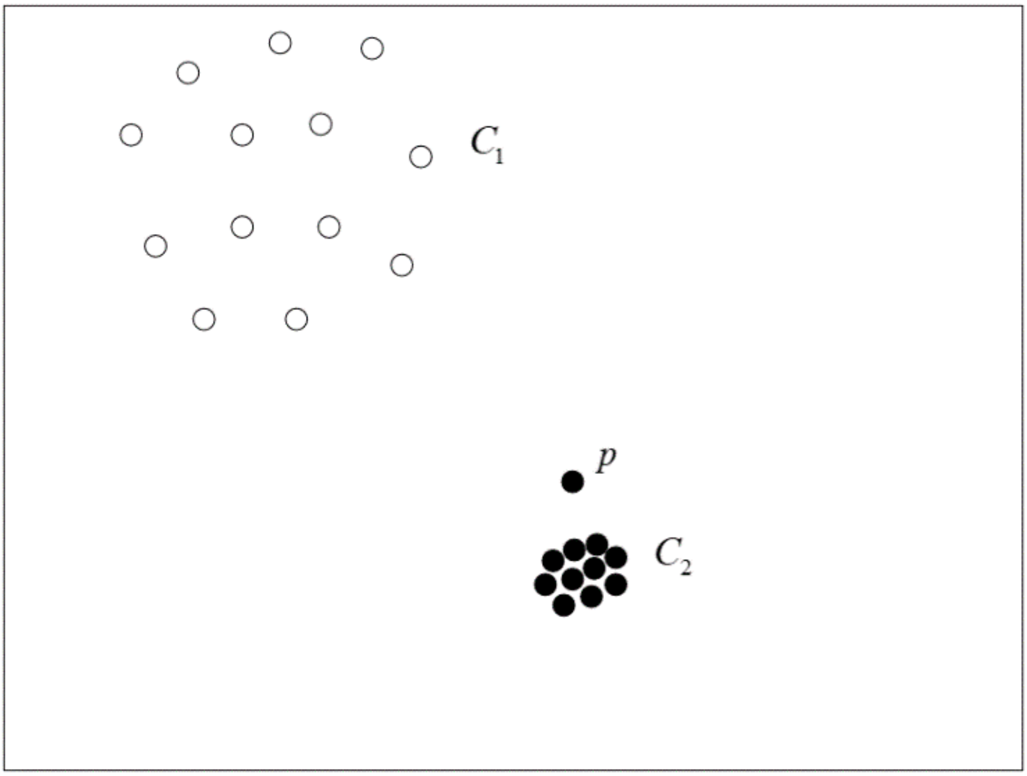


离群点检测——局部离群因子(Local Outlier Factor，LOF)算法

禺垣

1 概述

离群点是观察的数据集中明显异常的数据点，或者说，离群点的数据分布与数据集的整体分布不同。离群点检测的目的是检测出那些与正常数据差别较大的数据点，然后根据具体的问题作进一步处理。

离群点检测算法主要有基于统计、聚类、分类、信息论、距离、密度等相关的方法，列表如下

检测方法	方法描述	优缺点
基于统计	根据数据的分布特点，选择一个概率分布模型对数据进行匹配，将不能匹配的数据点识别为离群点。	优点： 统计方法广泛。缺点：在高维数据上的应用效果不够理想；实际数据分布规律无法预估，难以用单一的分布模型来刻画。
基于聚类	应用聚类算法对数据进行聚类操作，将不归属于任何一个类簇的点识别为离群点。	优点： 聚类算法理论完善。缺点：主要做聚类，附带检测离群点，检测效果不够理想；时间复杂度较高。
基于分类	应用分类算法，对数据点做是否离群的类别判定。	优点： 分类算法理论完善。缺点：对训练集的数据质量要求较高。
		优点： 仅依赖于数据对象的本身属性特性；数据属性类型适应性

基于信息论	将信息论的理论应用到离群点检测中。	强，既可以是数值型，也可以是标称属性。缺点：计算和度量复杂数据的信息熵或Kolomogorov复杂度较为困难。
基于距离	对某一个数据点，超过一定部分的数据与它的距离都大于一定值，那么将它识别为离群点。	优点： 方法简单，易于操作。缺点：对参数敏感；时间复杂度偏高；在高维稀疏数据集上效果不理想。
基于密度	根据数据的密集情况，计算每个数据对象的局部离群因子，用以标识数据的离群程度。选出top(n)个离群程度最大的点作为离群点。	优点： 方法简洁，不受数据分布影响。缺点：对近邻参数较为敏感；时间复杂度较高；在高维大数据集上效率较低。

> 【注】
> 1)离群点不同于噪声，非噪声点也可能离群，噪声应该在离群点检测前完成去除。
> 2)离群点检测算法的评价指标同二分类，可使用正确率(Accuracy)、查准率(Precision)、查全率(Recall)、F值(F1-scores)等指标进行评估。

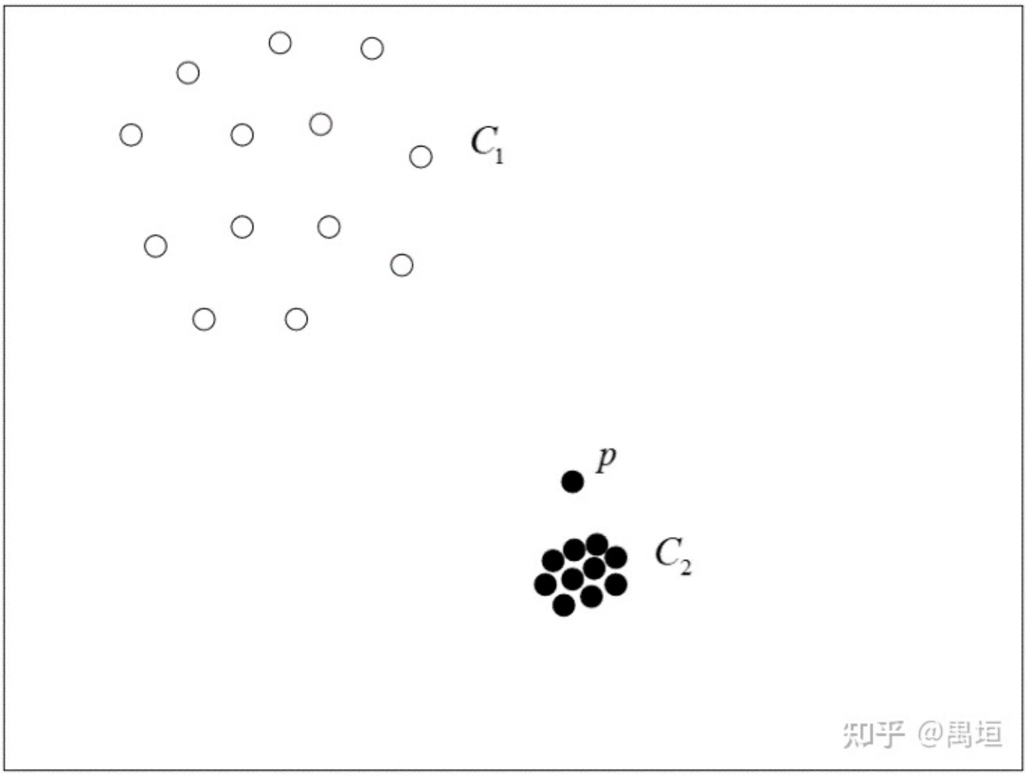
本文介绍一种基于密度的离群点检测方法——局部离群因子算法。

2 局部离群因子(Local Outlier Factor，LOF)算法

2.1 算法思想

局部离群因子(LOF，又叫局部异常因子)算法是Breunig于2000年提出的一种基于密度的局部离群点检测算法，该方法适用于不同类簇密度分散情况迥异的数据。

如下图中，集合C1是低密度区域，集合C2是高密度区域，依据传统的基于密度的离群点检测算法，点p与C2中邻近点的距离小于C1中任何一个数据点与其邻近点的距离，点p会被看作是正常的点，而在局部来看，点p却是事实上的孤立点，LOF算法即可以有效地实现对该种情形的离群点检测。



LOF算法的基本思想是，根据数据点周围的数据密集情况，首先计算每个数据点的一个局部可达密度，然后通过局部可达密度进一步计算得到每个数据点的一个离群因子，该离群因子即标识了一个数据点的离群程度，因子值越大，表示离群程度越高，因子值越小，表示离群程度越低。最后，输出离群程度最大的top(n)个点。

2.2 概念定义

（1）点到点的距离：

$d(p, o)$，数据点p到数据点o的距离。

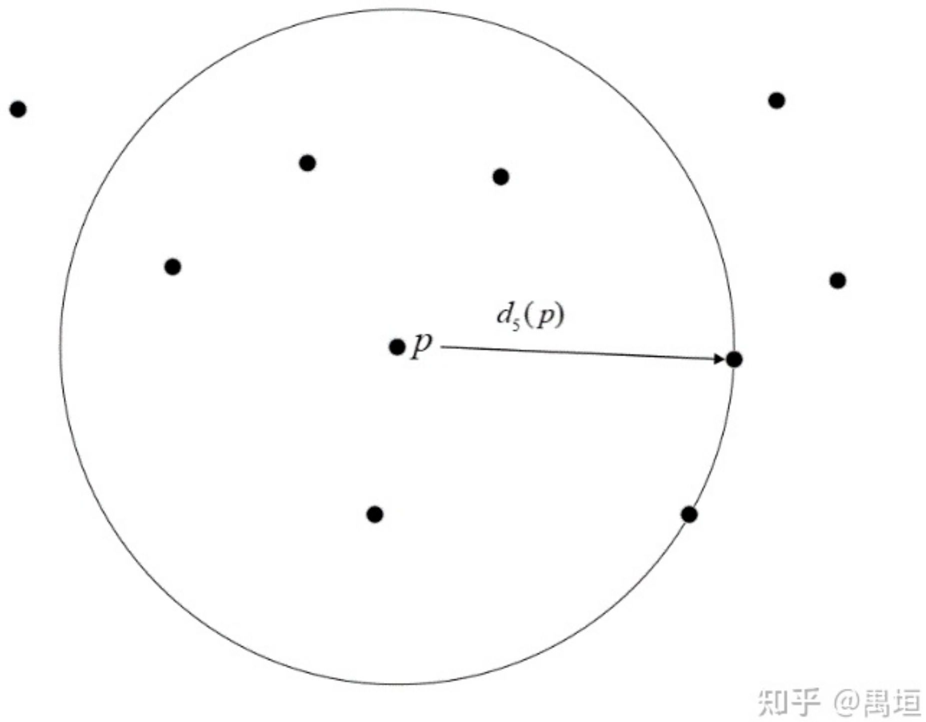
（2）第k距离：

数据点p的第k距离 $d_k(p)$，定义为： $d_k(p) = d(p, o)$，满足

a)在集合中至少有不包括p在内的k个点o′，使得 $d(p, o') \leq d(p, o)$；

b)在集合中至多有不包括p在内的k-1个点o′，使得 $d(p, o') < d(p, o)$.

通俗地讲，就是以p为圆心向外辐射，直至涵盖了第k个邻近点。下图中示意了p的第5距离



(3)第k距离邻域：

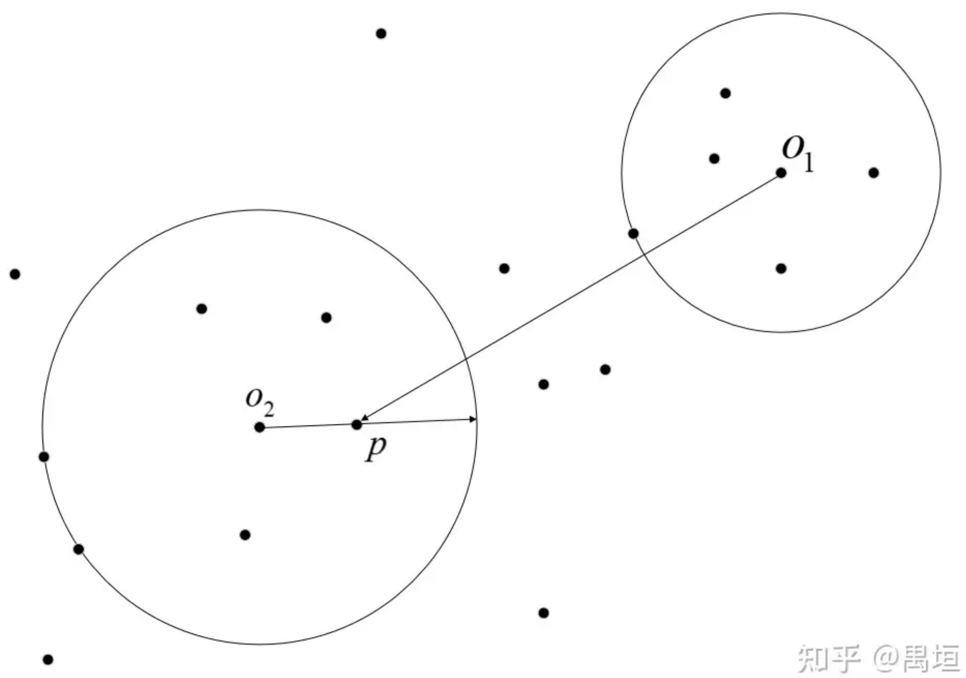
数据点p的第k距离邻域$N_k(p)$，指点p的第k距离内的所有点的集合，包括第k距离上的点.

易知，有 $|N_k(p)| \geq k$.

(4)第k可达距离：

$$reach\_dist_k(o,p) = max\left\{d_k(o), d(o,p)\right\}$$

数据点o到数据点p的第k可达距离，定义为点o的第k距离和点o到点p的距离中的较大者。如下图中，o1到p的第5可达距离为 $d(o_1,p)$ ，o2到p的第5可达距离为 $d_5(o_2)$



易知，点o到点o的第k邻域内所有点的第k可达距离均为 $d_k(o)$.

(5)局部可达密度(local reachability density)：

$$lrd_k(p) = 1/\left(\frac{\sum_{o\in N_k(p)} reach\_dist_k(o,p)}{|N_k(p)|}\right)$$

数据点p的第k局部可达密度，即点p的第k距离邻域内的所有点到点p的平均第k可达距离的倒数。它表征了点p的密度情况，点p与周围点密集度越高，各点的可达距离越可能是较小的各自的第k距离，lrd值越大；点p与周围点的密集度越低，各点的可达距离越可能是较大的两点间的实际距离，lrd值越小。

(6)局部离群因子：

$$LOF_k(p) = \frac{\sum_{o\in N_k(p)} \frac{lrd_k(o)}{lrd_k(p)}}{|N_k(p)|} = \frac{\sum_{o\in N_k(p)} lrd_k(o)}{|N_k(p)|}/lrd_k(p)$$

数据点p的第k局部离群因子，意为将点p的 $N_k(p)$ 邻域内所有点的平均局部可达密度与点p的局部可达密度作比较，这个比值越大于1，表明p点的密度越小于其周围点的密度，p点越可能是离

2.3 算法描述

输入：数据点集合D;

输出：离群点集合O.

计算每个点的局部可达密度，进而计算得到每个点的局部离群因子，选取输出离群程度最高的n个点：

(1)计算每个点的第k距离邻域内各点的第k可达距离：

$$reach\_dist_k(o,p) = max\left\{d_k(o), d(o,p)\right\}$$

其中， $d_k(o)$ 为领域点o的第k距离， $d(o,p)$ 为邻域点o到点p的距离.

(2)计算每个点的局部第k局部可达密度：

$$lrd_k(p) = 1/\left(\frac{\sum_{o\in N_k(p)} reach\_dist_k(o,p)}{|N_k(p)|}\right)$$

其中， $N_k(p)$ 为p点的第k距离邻域.

(3)计算每个点的第k局部离群因子：

$$LOF_k(p) = \frac{\sum_{o\in N_k(p)} \frac{lrd_k(o)}{lrd_k(p)}}{|N_k(p)|} = \frac{\sum_{o\in N_k(p)} lrd_k(o)}{|N_k(p)|}/lrd_k(p)$$

其中， $N_k(p)$ 为p点的第k距离邻域.

(4)对最大的n个局部离群因子所属的数据点，输出离群点集合：

$O = \{o_1, o_2, \ldots, o_n\}$.

3 python实现

算法实现，lof.py文件

```
#!/usr/bin/python
```

```python
# -*- coding: utf8 -*-
from __future__ import division


def distance_euclidean(instance1, instance2):
    """Computes the distance between two instances. Instances should be tuples of
    Returns: Euclidean distance
    Signature: ((attr_1_1, attr_1_2, ...), (attr_2_1, attr_2_2, ...)) -> float"""

    def detect_value_type(attribute):
        """Detects the value type (number or non-number).
        Returns: (value type, value casted as detected type)
        Signature: value -> (str or float type, str or float value)"""
        from numbers import Number
        attribute_type = None
        if isinstance(attribute, Number):
            attribute_type = float
            attribute = float(attribute)
        else:
            attribute_type = str
            attribute = str(attribute)
        return attribute_type, attribute

    # check if instances are of same length
    if len(instance1) != len(instance2):
        raise AttributeError("Instances have different number of arguments.")
    # init differences vector
    differences = [0] * len(instance1)
    # compute difference for each attribute and store it to differences vector
    for i, (attr1, attr2) in enumerate(zip(instance1, instance2)):
        type1, attr1 = detect_value_type(attr1)
        type2, attr2 = detect_value_type(attr2)
        # raise error is attributes are not of same data type.
        if type1 != type2:
            raise AttributeError("Instances have different data types.")
        if type1 is float:
            # compute difference for float
            differences[i] = attr1 - attr2
        else:
            # compute difference for string
            if attr1 == attr2:
                differences[i] = 0
            else:
                differences[i] = 1
    # compute RMSE (root mean squared error)
    rmse = (sum(map(lambda x: x ** 2, differences)) / len(differences)) ** 0.5
    return rmse


class LOF:
    """Helper class for performing LOF computations and instances normalization."

    def __init__(self, instances, normalize=True, distance_function=distance_eucl
        self.instances = instances
        self.normalize = normalize
        self.distance_function = distance_function
        if normalize:
            self.normalize_instances()

    def compute_instance_attribute_bounds(self):
        min_values = [float("inf")] * len(self.instances[0])  # n.ones(len(self.i
        max_values = [float("-inf")] * len(self.instances[0])  # n.ones(len(self.
        for instance in self.instances:
```

```
            min_values = tuple(map(lambda x, y: min(x, y), min_values, instance))
            max_values = tuple(map(lambda x, y: max(x, y), max_values, instance))
        self.max_attribute_values = max_values
        self.min_attribute_values = min_values

    def normalize_instances(self):
        """Normalizes the instances and stores the infromation for rescaling new
        if not hasattr(self, "max_attribute_values"):
            self.compute_instance_attribute_bounds()
        new_instances = []
        for instance in self.instances:
            new_instances.append(
                self.normalize_instance(instance))  # (instance - min_values) / (
        self.instances = new_instances

    def normalize_instance(self, instance):
        return tuple(map(lambda value, max, min: (value - min) / (max - min) if m
                         instance, self.max_attribute_values, self.min_attribute_

    def local_outlier_factor(self, min_pts, instance):
        """The (local) outlier factor of instance captures the degree to which we
        min_pts is a parameter that is specifying a minimum number of instances t
        Returns: local outlier factor
        Signature: (int, (attr1, attr2, ...), ((attr_1_1, ...),(attr_2_1, ...), .
        if self.normalize:
            instance = self.normalize_instance(instance)
        return local_outlier_factor(min_pts, instance, self.instances, distance_f


def k_distance(k, instance, instances, distance_function=distance_euclidean):
    # TODO: implement caching
    """Computes the k-distance of instance as defined in paper. It also gatheres
    Returns: (k-distance, k-distance neighbours)
    Signature: (int, (attr1, attr2, ...), ((attr_1_1, ...),(attr_2_1, ...), ...))
    distances = {}
    for instance2 in instances:
        distance_value = distance_function(instance, instance2)
        if distance_value in distances:
            distances[distance_value].append(instance2)
        else:
            distances[distance_value] = [instance2]
    distances = sorted(distances.items())
    neighbours = []
    k_sero = 0
    k_dist = None
    for dist in distances:
        k_sero += len(dist[1])
        neighbours.extend(dist[1])
        k_dist = dist[0]
        if k_sero >= k:
            break
    return k_dist, neighbours


def reachability_distance(k, instance1, instance2, instances, distance_function=c
    """The reachability distance of instance1 with respect to instance2.
    Returns: reachability distance
    Signature: (int, (attr_1_1, ...),(attr_2_1, ...)) -> float"""
    (k_distance_value, neighbours) = k_distance(k, instance2, instances, distance
    return max([k_distance_value, distance_function(instance1, instance2)])


def local_reachability_density(min_pts, instance, instances, **kwargs):
```

```python
    """Local reachability density of instance is the inverse of the average reach
    distance based on the min_pts-nearest neighbors of instance.
    Returns: local reachability density
    Signature: (int, (attr1, attr2, ...), ((attr_1_1, ...),(attr_2_1, ...), ...))
    (k_distance_value, neighbours) = k_distance(min_pts, instance, instances, **k
    reachability_distances_array = [0] * len(neighbours)  # n.zeros(len(neighbour
    for i, neighbour in enumerate(neighbours):
        reachability_distances_array[i] = reachability_distance(min_pts, instance
    sum_reach_dist = sum(reachability_distances_array)
    if sum_reach_dist == 0:
        return float('inf')
    return len(neighbours) / sum_reach_dist


def local_outlier_factor(min_pts, instance, instances, **kwargs):
    """The (local) outlier factor of instance captures the degree to which we cal
    min_pts is a parameter that is specifying a minimum number of instances to cc
    Returns: local outlier factor
    Signature: (int, (attr1, attr2, ...), ((attr_1_1, ...),(attr_2_1, ...), ...))
    (k_distance_value, neighbours) = k_distance(min_pts, instance, instances, **k
    instance_lrd = local_reachability_density(min_pts, instance, instances, **kwa
    lrd_ratios_array = [0] * len(neighbours)
    for i, neighbour in enumerate(neighbours):
        instances_without_instance = set(instances)
        instances_without_instance.discard(neighbour)
        neighbour_lrd = local_reachability_density(min_pts, neighbour, instances_
        lrd_ratios_array[i] = neighbour_lrd / instance_lrd
    return sum(lrd_ratios_array) / len(neighbours)


def outliers(k, instances, **kwargs):
    """Simple procedure to identify outliers in the dataset."""
    instances_value_backup = instances
    outliers = []
    for i, instance in enumerate(instances_value_backup):
        instances = list(instances_value_backup)
        instances.remove(instance)
        l = LOF(instances, **kwargs)
        value = l.local_outlier_factor(k, instance)
        if value > 1:
            outliers.append({"lof": value, "instance": instance, "index": i})
    outliers.sort(key=lambda o: o["lof"], reverse=True)
    return outliers
```

测试程序，test_lof.py文件

```python
# -*- coding: utf8 -*-
instances = [
 (-4.8447532242074978, -5.6869538132901658),
 (1.7265577109364076, -2.5446963280374302),
 (-1.9885982441038819, 1.705719643962865),
 (-1.999050026772494, -4.0367551415711844),
 (-2.0550860126898964, -3.6247409893236426),
 (-1.4456945632547327, -3.7669258809535102),
 (-4.6676062022635554, 1.4925324371089148),
 (-3.6526420667796877, -3.5582661345085662),
 (6.4551493172954029, -0.45434966683144573),
 (-0.56730591589443669, -5.5859532963153349),
 (-5.1400897823762239, -1.3359248994019064),
 (5.2586932439960243, 0.032431285797532586),
 (6.3610915734502838, -0.99059648246991894),
```

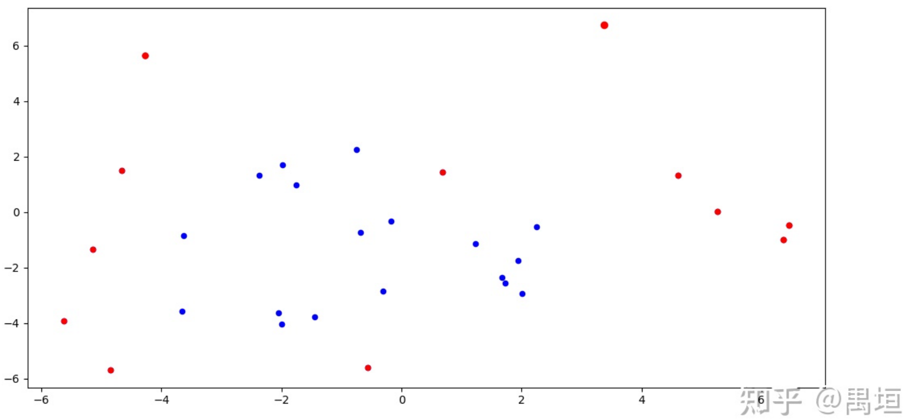
```
 (-0.31086913190231447, -2.8352818694180644),
 (1.2288582719783967, -1.1362795178325829),
 (-0.17986204466346614, -0.32813130288006365),
 (2.2532002509929216, -0.5142311840491649),
 (-0.75397166138399296, 2.2465141276038754),
 (1.9382517648161239, -1.7276112460593251),
 (1.6809250808549676, -2.3433636210337503),
 (0.68466572523884783, 1.4374914487477481),
 (2.0032364431791514, -2.9191062023123635),
 (-1.7565895138024741, 0.96995712544043267),
 (3.3809644295064505, 6.7497121359292684),
 (-4.2764152718650896, 5.6551328734397766),
 (-3.6347215445083019, -0.85149861984875741),
 (-5.6249411288060385, -3.9251965527768755),
 (4.6033708001912093, 1.3375110154658127),
 (-0.685421751407983, -0.73115552984211407),
 (-2.3744241805625044, 1.3443896265777866)]

from lof import outliers
lof = outliers(5, instances)

for outlier in lof:
    print (outlier["lof"],outlier["instance"])

from matplotlib import pyplot as p

x,y = zip(*instances)
p.scatter(x,y, 20, color="#0000FF")

for outlier in lof:
    value = outlier["lof"]
    instance = outlier["instance"]
    color = "#FF0000" if value > 1 else "#00FF00"
    p.scatter(instance[0], instance[1], color=color, s=(value-1)**2*10+20)

p.show()
```

运行结果：

输出离群点的lof值及坐标信息

```
2.2048496921690095 (3.3809644295064505, 6.749712135929268)
1.794844084823056 (-4.27641527186509, 5.6551328734397766)
1.5012186584843135 (6.455149317295403, -0.45434966683144573)
1.4794025326219273 (6.361091573450284, -0.9905964824699189)
1.3721695654932344 (5.258693243996024, 0.032431285797532586)
1.2910019510075679 (4.603370800191209, 1.3375110154658127)
1.2027400633270513 (-4.844753224207498, -5.686953813290166)
1.1871801839835139 (-5.6249411288060385, -3.9251965527768755)
1.108985678163174 (0.6846657252388478, 1.4374914487477481)
1.057283040066788 (-4.667606202263555, 1.4925324371089148)
1.0421629593470334 (-5.140089782376224, -1.3359248994019064)
1.0280116793513516 (-0.5673059158944367, -5.585953296315335)
```


可视化，其中红色的点为检测出的离群点



参考

1. 陈瑜. 离群点检测算法研究[D].兰州大学,2018.
2. blog.csdn.net/wangyibo0...
3. blog.csdn.net/ilike_pro...

编辑于 2021-06-30 10:38

异常检测 机器学习 数据挖掘

8 条评论

默认 最新

夯鳕

写的很棒，很容易看明白，就是我代码能力太差了，我把代码复制过去，运行没什么问题，就是不出任何数据😭

08-04

回复 喜欢

屠美狗

写得很好，我谢谢你

08-02

回复 喜欢

春夏秋冬

博主还在吗，我对这个有不明白的地方，就是我自己手算的结果和模型跑出来的结果不一样，我不知道哪个地方出错了，数据点用的是(0, 0),(5, 0),(6, 0),(8, 0)K =2，就只有第一个点计算出来和模型跑出来的结果一样😭😭😭😭😭😭😭

06-20

回复 喜欢

ajdhrn

局部可达密度的概念感觉读起来怪怪的

2022-11-01

回复 喜欢

ajdhrn

怎么说呢，能够理解，但是有些费力。如果有可视化图的话可能会更好的理解

2022-11-01

回复 喜欢

cc酱

写得很好，准备放到模型里试试

2022-04-08

回复 喜欢

禹垣 作者

谢谢，欢迎关注🫣

2022-04-09

回复 喜欢

悠游

非常感谢！

2022-03-30

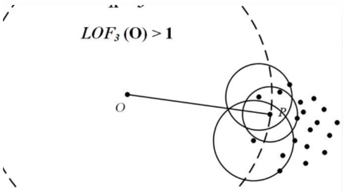
回复 喜欢

文章被以下专栏收录

算法笔记

算法笔记

推荐阅读



.OF离群因子检测算法及
ython3实现

iuranyi

Tesnor Ridge Regression 与多信息源因子张量表征

观鱼

【矿友必读】多因子模型中常见的因子合成方法

优矿量化实验室

因子模型实践：因子合成

苏什么来看