 - \rho + \frac{1}{mC}\sum_i \xi_i)$$

$$w.\ \Phi(x_i) \geq \rho - \xi_i,\ \xi_i \geq 0, \forall i = 1, \cdots, m$$

其中，w 是超平面的正交向量，C 表示允许被拒绝的训练模式的比例（也就是说这部分训练模式没能被超平面分离），x_i是第 i 个训练模式，m 是训练模式的总数，$\xi = |\xi_i, \cdots, \xi_m|$是一组松弛向量用来惩罚拒绝模式。$\rho$表示间隔，也就是超平面和源的距离。

上述问题的解就对应着一个决策函数，对于一个测试模式 z，可以定义为：

$$f_{svc}(z) = I(\sum_i \alpha_i K(x_i, z) \geq \rho)$$

$$其中 \sum_{i=1}^{m} \alpha_i = 1$$

这里 I 是指示函数，如果 x 为 true,那么 I(x) = 1,否则为 0.

可以看出，一个模式 z 要么被拒绝，判定为 0；要么被接受，判定为 1。

因为训练样本只有一类，因此函数的目标不再是找到正负样本的最大间隔，而是寻找与源距离最大的间隔ρ，从而刻画出训练数据的轮廓。如果一个测试样本被接收，那么视为正常，被拒绝，则视为异常。

应用实例

在 python sklearn 库中已有写好的 one-class svm 程序包供我们使用。实际调参时，大家可以参考一般 svm 算法调参的方案。

我们给出一个算法应用实例（程序来自 sklearn 官网）：

```
import numpy as np
import matplotlib.pyplot as plt
```

```
import matplotlib.font_manager
from sklearn import svm
xx, yy = np.meshgrid(np.linspace(-5, 5, 500), np.linspace(-5, 5,
 500))
# 产生训练数据
X = 0.3 * np.random.randn(100, 2)
X_train = np.r_[X + 2, X - 2]
# 产生新颖值
X = 0.3 * np.random.randn(20, 2)
X_test = np.r_[X + 2, X - 2]
# 构建异常样本
X_outliers = np.random.uniform(low=-4, high=4, size=(20, 2))
# 构建模型
clf = svm.OneClassSVM(nu=0.1, kernel="rbf", gamma=0.1)
clf.fit(X_train)
y_pred_train = clf.predict(X_train)
y_pred_test = clf.predict(X_test)
y_pred_outliers = clf.predict(X_outliers)
n_error_train = y_pred_train[y_pred_train == -1].size
n_error_test = y_pred_test[y_pred_test == -1].size
n_error_outliers = y_pred_outliers[y_pred_outliers == 1].size
#绘制结果和异常边界
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.title("Novelty Detection")
plt.contourf(xx, yy, Z, levels=np.linspace(Z.min(), 0, 7), cmap=
plt.cm.PuBu)
a = plt.contour(xx, yy, Z, levels=[0], linewidths=2, colors='dar
kred')
plt.contourf(xx, yy, Z, levels=[0, Z.max()], colors='palevioletr
ed')
s = 40
```

```
b1 = plt.scatter(X_train[:, 0], X_train[:, 1], c='white', s=s, ed
gecolors='k')
b2 = plt.scatter(X_test[:, 0], X_test[:, 1], c='blueviolet', s=s,
                 edgecolors='k')
c = plt.scatter(X_outliers[:, 0], X_outliers[:, 1], c='gold', s=
s,
                edgecolors='k')
plt.axis('tight')
plt.xlim((-5, 5))
plt.ylim((-5, 5))
plt.legend([a.collections[0], b1, b2, c],
           ["learned frontier", "training observations",
            "new regular observations", "new abnormal observations"],
           loc="upper left",
           prop=matplotlib.font_manager.FontProperties(size=11))
plt.xlabel(
    "error train: %d/200 ; errors novel regular: %d/40 ; "
    "errors novel abnormal: %d/40"
    % (n_error_train, n_error_test, n_error_outliers))
plt.show()
```

运行结果如图 12 所示：

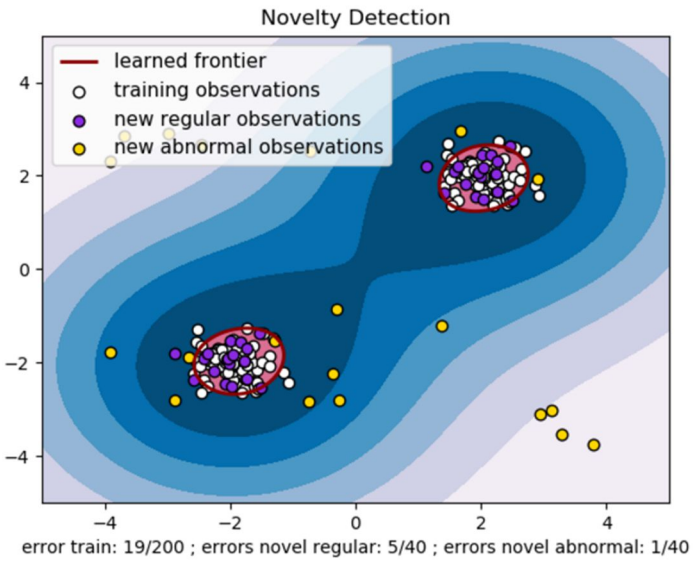


图 12： one-class svm 算法应用实例

小结

描述正常数据的轮廓进行异常探测，能够起到以不变应万变的作用，因而是异常探测领域十分重要的思想。

One-class svm 为了实现这一思想，将原来的 svm 求解正负样本最大间隔的目标进行改造，实现了异常探测的功能。

One-class svm 有能力捕获数据集的形状,因此具有良好的性能。严格来说，One-class svm 并不是一个异常点检测算法，而是一个新颖值检测算法：它的训练集不能包含异常样本，否则的话，可能会在训练模型时影响边界的选取。

对于高维空间中的样本数据集，One-class SVM 能表现出更好的优越性。

隔离森林算法、LOF 算法和 One-class svm 算法比较

前面三个小节中，我们分别介绍了隔离森林算法、LOF 算法和 One-class svm 算法，这一节我们通过实验来简单比较三者的特点。假设我们已经知道数据中的异常所占的比例（contamination）。程序如下（改编自 sklearn 官网）：

```
#coding:utf-8
import numpy as np
from scipy import stats
import matplotlib.pyplot as plt
import matplotlib.font_manager
from sklearn import svm
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor
#设置随机数
rng = np.random.RandomState(42)
#生成模拟数据
n_samples = 200
outliers_fraction = 0.25
clusters_separation = [0, 1, 2]
#生成比较算法
classifiers = {
    "One-Class SVM": svm.OneClassSVM(nu=0.95 * outliers_fraction +
0.05,
                                     kernel="rbf", gamma=0.1),
    "Isolation Forest": IsolationForest(max_samples=n_samples,
                                        contamination=outliers_fraction,
                                        random_state=rng),
    "Local Outlier Factor": LocalOutlierFactor(
        n_neighbors=35,
        contamination=outliers_fraction)}

#比较各个探测器
```

```
xx, yy = np.meshgrid(np.linspace(-7, 7, 100), np.linspace(-7, 7,
100))
n_inliers = int((1. - outliers_fraction) * n_samples)
n_outliers = int(outliers_fraction * n_samples)
ground_truth = np.ones(n_samples, dtype=int)
ground_truth[-n_outliers:] = -1

for i, offset in enumerate(clusters_separation):
    np.random.seed(42)
    #产生数据
    X1 = 0.3 * np.random.randn(n_inliers // 2, 2) - offset
    X2 = 0.3 * np.random.randn(n_inliers // 2, 2) + offset
    X = np.r_[X1, X2]
    #添加异常点
    X = np.r_[X, np.random.uniform(low=-6, high=6, size=(n_outliers,
2))]
    #生成模型
    plt.figure(figsize=(9, 7))
    for i, (clf_name, clf) in enumerate(classifiers.items()):
        #学习数据并标记异常
        if clf_name == "Local Outlier Factor":
            y_pred = clf.fit_predict(X)
            scores_pred = clf.negative_outlier_factor_
        else:
            clf.fit(X)
            scores_pred = clf.decision_function(X)
            y_pred = clf.predict(X)
        threshold = stats.scoreatpercentile(scores_pred,
                                            100 * outliers_fraction)
        n_errors = (y_pred != ground_truth).sum()
        #绘图
        if clf_name == "Local Outlier Factor":
            #对LOF算法需要特殊处理
            Z = clf._decision_function(np.c_[xx.ravel(), yy.ravel()])
        else:
            Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
        Z = Z.reshape(xx.shape)
        subplot = plt.subplot(2, 2, i + 1)
        subplot.contourf(xx, yy, Z, levels=np.linspace(Z.min(),
threshold, 7),
                         cmap=plt.cm.Blues_r)
        a = subplot.contour(xx, yy, Z, levels=[threshold],
                            linewidths=2, colors='red')
        subplot.contourf(xx, yy, Z, levels=[threshold, Z.max()],
```

```
                          colors='orange')
        b = subplot.scatter(X[:-n_outliers, 0], X[:-n_outliers, 1],
c='white',
                            s=20, edgecolor='k')
        c = subplot.scatter(X[-n_outliers:, 0], X[-n_outliers:, 1],
c='black',
                            s=20, edgecolor='k')
        subplot.axis('tight')
        subplot.legend(
            [a.collections[0], b, c],
            ['learned decision function', 'true inliers', 'true
outliers'],
            prop=matplotlib.font_manager.FontProperties(size=10),
            loc='lower right')
        subplot.set_xlabel("%d. %s (errors: %d)" % (i + 1, clf_name,
n_errors))
        subplot.set_xlim((-7, 7))
        subplot.set_ylim((-7, 7))
    plt.subplots_adjust(0.04, 0.1, 0.96, 0.94, 0.1, 0.26)
    plt.suptitle("Outlier detection")
plt.show()
```

运行结果如下：

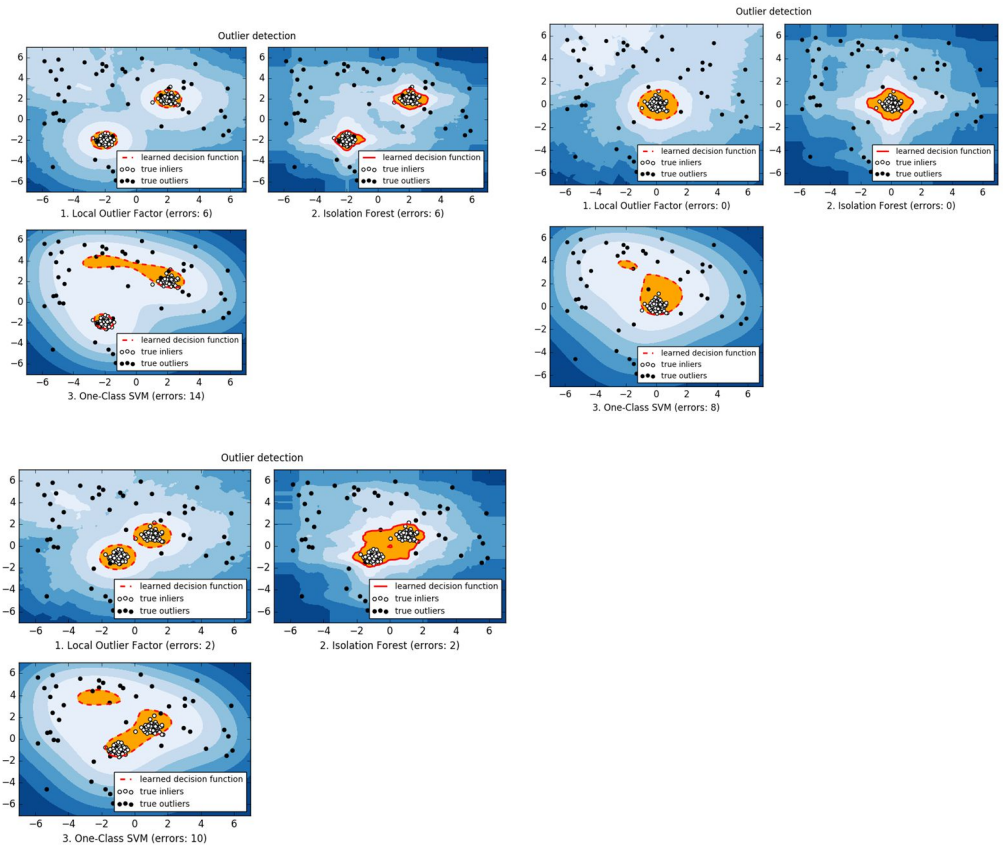


图 13 对比实验

比较结果如下：

（1） One-class svm 有能力捕获数据集的形状，因此如果数据是非高斯分布的，就会有更好的表现，比如图中有两个分离良好的簇的情况。

（2） 隔离森林基于随机森林，因此更适应于高维数据的环境，即使在这个例子中也表现很好。

通过图中“错误”和“决策边界”的信息，我们对于这三种算法会有更为清晰的了解和直观的认识。

将 PCA 应用到异常探测

Principal Component Analysis（PCA）是常见的数据降维的方法。作为一种降维方法，PCA 可以将原数据进行线性变换，并找出数据中信息含量最大的主要成分，去除信息含量较低的成分，从而减少冗余，降低噪音。

通常在异常检测问题中，噪音（noise）、离群点（outlier）和 异常值（anomaly）是同一件事情的不同表述。所以，PCA 虽然是一种降维方法，但是因为它可以识别噪音，所以被广泛的应用于异常探测问题中。

PCA 的过程我们就不再赘述，我们关注如何将 PCA 算法应用到异常检测问题中。

基本原理

PCA 应用在异常检测方面的方法，主要有两种思路：

(1) 将数据映射到低维特征空间，然后在特征空间的不同维度上查看每个数据跟其它数据的偏差；

(2) 将数据映射到低维特征空间，然后由低维特征空间重新映射到原空间，尝试用低维特征重构原始数据，比较重构误差的大小。

我们先来介绍第一种思路。PCA 在做特征值分解之后得到的特征向量反应了原始数据方差变化程度的不同方向，特征值为数据在对应方向上的方差大小。所以，最大特征值对应的特征向量为数据方差最大的方向，最小特征值对应的特征向量为数据方差最小的方向。

原始数据在不同方向上的方差变化反应了其内在特点。如果单个数据样本跟整体数据样本表现出的特点不太一致，比如在某些方向上跟其它数据样本偏离较大，那么这可能就表示该数据样本是一个异常点。

形式化的来说：

对于某个特征向量e_j，数据样本x_i在该方向上的偏离程度d_{ij}可以用如下公式计算：

$$d_{ij} = \frac{\left(x_i{}^T \cdot e_j\right)^2}{\lambda_j}$$

这里，λ_j表示对应特征向量e_j的特征值，起到了归一化的作用，使得不同方向

上的偏离程度具有可比性。

计算了数据样本在所有方向上的偏离程度之后，为了给出一个综合的异常得分，最自然的做法是将样本在所有方向上的偏离程度加起来，即：

$$\text{Score}(x_i) = \sum_{j=1}^{n} d_{ij} = \sum_{j=1}^{n} \frac{\left(x_i{}^T \cdot e_j\right)^2}{\lambda_j}$$

$\text{Score}(x_i)$就是计算异常得分的一种方式，当然不同的算法也有不同的评分方式，这里只是给出一种简单的思路。

这里再介绍一种判定异常的策略，有的算法只考虑前 k 个特征向量上的偏差，这种情况下，当：$\sum_{j=1}^{k} d_{ij} > C$ 时，认为样本x_i为异常。

对于第二种思路，可以从直观上理解，PCA 提取了数据的主要特征，如果一个数据样本不容易被重构出来，表示这个数据样本的特征跟整体数据样本的特征不一致，那么它更有可能是一个异常的样本。

对于样本x_i，假设其基于 k 个特征向量重构得到的样本为x_{ik}，那么我们只需定义“这种重构误差”，就能进行异常探测。可以用如下公式进行计算：

$$\text{Score}(x_i) = \sum_{k=1}^{n} (|x_i - x_{ik}|) \times \text{ev(k)}$$

$$\text{ev(k)} = \frac{\sum_{j=1}^{k} \lambda_j}{\sum_{j=1}^{n} \lambda_j}$$

上面的公式考虑了重构使用的特征向量的个数 k 的影响，将 k 的所有可能做了一个加权求和，得出了一个综合的异常得分。显然，基于重构误差来计算异常得分的公式也不是唯一的。

小结

虽然 PCA 是一种降维方法，但是在异常探测领域应用十分广泛，许多论文研究工作都将 PCA 作为异常探测的一个重要环节。

将降维方法应用到异常探测普遍都有两种方案：

（1） 将高维数据映射到低维空间，根据不同属性上数值的偏差进行异常探测。

（2） 将高维映射为低维，再将低维映射到高维后的还原偏差视为异常得分进行异常探测。

集成学习应用到异常探测

集成学习是一种机器学习方法：它使用一系列学习器进行学习，并使用某种规则把各个学习结果进行整合从而获得比单个学习器更好的学习效果。

集成学习被广泛的应用在监督学习问题中，本节则介绍如何将集成学习应用到异常探测问题中来。与聚类和分类问题相比，集成学习在异常探测领域的研究比较有限。在一些情况下，许多异常检测算法隐含地使用集成学习技术，本节将

介绍针对异常领域的集成学习技术的一般方法论。

基本原理

集成分析是用于提高各种数据挖掘算法准确性的流行方法。集成方法结合了多种算法(基本探测器)的输出，从而创建统一的输出。这种方法的基本思想是有些算法可以在部分数据子集上表现良好，而其他算法在其他数据子集上的效果会更好。但是，集成组合往往通过将多种算法的输出结合起来从而拥有更强大的表现。

典型的异常集成学习算法包含许多不同的子模型，用于构建最终结果。通常涉及三个问题：

（1）模型创建：独立的创建各个子模型，比如采用随机子空间采样的方法。

（2）异常分数规范化：不同的方法可能会产生非常不同的异常值分数。在某些情况下，异常分数可能按升序排列，而在其他情况下，异常分数则可能按降序排列。在这种情况下，有意义的异常分数对于集成学习而言非常重要，应当使异常值得分在不同的子模型上大致相当。

（3）子模型组合：这指的是最后的组合函数，这被用来生成最后的异常得分。

子模型的设计及其组合方法都取决于一个特定的集成学习方法的目标。这方面取决于异常组合分析的基础理论，它将异常值检测的误差分解为两部分，称为偏差（bias）和方差（variance）。

需要注意的是，如果我们知道数据的基本分布，我们就可以按照我们对于数据的知识产生无穷无尽的训练数据。但是，我们无法了解数据的真实分布，我们只能访问训练数据集的单个实例。我们用有限的训练数据集所创建的模型将不可避免地导致错误的发生，因为我们无法获取数据的真实分布。即使我们有无限的数据，所使用的特定模型可能并不适合已有数据的实际分布。模型自身的限制和数据的限制导致了方差和偏差。

方差：假设训练数据集是从基础分布生成的，可以进行异常值打分。通常情况下，分析师只能从数据集的基础分布中抽取和访问有限的数据，因此当使用不同的数据时，即使使用的算法相同，对于同一个数据，得到的异常分值也可能不同。例如，使用 k-近邻算法在两个不同的 100 个训练点构成的集合上表现会有所不同。这种结果的差异来自于不同的训练数据集（来自相同的分布），是模型方差的一种表现。

偏差：注意到一个特定的异常值检测模型可能没有适当地反映“理想”的异常分数，因此算法返回的预期分数将与真实分数不同。这种预期和真实的差距就是偏差。

与一般的集成学习算法相似，子模型之间的关系有两类：

（1） 子模型之间相互依赖。我们称之为顺序集成（*sequential ensembles*）。

（2） 子模型之间相互独立。我们称之为独立集成（*independent ensembles*）。

顺序集成

在顺序集成中，将一个或多个异常值检测算法按顺序应用于全部或部分数据之中。该方法的核心原则在于：每个算法都提供了对数据良好的理解，从而使用修改的算法或数据集来得到更精确的执行结果。因此，数据集或算法可能在顺序执行中被改变。

顺序集成的算法如下：

算法: 顺序集成(数据集:D;基本算法: A1,A2,…,Ar)

Begin
j = 1;
repeat
基于上次的执行结果选择一个算法 Aj；
基于上次的执行结果从数据集 D 中选择一个新的数据集 fj(D)；
将算法 Aj 应用到 fj(D)；
j =*j* + 1;
until(结束条件);
report　过去执行结果的组合所探测到的异常
end

在每次迭代中，可以根据过去的执行结果，使用不同的数据对算法进行连续的改进。函数 fj( )用来创建更合适的数据，这可能来自不同的数据子集选择，属性子集选择或通用数据转换方法。上面的描述使用了非常一般的形式，并且可能有许多特殊情况可以从这个通用框架中实例化。例如，在实际中，可能只将同一个算法应用在连续修改的数据上。

算法: 独立集成 (数据集:D;基本算法: A1,A2,…,Ar)

Begin
j = 1;
repeat
选择一个算法 Aj；
从数据集 D 中创建一个新的数据集 fj(D)；
将算法 Aj 应用到 fj(D)；
j =*j* + 1;
until(结束条件);
report　过去执行结果的组合所探测到的异常
end

在独立集成中，算法的不同实例或数据的不同部分被应用于异常检测算法。或者，可以应用相同的算法，但使用不同的初始化参数，或者在随机算法的情况下设置不同的随机种子。

将这些不同算法执行的结果进行整合以获得更强大的异常探测器。

可以看出，上面介绍的顺序集成和独立集成分别类似于集成学习算法中的 boosting 和 bagging 理论。

算法应用

我们举 2 个应用集成学习理论进行异常探测的例子，它们具有非常优异的性能和启发性。

首先使用 LOF 算法介绍一个能够处理高维数据的算法实现：

算法：LOF 特征 bagging(数据集 D)

begin
repeat
对子空间进行采样，若 D 的维数为 d，每个采样子空间的大小在 2 到 d 之间

对于每个点，使用 LOF 算法得到在投影空间的 LOF 得分 **Until n 次迭代；** 返回从不同子空间整合后的分数 **End**

这个算法可以有效的应对高维数据，通过对数据不同子空间投影点进行异常打分，可以判断数据异常与否。

我们举的第二个例子称为“参数集成”。我们知道在异常探测领域的算法中，往往需要设置大量的参数。比如在隔离森林算法中，我们要设置采样大小，基模型的个数；基于距离的异常探测算法中，我们要设置邻居的个数、异常的阈值等等。

而在实际应用中，这些参数的设定是很困难的，人工经验设置的参数往往表现很差，下面要介绍的“参数集成”算法就提供了这样一个“参数设置自动化”的思路。“参数集成”算法其实是解决方差问题的一个很好的应用。

算法：参数集成(数据集 D；集成子模型的个数：T)

Begin
{算法 A 的参数是 θ1 . . . θr }
t = 1;
for each θi 确定其合理的范围[mini, maxi]
repeat
for each i 从范围[mini, maxi]中随机选择 θi;
通过执行算法 A(D, θ1 . . . θr)计算分数向量 S(t);
对分数向量 S(t)进行标准化;
t = t + 1;
until(t = T);
return 平均后的分数向量$\frac{\sum_{i=1}^{T} S(i)}{T}$
end

可以看出，这种“参数自动化”设置具有很广阔的应用前景，也具有很好的启发性。

应用实例

在本章的习题中，我们也使用 shuttle 数据集进行异常探测，在本小节，我们将介绍使用集成学习进行异常探测的应用实例：参数设置自动化。

Shuttle 数据集的信息如下：

类型	数值型
第 1 列属性	时间
第 2 列属性	Rad Flow
第 3 列属性	Fpv Close
第 4 列属性	Fpv Open
第 5 列属性	High
第 6 列属性	Bypass
第 7 列属性	Bpv Close

第 8 列属性	Bpv Open
第 9 列属性	类别

我们首先统计一下数据信息：

```
# -*- coding: utf-8 -*-
import pandas as pd
import numpy as np
names = [1,2,3,4,5,6,7,8,9,10]
df = pd.read_table('shuttle.txt',sep=' ',names = names)
df.sort_index(by=1)#按照时间排序
df = df.drop([1],axis =1)#去掉时间列
shuttle_class = df[10].values
d = {}
for i in shuttle_class:
    if i in d:
        d[i] += 1
    else:
        d[i] = 1
sum = 0
for i in d:
    print i," : ",d[i]
    sum += d[i]
print "sum:",sum
print "ratio for class 1:", 1.0*(d[1])/sum
```

统计信息如下：

```
1  :  11478
2  :  13
3  :  39
4  :  2155
5  :  809
6  :  4
7  :  2
sum: 14500
ratio for class 1: 0.791586206897
```

可以看出类别 1 的数据占据主要部分，因此我们可以将类别 1 的数据视为正常数据，其余的数据视为异常数据。

现在假设**我们对于要检测的数据中异常数据所占的比例一无所知**，那么除了知道异常是“少而不同的”，有什么能让探测效果较好的方法吗？

如果直接使用 LOF 算法：

```
# -*- coding: utf-8 -*-
import pandas as pd
import numpy as np
def print_metric(o,y):
    TP = 0#真正
    FP = 0#假正
```

```
    TN = 0#真负
    FN = 0#假负
    for i in range(len(o)):
        if o[i] == 1 and y[i] == 1:
            TN += 1
        if o[i] == 1 and y[i] == -1:
            FN += 1
        if o[i] == -1 and y[i] == -1:
            TP += 1
        if o[i] == -1 and y[i] == 1:
            FP += 1
    print "Recall:",TP*1.0/(TP+FN)
    print "Precision:",TP*1.0/(TP+FP)
names = [1,2,3,4,5,6,7,8,9,10]
df = pd.read_table('shuttle.txt',sep=' ',names = names)
df.sort_index(by=1)
shuttle_class = df[10]
y = shuttle_class
for i in range(len(shuttle_class)):
    if shuttle_class[i] != 1:
        y[i] = -1
    else:
        y[i] = 1
y = y.values# -1(异常)和1（正常）构成的点
df = df.drop([1,10],axis =1)
from sklearn.neighbors import LocalOutlierFactor
from sklearn.ensemble import VotingClassifier
clf1 = LocalOutlierFactor(n_neighbors=20,contamination=0.1)
o = clf1.fit_predict(df)
print_metric(o,y)#打印指标
```

运行结果如下：

```
Recall: 0.149900727995
Precision: 0.312413793103
```

注意参数 **contamination 表示对于数据中异常所占比例的先验知识。**Sklearn 中函数默认值为 0.1。

现在我们使用我们介绍过的“参数自动化”算法，选择多个参数进行集成学习，程序代码如下：

```
# -*- coding: utf-8 -*-
import pandas as pd
import numpy as np
def vote(clf1,clf2,clf3,clf4,clf5,df):
    o1 = clf1.fit_predict(df)#探测结果为1表示正常，为-1表示异常
    o2 = clf2.fit_predict(df)
    o3 = clf3.fit_predict(df)
```

```
    o4 = clf4.fit_predict(df)
    o5 = clf5.fit_predict(df)
    o = []#保存投票探测结果
    for i in range(len(o1)):
        if o1[i] + o2[i] + o3[i] + o4[i] + o5[i] >= 0:#相当于投票
            o.append(1)
        else:
            o.append(-1)
    return o
def print_metric(o,y):
    TP = 0#真正
    FP = 0#假正
    TN = 0#真负
    FN = 0#假负
    for i in range(len(o)):
        if o[i] == 1 and y[i] == 1:
            TN += 1
        if o[i] == 1 and y[i] == -1:
            FN += 1
        if o[i] == -1 and y[i] == -1:
            TP += 1
        if o[i] == -1 and y[i] == 1:
            FP += 1
    print "Recall:",TP*1.0/(TP+FN)
    print "Precision:",TP*1.0/(TP+FP)
names = [1,2,3,4,5,6,7,8,9,10]
df = pd.read_table('shuttle.txt',sep=' ',names = names)
df.sort_index(by=1)
shuttle_class = df[10]
y = shuttle_class
for i in range(len(shuttle_class)):
    if shuttle_class[i] != 1:
        y[i] = -1
    else:
        y[i] = 1
y = y.values# -1(异常)和1（正常）构成的点
df = df.drop([1,10],axis =1)
from sklearn.neighbors import LocalOutlierFactor
#构造具有不同参数的探测器
clf1 = LocalOutlierFactor(n_neighbors=20,contamination=0.1)
clf2 = LocalOutlierFactor(n_neighbors=20,contamination=0.14)
clf3 = LocalOutlierFactor(n_neighbors=20,contamination=0.18)
clf4 = LocalOutlierFactor(n_neighbors=20,contamination=0.22)
clf5 = LocalOutlierFactor(n_neighbors=20,contamination=0.26)
```

```
o = vote(clf1,clf2,clf3,clf4,clf5,df)
print_metric(o,y)#打印指标
```

我们对contamination采样了5个参数，采样间隔为0.04，最后对5个探测器的结果进行投票。运行结果如下：

```
Recall: 0.252150893448
Precision: 0.291954022989
```

可以看出，召回率有显著提升，而精度有所损失。

小结

本节介绍了将集成学习应用到异常探测问题的方法论，分别介绍了顺序集成和独立集成。

大多数异常检测算法都能输出异常得分，比如我们介绍的隔离森林算法和LOF 算法。通过集成学习有效的处理这些异常得分，将多种异常探测模型进行集成，能够获得更强大的异常探测器。

监督学习异常探测

本节我们来探讨使用监督学习进行异常探测的方法。前面几节我们主要探讨了非监督学习进行异常探测的问题。

那么既然已经有许多优秀的无监督学习异常探测算法，为什么要引入监督学习算法呢？

据观察，在不同的应用中，例如系统异常检测，财务欺诈和 Web 机器人检测，在这样的案例中，一个无监督的异常值检测方法可能会发现噪声，而这些噪声并不是分析师感兴趣的异常事件。在很多情况下，可能会出现多个不同的异常类型的实例，并且可能需要区分他们。例如，在入侵检测场景中，可能存在不同类型的入侵事件，并且具体的入侵类型是很重要的信息。

监督异常探测算法的目标是增强学习方法和特定领域知识之间的联系，以获取与应用相关的异常。这种知识常常包含相关异常的例子。由于异常情况比较少见，甚至罕见，这种与异常相关的例子往往是有限的。这就会为创建模型带来挑战。尽管如此，即使只有少量的数据可用于监督学习，通常情况下也可以通过一定技术提高异常值检测的准确性。异常分析的一般建议是：*总是尽可能的使用监督学习。*

监督学习由正常数据和异常数据来建模。这些例子作为训练数据，并且可以用来创建分类模型，从而区分正常和异常情况。

那么基于监督学习的异常检测问题与分类问题有什么不同呢？监督异常值检测问题可能被认为是一个非常困难的特殊情况的分类问题。这是因为这个问题与几个具有挑战性的特征相关，这些特征可能以孤立的方式或组合方式存在：

类别不平衡：由于异常值被定义为数据中罕见的实例，因此很自然的就会有这样的情况：正常类别和少数类别之间数据的分配将非常不平衡。从实践的角度来看，这意味着对分类精度进行优化可能是无意义的，特别是正常（异常）实例的错误分类和异常（正常）实例的错误分类意义相差很大。换句话说，假正比假负更容易被接受。这导致分类问题就有代价敏感的性质，因此优化函数都基于代

价敏感策略。例如，在某些应用场景下，“宁可误杀一百，也不放过一个”，有很多假正是可以被接受的，但是漏报（存在异常，但是没有探测出来）是不被允许的。

受污染的正常类别的样本（正向未标记分类问题）：在许多实际情况下，只有正类样本被标记，剩下的“正常”数据中包含一些异常数据。这在像 Web 和社交网络这样的大规模环境中很常见，其中基础数据的庞大数量使得正常类别的数据更有可能被污染。例如，考虑一个社交网络应用程序，期望确定社交网络反馈中的垃圾邮件。一小部分文件可能是垃圾邮件。在这种情况下，识别和标记一些文件为垃圾邮件是可行的，但许多垃圾邮件文件可能留在正常类别的样本中。因此，“正常”类别也被认为是一个无标签的类。然而，在实践中，无标签类主要是正常类，其中的异常可能被视为污染物。从技术上讲，这种情况可以被看作是监督学习的困难特例，也就是说正常类别数据中混有噪声和污染。只要污染数据的比例较小，可以使用现成的分类器来处理。

部分训练信息（半监督或新颖值发现）：在许多应用中，存在一些新异常，而我们没有已知对应的实例。例如，在入侵检测应用程序中，可能有一些正常类和入侵类的例子，但是随着时间的推移，就会有新类型的入侵出现。在某些情况下，我们拥有一个或多个正常类的示例。一个特别常见的研究案例是 one-class 的一类变体，只有正常数据可用（比如我们已经讲过的 one-class svm）。 这个特殊的情况下，其中训练数据只包含正常类，更接近无监督版本的异常值检测问题。

上述这些场景可能会以组合的方式出现，并且两者之间的界限也可能是模糊不清的。本节我们将讲述如何**修改**已有的分类算法，使之适用于异常检测问题。

罕见类别检测

罕见类别检测或类别失衡的问题在监督学习异常检测问题中是很常见的。直接使用评估指标和分类器，而没有认识到这种类别的不平衡就可能会得到非常令人惊讶的结果。例如，考虑一种医学应用，希望从医学扫描的结果中鉴别肿瘤。在这种情况下，99％的情况下检测结果可能是正常的，只有剩下 1％的情况是异常的。考虑一般的分类算法，将每个实例都标记为正常，甚至没有检查特征空间。这样的分类器将具有 99％的非常高的准确性，但在实际应用环境中不会有用。

考虑一个 k-近邻分类器。如果测试实例的 k 个最近邻居中的 49％的训练数据是异常的，那么这个实例更可能是异常的。然而由于正常类别的数据在准确性计算中占主导地位，这个实例会被判定为正常，因此分类器将总是具有较低的准确度。

这些情况下适当的评估机制将异常实例的错误与正常实例的错误进行权衡。基本假设是与正常实例相比，将异常实例错误分类的成本更高。例如，欺诈交易的错误分类（可能导致数百万美元的损失）比错误分类正常交易更昂贵（这会导致用户被错误警告）。

代价敏感学习：修改分类算法的目标函数，对于异常类别和正常类别的数据用不同的方式有效权衡分类错误。典型的假设是，将一个罕见类错误的分类会导致更高的成本。在许多情况下，只需对现有的分类模型进行较小的更改就能实现这种目标函数的变化。

自适应重采样：对罕见类别的数据进行重采样以放大稀有类别数据的相对比例。这种方法可以被认为是代价学习的一种间接的形式，因为数据重采样相当于

隐含地假设将罕见类别数据错误分类的成本较高。毕竟，样本中特定类的实例的相对数量较大，探测结果倾向于将预测算法偏向于该类别。

设数据集为$\mathcal{D}$，类别标签用$\mathcal{L}$表示，$\mathcal{L} = \{1, \cdots, \mathrm{k}\}$。不失一般性，我们可以假设正常数据的类别为 1，剩下的类别$2 \cdots, \mathrm{k}$是罕见的类别。属于第 i 类的样本的总数用N_i来表示。因此如果数据集$\mathcal{D}$的大小为 N，则有$\sum_{i=1}^{k} N_i = N$。类别不平衡的假设也就是$N_1 \gg N - N_1$。

代价敏感学习

在代价敏感学习中，目标是学习一个分类器，该分类器能够在不同的类别之间实现加权准确度的最大化。

第 i 个类别的误分类代价 misclassification cost 用c_i表示。许多方法都使用一个 k × k 的代价矩阵来表示误分类的代价（也就是将类别 i 误分类为类别 j 的代价）。这种情况下，代价不仅依赖于误分类实例本身的类别，还依赖于该实例被误分类为其他类的类别。我们现在的目标就是训练一个模型，能够最小化*加权误分类率*。目标函数就可以表示为：

$$\mathrm{J} = \sum_{i=1}^{k} c_i \cdot n_i$$

其中$n_i \leq N_i$，是第 i 类被误分类的样本总数。

与传统的分类准确度度量的区别就在于目标函数中权重c_i的使用。

c_i的选择由特定的问题需要而设定，因此是输入的一部分。c_i的选择有一定的规律可寻，例如一般c_i的值与$\frac{1}{N_i}$成比例。

我们举两个算法实例来展示如何将代价敏感加入到现有的算法中：

贝叶斯分类器

贝叶斯分类器的修改为代价敏感学习提供了最简单的情况。在这种情况下，改变样本的权重只会改变类别的先验概率，贝叶斯分类中的其他内容均保持不变。也就是说，未加权情况下的先验概率需要与代价相乘。

当我们获得了表现良好的贝叶斯分类器，就可以直接用于预测。

决策树

在决策树中，训练数据被递归地划分，以便不同类的实例在树的较低层中被连续分离出来。划分通过使用数据中的一个或多个特征来执行。通常情况下，划分标准使用各种熵度量，如基尼指数来决定选择的属性和分裂的位置。

对于一个节点，它包含不同类别的实例，这些实例的类别用$p_1, p_2 \cdots, p_k$表示，基尼系数可以表示为：$1 - \sum_{i=1}^{k} {p_i}^2$。通过使用代价作为权重，可以影响基尼系数的计算，从而有选择的创建节点，属于罕见类的数据会被给予更高的重视。这种方法通常能够在正常类和异常类之间得到良好的划分。在叶子节点不完全属于一个特定类别的情况下，该叶子节点中的实例会通过误分类率来权衡。

应用实例

这一节我们用监督学习方法实现异常探测，在前面我们给出了使用集成学习对 shuttle 数据集进行异常探测的实例。我们使用随机森林算法进行实验，代码如下：

```
# -*- coding: utf-8 -*-
import pandas as pd
import numpy as np
def print_metric(o,y):
    TP = 0#真正
    FP = 0#假正
    TN = 0#真负
    FN = 0#假负
    for i in range(len(o)):
        if o[i] == 1 and y[i] == 1:
            TN += 1
        if o[i] == 1 and y[i] == -1:
            FN += 1
        if o[i] == -1 and y[i] == -1:
            TP += 1
        if o[i] == -1 and y[i] == 1:
            FP += 1
    print "Recall:",TP*1.0/(TP+FN)
    print "Precision:",TP*1.0/(TP+FP)
names = [1,2,3,4,5,6,7,8,9,10]
df = pd.read_table('shuttle.txt',sep=' ',names = names)
df.sort_index(by=1)
shuttle_class = df[10]
y = shuttle_class
for i in range(len(shuttle_class)):
    if shuttle_class[i] != 1:
        y[i] = -1
    else:
        y[i] = 1
y = y.values# -1(异常)和1（正常）构成的点
#构造训练数据
dfnormal = df[df[10] == 1]#获取正常数据
dfanomaly = df[df[10]!=1]#获取异常数据
dfnormal = dfnormal.drop([1,10],axis =1)
dfanomaly = dfanomaly.drop([1,10],axis = 1)
df = df.drop([1,10],axis =1)#全部数据
train = pd.concat([dfnormal[:900],dfanomaly[:100]])
label = np.ones(1000)
```

```
label[900:] = -1
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(max_depth=3,
random_state=2018,class_weight = 'balanced_subsample')
rf.fit(train,label)
o = rf.predict(df)
print_metric(o,y)
```

运行结果为：

```
Recall: 0.894109861019
Precision: 0.951073565646
```

这里用于训练的数据中正常样本有 900 个，异常样本有 100 个，使用随机森林算法的时候，我们可以指定权重，也可以简单的使用 `class_weight = 'balanced_subsample'` 进行样本平衡处理。通过与之前实验结果的对比可以发现，使用监督学习得到的结果远好于非监督学习，因而工业界有许多人认为在实际异常探测应用时，监督学习的作用是无监督学习不能够替代的。

异常检测应用实例——时空异常检测

前面我们介绍了大量的基础知识，本节我们来关注一个具体的异常探测领域——时空异常探测问题。

空间数据是一种上下文数据类型，可以将空间数据的属性区分为两种数据类型：

行为属性：这是对感兴趣的目标进行度量的属性。例如，这个属性可能对应于海面温度，风速，车速，疾病爆发数量，图像像素的颜色等等。在给定的应用程序中可能有多个行为属性。在许多应用中，这个属性是非空间的，因为它测量了一些在给定空间位置的数据。但是，在某些数据类型（如轨迹）中，行为属性可能是空间的。

上下文属性：在许多空间数据类型中，上下文属性是空间的，尽管它在某些偶然的情况下可能不是空间的。海面温度，风速和汽车速度通常是在特定空间位置的情况下进行测量的。空间上下文通常以坐标表示，其通常对应于两个或三个坐标数值。

空间数据与时间序列数据在上下文方面有许多相似之处。 事实上，空间和时间属性往往可能以行为和上下文属性的组合形式出现。这些数据也被称为时空数据。例如，在一些应用中，例如飓风跟踪，上下文属性既是空间的也是时间的。

空间应用的一些例子如下：

•气象数据在不同的地理位置测量众多天气参数，可用于预测异常天气模式。

•交通数据：移动物体可能与许多参数相关联，如速度，方向等。在很多情况下，这些数据也是时空的，因为它有一个时间分量。发现移动物体的异常行为有很多应用。例如，异常滑行轨迹的发现可以用来发现贪婪和不诚实的出租车司机。

•地球科学数据：不同空间位置的土地覆盖类型可能是行为属性。这种模式的异常提供了有关人类活动异常趋势的见解，如植被减少或其他异常植被趋势。

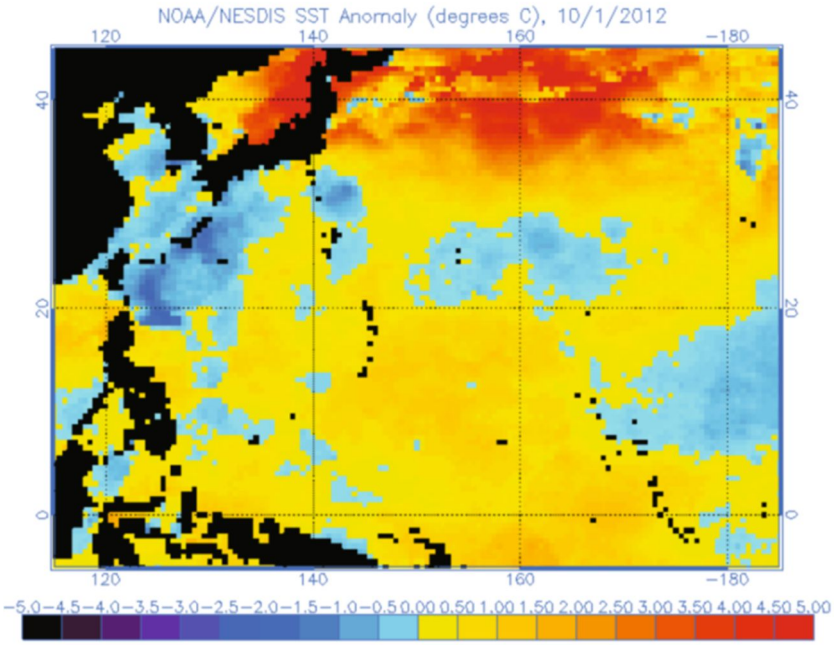


图 14 海平面温度异常

就空间数据而言，行为属性的突变会违反空间连续性，从而被用来识别上下文异常。例如，考虑一个气象应用程序，实时测量海面温度和压力。在一个非常小的局部地区的海面温度出现高温，这可能是由于这个地区地表下火山活动的结果。在这种情况下，空间连续性被违反，这是我们感兴趣的。

在时空数据中，空间和时间连续性都可以用于建模。例如，一个小型局部区域内的几辆汽车的速度突然变化可能表明发生了事故或其他异常事件。同样，不断演变的事件如飓风和疾病的爆发本质上也是时空的。

空间数据有两个主要特征通常在异常点探测问题中被利用：

•空间自相关：这对应于行为属性的事实：空间邻域的数值彼此密切相关。然而，不像时间数据，时间序列的未来值是未知的，而空间数据在各个方向上的值都可以直接使用。请注意空间自相关与时间序列中的时间自相关完全类似。

•空间特异性：行为属性取决于特点的空间位置。

下面我们介绍非常具有代表性的时空异常探测方法：基于邻域的探测算法。

基于邻域的算法在许多任务中可能非常有用。在这些算法中，数据点在空间邻域中的突然变化被用于检测异常值。这些算法取决于空间邻域的具体定义，将这些邻域值组合成一个函数来计算期望值，并计算期望值与期望值的偏差来衡量异常程度。

多维邻域：这种情况下，邻域被定义为数据点之间的距离。

基于图的邻域：在这种情况下，邻域由空间对象之间的连接关系来定义。空间连接关系可能由领域专家来定义。在空间对象的位置可能不对应于确切坐标（例如，县或邮政编码）的情况下，基于图的邻域可能更有用，并且图表示提供了更一般的建模工具。

多维邻域：

虽然传统的多维离群点检测方法（例如 LOF）也可以用来检测空间数据中的异常值，但这些方法不区分上下文属性和行为属性。因此，这些方法并未针对空间数据中的异常值检测问题进行优化，特别是异常值在特定语境被定义的情况下。许多方法利用上下文属性来确定 k 个最近邻居，以及行为属性的偏差值，从而用

于预测异常值。对于具有行为属性值f(o)的给定空间对象 o，设$o_1, o_2, \cdots, o_k$是它的 k 个最近的邻居。然后，对象的行为属性的预测值g(o)可以使用邻域的平均值来计算：

$$g(o) = \sum_{i=1}^{k} f(o_i)/k$$

我们也可以使用邻域中心点来减少极端值的影响。对于数据对象 o，f(o) − g(o)代表了预期值和实际值的偏差。

一个值得注意的问题叫做“局部异常”：

据观察，在异常分析中，局部数据方差的重要性是不同的。例如，考虑一个海平面温度监控的实例。在一些空间区域中，温度的改变比其他区域更明显。通常情况下，高方差区域的异常分数需要被缩减。例如，我们不应该使用f(o) − g(o)，而是使用一个标准化的值$\frac{f(o)-g(o)}{L(o)}$，其中$L(o)$表示 o 附近的空间局部的值。比如$L(o)$可以是 o 的空间邻居的标准差。

为了刻画与对象 o 的空间偏差，提出了许多方法，比如 SLOM 算法就基于 LOF 算法实现了空间异常探测。

基于图的邻域

在基于图的方法中，空间邻近度使用节点之间的链接来建模。因此，节点与行为属性相关联，并且相邻节点之间的行为属性的强烈变化被识别为异常值。在单个节点不与特定的坐标相关联，但可能对应于任意形状区域的情况下，基于图的方法可能特别有用。在这种情况下，节点之间的链接可以基于不同的邻域关系建模。

基于图的方法以自然的方式定义空间关系，因为语义关系也可以用来定义邻域。通常情况下，空间领域专家可能会构建邻域图。如果两个对象的位置语义相同，则可以通过边相互链接，比如建筑物，餐馆或办公室。在许多应用程序中，链接可能根据邻近关系的强度进行加权。例如，考虑一个疾病暴发实例，空间对象对应于县区。在这种情况下，链接的权重可以对应相邻县区之间的距离。

令 S 为给定节点 o 的邻居集合。那么利用空间连续性的性质，基于 o 的邻居节点就能得到一个预测值。o 和邻居之间的权重可以被用来计算加权平均数。对于给定空间节点 o,其属性值为f(o)，令$o_1, o_2, \cdots, o_k$为其 k 个基于关系图的邻居。令链接（o, o_i）的权重为 w(o, o_i)。那么，基于链接的加权平均数可以用于计算对象 o 的平均值：

$$g(o) = \frac{\sum_{i=1}^{k} w(o, o_i) \cdot f(o_i)}{\sum_{i=1}^{k} w(o, o_i)}$$

同样的，f(o) − g(o)代表了预期值和实际值的偏差。我们使用这个偏差来寻找异常值。

小结

与一般的异常探测算法相比，时空异常检测算法应当结合时空数据特有的数据特点进行优化，比如同一个区域的数据是相似的。

本节我们重点介绍了基于邻域的异常探测算法，并分别讲述了基于距离和基

于图的探测算法的思想，这两种方法都用估计值和实际值之间的偏差来进行异常探测。

这种思想在异常探测领域也十分常见：**如果对一个数据的估计（预期）值与实际值偏差较大，则认为该数据是异常的**。

异常检测大数据应用实例

我们应用 pyspark 解决这样一个实际问题：

某个工业设备制造公司生成一种零件，其中有一项重要的参数能够反应零件的质量。由于数据量很大，并且也没有规定的参数“标准值”，我们要使用三西格玛原则进行异常探测，这样有助于找到异常数据，提高产品质量。

解决问题的步骤分为两步：

（1） 求解原始数据集的平均值μ和标准差δ

（2） 返回数值超过$[\mu - 3 * \delta,\ \mu + 3 * \delta]$的加工数据，这些数据被判定为异常。

步骤一对应的代码如下：

```
# -*- coding: utf-8 -*-
import pyspark
from pyspark import SparkContext as sc
from pyspark import SparkConf
import math
conf = SparkConf().setAppName("calculate avg and
std").setMaster("local[*]")
sc = SparkContext.getOrCreate(conf)
'''
从本地读取文件
'''
fileRDD=sc.textFile("file:///home/dawnson/anomaly/metrics") #文件路径
'''
将每行的数值转化为 float 类型
'''
numberRDD = fileRDD.map(lambda x: float(x))
'''
计算平均值和方差
'''
```

```
avg = numberRDD.sum()*1.0/numberRDD.count()
difRDD = numberRDD.map(lambda x: (x-avg)*(x-avg))
std = difRDD.sum()*1.0/numberRDD.count()
delta = math.sqrt(std)
```

这里需要注意，我们的数据从本地文件中读取（我的测试文件存储在 linux 系统/home/dawnson/anomaly/metrics 路径上），你也可以从 HDFS 上读取相关数据。通过上述代码，我们可以统计出数据的平均值和方差，并将其作为步骤二的输入。

步骤二对应的代码为：

```
# -*- coding: utf-8 -*-
import pyspark
from pyspark import SparkContext as sc
from pyspark import SparkConf
conf = SparkConf().setAppName("Anomaly
detection").setMaster("local[*]")
sc = SparkContext.getOrCreate(conf)
def ThreeSigmaDetecion(numberRDD,avg,std):
    return numberRDD.filter(lambda x: x> avg+3*std or x< avg-3*std)
'''
从本地读取文件
'''
fileRDD=sc.textFile("file:///home/dawnson/anomaly/metrics")
'''
将每行的数值转化为 float 类型
'''
numberRDD = fileRDD.map(lambda x: float(x))
'''
查找满足条件的 item,我们使用 3 西格玛原则探测异常值
'''
anomalyRDD = ThreeSigmaDetecion(numberRDD, μ, δ)
print(anomalyRDD.collect())
```

注意在步骤二倒数第二行黑体**μ**, **δ**在实际运行时务必替换成第一步计算出来的数值。

我们可以用简单的数据测试程序的正确性：测试文件的前五条数据如下：

```
10.81
10.82
10.83
10.81
10.823
10.81
```

测试文件的末尾我们添加两个异常值 18.3 和 5.7。通过实际测试，程序正确返回异常值，这样我们就可以将程序应用到实际的 G 级数据上了。

本章小结

通过本章的介绍，我们可以发现，不同的异常探测算法对于异常有着不同的假设和出发点。比如隔离森林认为：异常是“少而不同”的，因此更容易被隔离，从而与正常数据区分开来；LOF 算法认为异常是与邻居密度不同的数据；One-class svm 认为异常数据是不符合正常数据轮廓的数据。

在实际的异常探测问题当中，针对具体的问题，灵活的使用异常探测算法是必要的。比如有许多网络入侵相关研究喜欢应用 PCA 算法和 One-class svm 算法，能够有效识别新异常，以不变应万变。

本章我们还介绍了使用集成学习算法构造更强大异常探测器的方法。同时我们也展示了应用集成学习技巧解决高维数据异常探测以及参数设置自动化的算法应用。最后我们还介绍了时空数据异常检测的方法。

参考文献

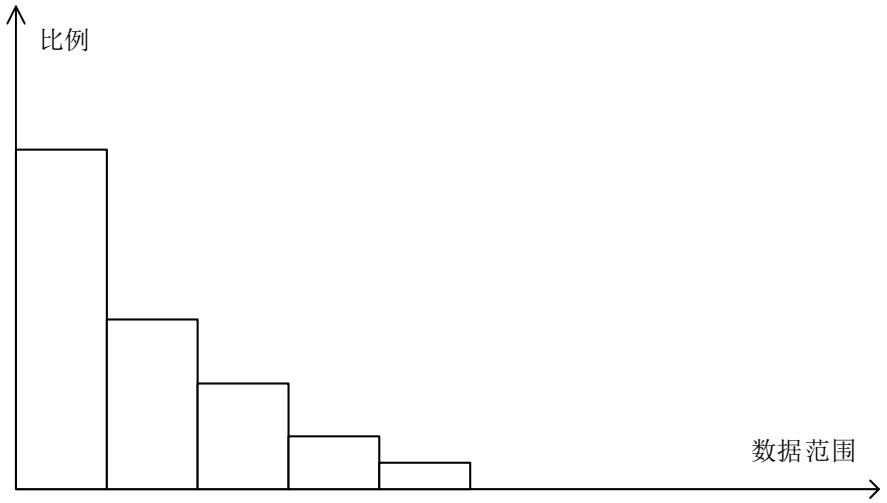
1. Liu F T, Kai M T, Zhou Z H. Isolation-Based Anomaly Detection[J]. Acm Transactions on Knowledge Discovery from Data, 2012, 6(1):1-39.
2. Breunig, Kriegel, Ng, and Sander (2000) LOF: identifying density-based local outliers. Proc. ACM SIGMOD
3. Perdisci R, Gu G, Lee W. Using an Ensemble of One-Class SVM Classifiers to Harden Payload-based Anomaly Detection Systems[C]// International Conference on Data Mining. IEEE, 2007:488-498.
4. Agarwal C. Outlier ensembles[C]// ACM SIGKDD Workshop on Outlier Detection and Description. 2013:6-6.

习题

1. 本章我们首先介绍了应用统计知识探测异常的基本方法。我们要解决这样一个实际应用问题：网络入侵问题中，我们需要提取各种特征：例如 URL 参数个数、参数值长度的均值和方差等来探测非法 URL。现在给出切比雪夫不等式如下：

$$P(|X-\mu| \geq k\sigma) \leq \frac{1}{k^2}$$

   请应用切比雪夫不等式设计一种异常探测方法，有效探测 URL 参数值长度异常。
2. 本章介绍了异常探测问题和分类问题的区别，请举出 5 个例子，说明每个例子适用于哪类问题。
3. 请查看 sklearn 中隔离森林实现的源码，并说明在 sklearn 如何计算隔离森林的异常得分。
4. 本章我们介绍了使用聚类方法进行异常探测的思想：离群点往往属于小的或者稀疏的簇，或者不属于任何簇；而正常数据往往属于大的，稠密的簇。请应用 kmeans 算法实现一种异常探测算法。
5. 基于统计的异常探测问题中，有一种非参数方法称为直方图异常检测法：



   如图所示，通过按照数据范围进行划分，将数据分配到相应的直方图中，我们可以统计出每个直方图中数据所占的比例。

   请设计一种算法，利用直方图进行异常探测，给出构造直方图和检测异常点的详细步骤，并给出异常得分的计算公式。
6. 在互联网应用中，有一种技术叫做用户行为检测。通过对用户行为进行分析，我们能找到用户异常行为事件，这种异常是我们十分关心的。比如，在网上交易平台中，假如我们探测到了某个在线用户行为存在异常，那么很可能发生了用户账号被盗等非法事件；在电商平台中，对于每个客户，我们都记录了客户的浏览记录，包括用户浏览和搜索的商品信息，假如一个客户突然购买了一件与他之前浏览很不相关的物品，我们就会非常在意这次异常事件。问题是，哪些算法适用于用户行为检测，请分析原因，并查阅相关资料。
7. 许多异常探测算法都定义了一系列的阈值，超过阈值就认为是异常。例如在

多元离群点的检测方法中，我们通过 p(x)来判断新样本是否异常：

$$\begin{cases} if\ \mathrm{p(x)} > \mathrm{C}, \text{输出异常} \\ \quad \text{否则输出正常} \end{cases}$$

其中 C 为阈值，请思考如何通过训练数据确定 C 的值。（提示：使用交叉验证等方法）

8. 我们简单介绍了基于距离和近邻进行异常探测的思想，如果一个点与其最近邻居的距离都超出了阈值，则认为改点是异常。请设计算法，应用 KNN 和阈值规则进行异常探测，并给出阈值设定的方法（请充分思考 KNN 算法中 K 值的设定，阈值如何选取）。
9. 本章我们介绍了使用集成学习技术实现异常检测的方法。请使用独立集成技术，将隔离森林算法和 LOF 算法进行结合得到一种新算法。并重点关注如何处理隔离森林的异常得分和 LOF 算法的异常得分。
10. （编程题）本章我们使用 sklearn 软件包向大家展示了隔离森林、one-class svm 和 LOF 算法的简单使用，下面请通过实际编程来解决一个问题：
    许多异常探测的论文（比如我们介绍过的隔离森林）都使用到了 shuttle 数据集：http://archive.ics.uci.edu/ml/datasets/Statlog+%28Shuttle%29
    我们认为数量很少的某类数据是异常数据。因此在 shuttle 数据集中类别为 2,3,5,6,7（约占 7%）的数据是异常数据。
    （1） 请分别使用隔离森林、one-class svm 和 LOF 算法进行异常探测，并计算精度 precision，召回率 recall 和 F 度量；比较各个算法的表现，并分析原因。
    （2） 使用集成学习的方法将 2 种或 3 种异常算法进行集成，并计算精度 precision，召回率 recall 和 F 度量。