[testing]时间序列相似性对比

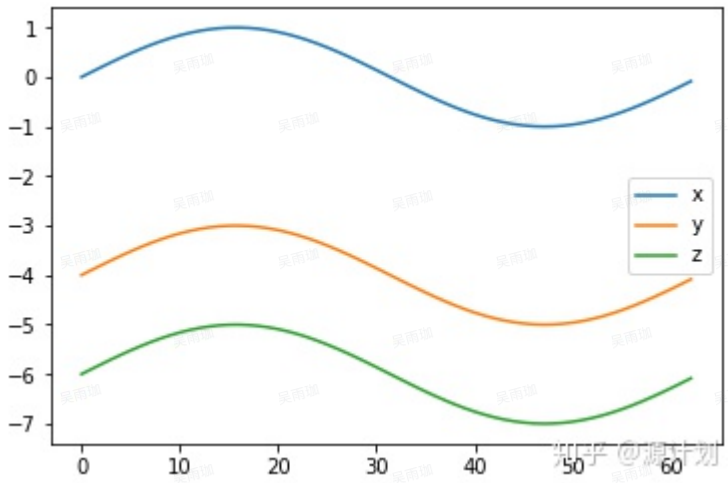
similarity search in time series subsequeces

该算法属于的领域：曲线相似性/曲线匹配curve matching/Data Mining

一、问题定义

1、概念

1. 时间序列 Time series：一组按照时间顺序进行排列的数据点序列。
   a. 离散，一般时间序列的时间间隔是一个恒定值，如1s, 1day, 1week等
   b. 时间序列分析：目的是从中提取有效的信息
2. 相似 Similarity
   a. 两个时间序列「相似」，一般指它们满足以下两方面的条件
      i. 距离最近
      ii. 形状相似



如上图，一般认为x, y, z形状相似，y, z距离最相近，因此y和z最相似。

3. 子序列匹配 subsequence matching：已有一短时间序列B，需要在另一长时间序列A中找出与B最相似的部分。
   a. 方式：B从A的开头开始匹配，逐步滚动，直到匹配到A的尾部，最终得到最匹配的A的子序列。

2、规范描述

1. 时间序列的表示

a. 时间序列 T：有限集，每个元素都是一个二元组 (p_i, t_i)，时刻 t_i 生成数据变量 p_i

b. 其子序列的表示：$T_{i,k}$，表示从 t_i 时刻开始的长度为k的序列

2. 子序列匹配问题中，两个时间序列的表示

a. 查询序列Q：在时间序列T中查询与Q最相似的序列。

i. 长度表示：|Q|=n

b. 候选序列C：与Q最相似/匹配的子序列 $T_{i,k}$ 作为候选序列，记作C

3、欧式距离

衡量距离和形状，首先想到的方法是欧式距离。

1. 计算

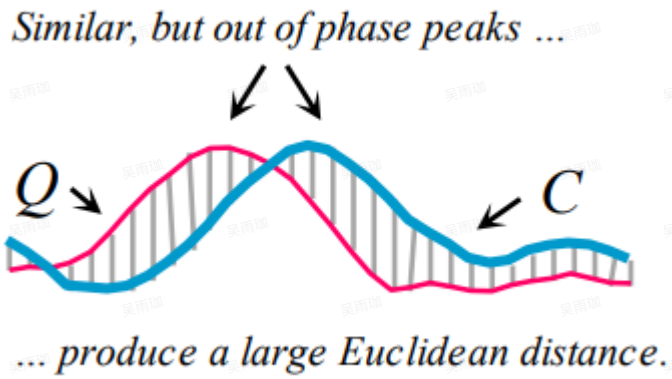
计算每两点之间的距离然后求和，如下

$$ED(Q,C)=\sqrt{\sum_{i=1}^{n}(q_i-c_i)^2}, |Q|=|C|=n$$

2. 缺陷

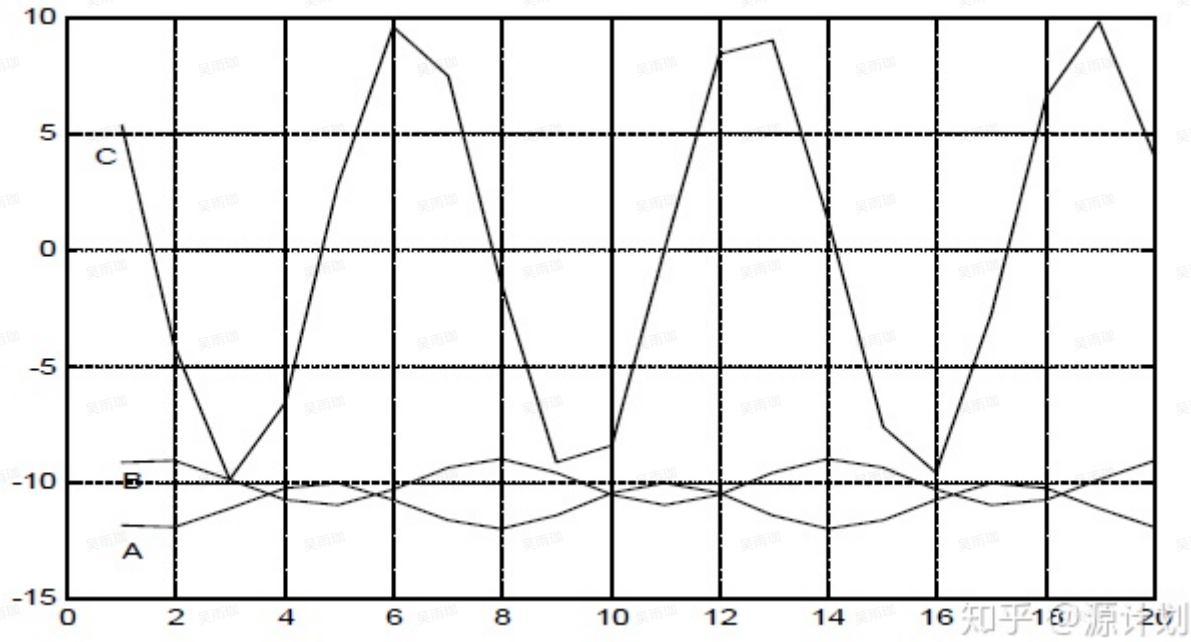
a. 不能辨别形状相似性

i. 平移：如下图，C看起来只是Q的简单平移，但产生了较大的欧氏距离



ii. 压缩/拉伸：比如同一个单词apple，不同的人对这个单词的发音并不相同（有的人an发音长，有的人发音短），但是匹配算法需要识别出它是同一个单词

b. 不能反映变化趋势



如上图，A与B的变化趋势几乎完全相反，与C的变化基本相同。在变化上，A与C相似，而用欧式距离判断会得出A与B相似的结论。

4、解法要求

现实问题：

对指定的某个异常监控指标曲线，通过波动相似性对比分析，从海量监控指标中筛选出所有具有相似波动规律的监控指标曲线（统一提供监控数据）

- 可以使用经典算法模型实现，算法模型能根据已有波动规律曲线筛选出一组具有相似波动规律的监控指标曲线并按相似的程度大小（给出相似度分值）对他们进行排序即可。
- 要求最终结果通过前端展示，展示内容为算法计算出的具有较大相似度的监控指标曲线集合，可以提供筛选结果范围的功能。

备注：相似波动规律的监控指标曲线时间维度并不一定在同一个周期内，可能会存在先后。

二、解决方法

1、PD方法

模式距离 Pattern Distance，需要用下面的表示方法来构建时间序列。

- PLR：分段线性表示 Piecewise Linear Representation
- PMR：Pattern Model Representation

「模式Pattern」：时序序列的某一段的变化趋势，可按不同标准进行划分：

- 3种模式：上升、下降、不变 ---> $M = \{1, -1, 0\}$

- 7种模式：加速下降，水平下降，减速下降，不变，减速上升，水平上升，加速上升 ---> $M=\{-3,-2,-1,0,1,2,3\}$

（1）思路

1. 时间序列 ---> PLR

PLR根据时间序列的变化趋势，将其划分为子序列，每个子序列表示一个模式模型（pattern model），然后用直线表示每个子序列。
目的是得到时间序列变化趋势的抽象。

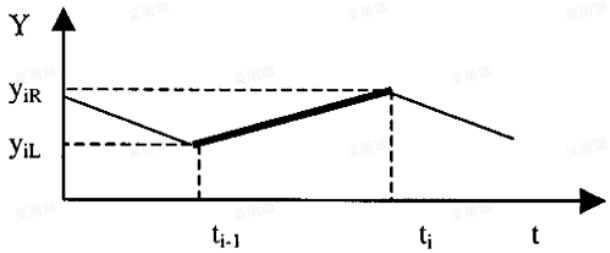


Figure 2: PLR of time series.

PLR表示：

$$S=\{(y_{1L},y_{1R},t_1),...,(y_{kL},y_{kR},t_k)\}$$

y_{iL},y_{iR} 分别是第i段的起始值和结束值，共有k段。

2. PLR ---> PMR

PMR是模式的时间序列，序列中的元素为(pattern value, time stamp)（pattern是时间戳前面那一段的模式），即

$$S=\{(p_1,\ t_1),\ ...,\ (p_N,t_N)\}$$

将PLR序列转化为PMR序列的过程：对PLR中的每一段求斜率，即可以判断其Pattern，若相邻的两段的变化趋势相同（假设都上升），那么在PMR中只用一个元素记录（即只记一个pattern value=1的元素）。

3. 【快捷方式】时间序列 ---> PMR

通过等分法划分节点，对每个节点可以计算其斜率，得到值，这样就得到了一串节点所对应值的时间序列，如[1,1,0,-1...]。

接着需要将相邻的相同模式进行合并（相邻的相同值的节点用一个节点表示），可得到[1,0,-1...]的序列，节点与节点之间用线性段表示。

4. 计算模式距离，作为相似度衡量值

设两个需对比相似度的序列PMR如下：

$$S_1=\{(p_{11},\ t_{11}),\ ...,\ (p_{1N},t_{1N})\}$$

$$S_2=\{(p_{21},\ t_{21}),\ ...,\ (p_{2M},t_{2M})\}$$

由于两个序列的时间戳（即模式的分割点）的集合可能不同，因此需要等模式数化（Equal pattern number process），让它们使用共同的分割点。实例如下图：

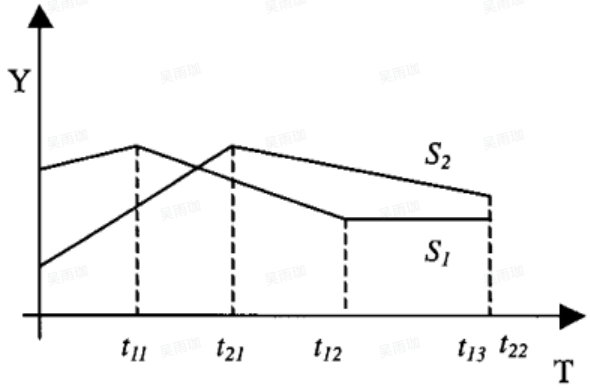


Figure 3: Non-EPN series.

$$S1 = \{(1, t_{11}), (-1, t_{12}), (0, t_{13})\}$$

$$S2 = \{(-1, t_{21}), (1, t_{22})\}$$

等模式数化后将他们变形为：

$$S1 = \{(1, t_1), (-1, t_2), (-1, t_3), (0, t_4)\}$$

$$S2 = \{(1, t_1), (1, t_2), (-1, t_3), (-1, t_4)\}$$

模式距离的计算：算出节点的欧式距离/绝对值距离之和（考虑每个pattern持续的时间，作为加权）

$$D_{(S1,S2)} = \sum_{i=1}^{k} t_{wi} * |m_{1i} - m_{2i}|, t_{wi} = \frac{ti}{t_N}$$

t_{wi} 是该模式所持续的时间的占比， t_i 为第i个pattern所跨越的时间长度， t_N 为总时间长度。

（2）优缺点

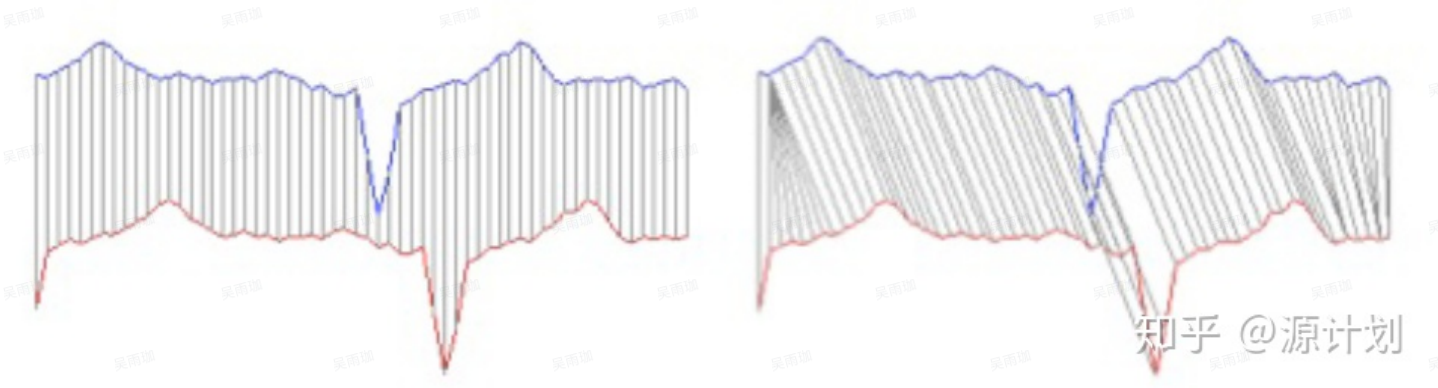
可以用于对比时间序列的变化趋势的相似度

2、DTW方法

动态时间规整 Dynamic Time Warping

将时间序列进行「规整」

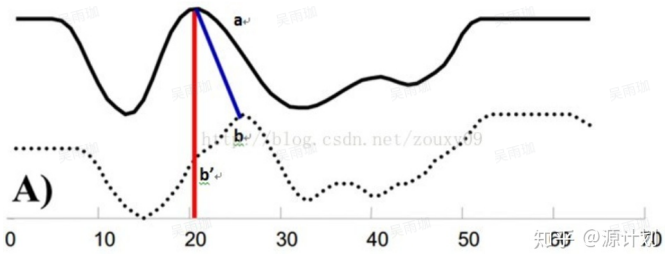
规整：延伸、缩短 ---> 使两个序列的形态尽可能一致，得到最大可能的相似度



左：欧式距离；右：DTW

（1）思路

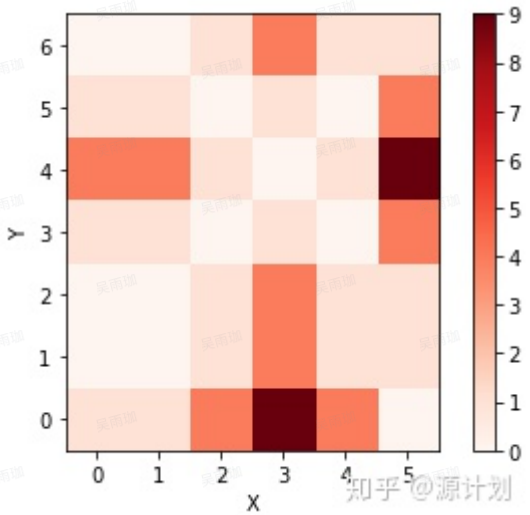
需要找到序列之间正确的对应点，再计算它们之间的距离。



1. 距离矩阵

由于我们不知道Q和C上的点的对应关系，因此需要计算出Q的每个点（|Q|=n）与C（|C|=m）的每个点之间的距离，也就得到了一个大小为n×m的距离矩阵。

我们可以把该矩阵可视化，将矩阵元素表示成n×m的格子（(0,0)在左下角），颜色越深表示距离越远，如下图。



2. 累加距离矩阵

（1）什么是累加距离

单知道分散的点之间距离最小还不够，我们需要找到两个时间序列的整体对应距离最小，即需要将不同对应关系的点之间距离之和加起来，作为整体距离，来找到整体距离最小。因此需要计算累加距离矩阵。

这个整体距离最小的对应关系可以体现在累加矩阵上，我们称之为「规整路径」，规整路径应满足以下限制条件：

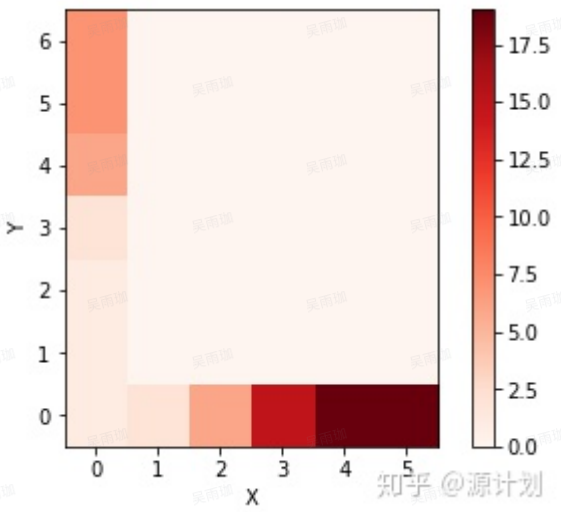
- 边界条件：必须从对角格子开始，到对角格子结束
  - 即Q和C的首尾点需要对齐
- 连续性：确定了一个格子，那么下一个格子必须是当前格子相邻的格子
  - 即不能略过Q或C中的任何一点，每一点都必须有对应
- 单调性
  - 即Q和C对应时可以拉伸/收缩，但不能翻折

（2）怎么计算累加距离矩阵

由于在找规整路径时，每走一个格子（即对应Q和C上的一对点）都需要和已经走过的路径联系起来，对比从那一条路径走到该格子所累加的距离最短（即确定上一个格子是左边，下边还是斜左下），因此需要使用「动态规划」算法。

步骤：

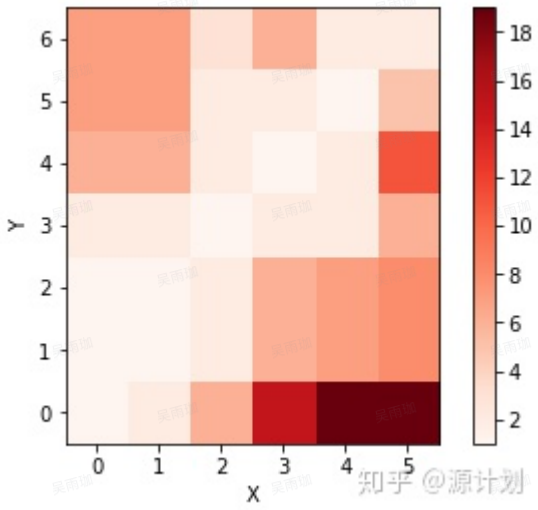
a. 计算基础路径：一直向右走（整个C都对应Q的第一个点）和一直向上走（整个Q都对应C的第一个点）



b. 依次计算每个格子的累加最短路径

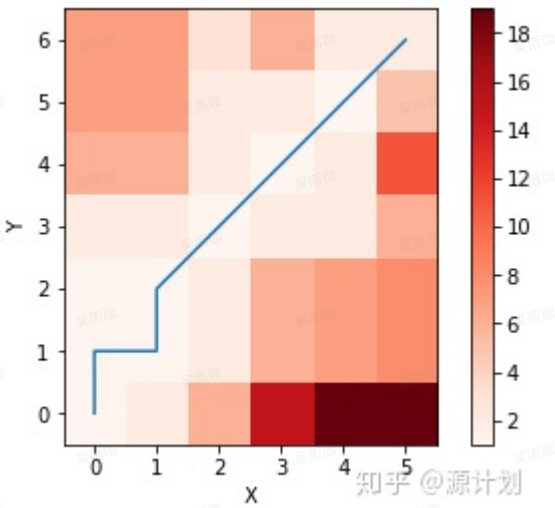
i. 路径的前进方向只有三个：向右，向上，斜右上

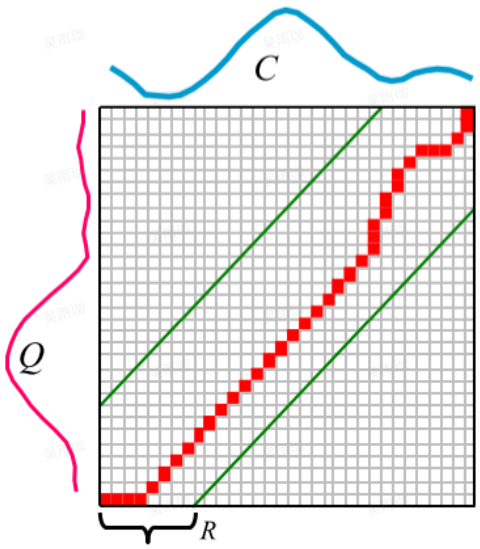
ii. $\gamma(i,j) = d(q_i, c_j) + min\{\gamma(i-1,j), \gamma(i,j-1), \gamma(i-1,j-1)\}$



c. 根据累加距离矩阵，找出规整路径

使用回溯方法，从右上角（Q与C尾）开始，往前依次确定路径。





（2）伪码

复杂度

Python实现

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
x = np.array([1, 1, 2, 3, 2, 0])
y = np.array([0, 1, 1, 2, 3, 2, 1])
plt.plot(x,'r', label='x')
plt.plot(y, 'g', label='y')
plt.legend()
# 计算距离矩阵
distances = np.zeros((len(y), len(x)))
for i in range(len(y)):
    for j in range(len(x)):
        distances[i,j] = (x[j]-y[i])**2
# 输出距离矩阵
# Out[60]:
# array([[1., 1., 4., 9., 4., 0.],
#        [0., 0., 1., 4., 1., 1.],
#        [0., 0., 1., 4., 1., 1.],
#        [1., 1., 0., 1., 0., 4.],
#        [4., 4., 1., 0., 1., 9.],
#        [1., 1., 0., 1., 0., 4.],
#        [0., 0., 1., 4., 1., 1.]])

# 可视化
def distance_cost_plot(distances):
    plt.imshow(distances, interpolation='nearest', cmap='Reds')
    plt.gca().invert_yaxis()#倒转y轴，让它与x轴的都从左下角开始
    plt.xlabel("X")
    plt.ylabel("Y")
#    plt.grid()
    plt.colorbar()
distance_cost_plot(distances)

# 计算累加距离矩阵
accumulated_cost = np.zeros((len(y), len(x)))
accumulated_cost[0,0] = distances[0,0]
# a. 计算基础路径
for i in range(1, len(x)):
    accumulated_cost[0,i] = distances[0,i] + accumulated_cost[0, i-1]
for i in range(1, len(y)):
    accumulated_cost[i,0] = distances[i, 0] + accumulated_cost[i-1, 0]
# b. 计算完整路径
for i in range(1, len(y)):
    for j in range(1, len(x)):
        accumulated_cost[i, j] = min(accumulated_cost[i-1, j-1], accumulated_cos
# c. 回溯找出规整路径
```

```
path = [[len(x)-1, len(y)-1]]
i = len(y)-1
j = len(x)-1
while i>0 and j>0:
    if i==0:
        j = j - 1
    elif j==0:
        i = i - 1
    else:
        if accumulated_cost[i-1, j] == min(accumulated_cost[i-1, j-1], accumulat
            i = i - 1#来自于左边
        elif accumulated_cost[i, j-1] == min(accumulated_cost[i-1, j-1], accumul
            j = j-1#来自于下边
        else:
            i = i - 1#来自于左下边
            j= j- 1
    path.append([j, i])
path.append([0,0])
path
Out[89]: [[5, 6], [4, 5], [3, 4], [2, 3], [1, 2], [1, 1], [0, 1], [0, 0]]
path_x = [point[0] for point in path]
path_y = [point[1] for point in path]
distance_cost_plot(accumulated_cost)
plt.plot(path_x, path_y)
```

（3）优缺点

对序列的压缩/拉伸不敏感

三、代码实现

本项目选择用DTW算法来实现监控数据相似度对比的代码部分，原因如下：

1. DTW的高效性

在论文Searching and Mining Trillions of Time Series Subsequences under Dynamic Time Warping（参考文献 第二篇）中1.2.2节论证了"Dynamic Time Warping is the Best Measure"。

2. DTW的易用性

python为DTW算法提供了两个经典的库，分别是dtw和dtw-python

- dtw库的功能少但简单容易理解
- dtw-python的功能齐全并提供了清晰的作图。

可以使项目代码易读、规范。

1、Python的dtw-python库

1. 下载：`pip install dtw-python`
2. 模块引入：`from dtw import *`
3. API：

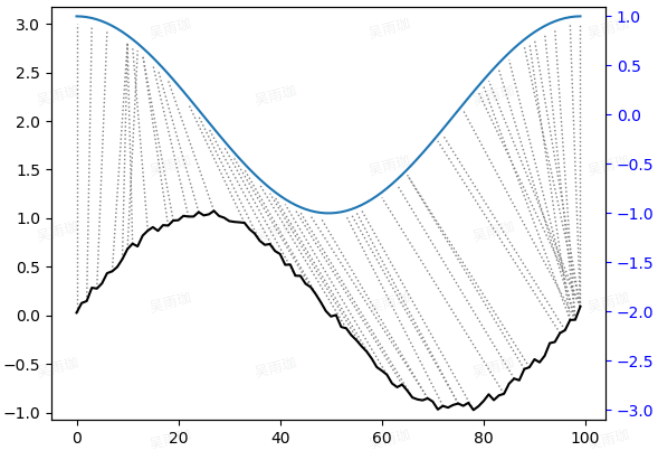
https://dynamictimewarping.github.io/py-api/html/

Welcome to the dtw-python package — The dtw-python package 1.3.0 documentation

dtw-python 1.3.0 🏠 The DTW suite Docs countPaths countPaths() dtw dtw() dtwPlot dtwPlot() dtwPlotAlignment dtwPlotAlignment() dtwPlotDensity dtwPlotDensity() dtwPlotThreeWay dtwPlotThreeWay() dtwPlotT

示例：

```
import numpy as np

## A noisy sine wave as query
idx = np.linspace(0,6.28,num=100)
query = np.sin(idx) + np.random.uniform(size=100)/10.0

## A cosine is for template; sin and cos are offset by 25 samples
template = np.cos(idx)

## Find the best match with the canonical recursion formula
from dtw import *
alignment = dtw(query, template, keep_internals=True)

## Display the warping curve, i.e. the alignment curve
alignment.plot(type="threeway")

## Align and plot with the Rabiner-Juang type VI-c unsmoothed recursion
dtw(query, template, keep_internals=True,
    step_pattern=rabinerJuangStepPattern(6, "c"))\
    .plot(type="twoway",offset=-2)

## See the recursion relation, as formula and diagram
print(rabinerJuangStepPattern(6,"c"))
rabinerJuangStepPattern(6,"c").plot()

## And much more!
```



2、项目代码

核心函数：

- dtw.dtw(x, y=None, dist_method='euclidean', step_pattern='symmetric2', window_type=None, window_args={}, keep_internals=False, distance_only=False, open_end=False, open_begin=False)

  Parameters:

  - **x** – 需要查询的序列-query
  - **y** – 用于对比的序列-reference
  - **dist_method** – pointwise (local) distance function to use.
  - **step_pattern** – 选择dtw算法递归的不同变种
  - **window_type** –
  - **window_args** – additional arguments, passed to the windowing function
  - **open_begin** – 不要求x和y一定从首部开始对应（但若不是open_end则需要x和y尾部对应），只要在y中找到与x最匹配的一段即可。
  - **open_end** – 如上同理
  - **keep_internals** – 保留计算时得到中间结构，如累积距离矩阵、输入等（可以用于作图）
  - **distance_only** – 只计算距离，不必回溯来得到对应的结果。用于加快计算。

  Return type:

  An object of class `DTW` . See docs for the corresponding properties.

```
# 相似度比对
@search_blue.route("/similarity_search", methods=['POST'])
def similarity_search():
    rank = int(request.form['rank'])          # top10=>rank=10  top5=>rank=5
```

```
    ab_node = request.form['ab_node']           # 异常数据表名
    start_time = request.form['start_time']     # 开始时间
    end_time = request.form['end_time']         # 截止时间
    # 相似度分析，获取相似度最高的rank个时间序列
    res = process_similarity(rank, ab_node, start_time, end_time)
    return res

# 相似度分析函数
# 返回值：根据rank返回一个list，元素为相似度从高到低的<node_name, simi_value, x_data_
# simi_value：dtw算法算出的距离，距离越大表示相似度越低
# x_data_list：该表截取出的异常时间段的时间点list
# y_data_list：该表截取出的异常时间段对应的值list
def process_similarity(rank: int, ab_node:str, start_time: str, end_time: str):
    # 得到表名
    nodes_name = get_similarity_node()
    # 得到异常序列和其他序列在异常时间段的数据
    ret = get_similarity_data(nodes_name, ab_node, start_time, end_time)
    nodes_name.remove(ab_node)
    query = ret.pop(ab_node)    # 查询序列，即异常序列（0.csv）
    refers = ret                # 用于对比的序列，即其他所有序列

    simi_list = []    # 返回列表
    while(len(refers)):
        curr_node = nodes_name.pop()      # 当前用于对比的序列表名
        refer = refers.pop(curr_node)     # 当前用于对比的序列数据
        # dtw算法对比两个序列的相似度
        alignment = dtw(x=query['y_data_list'],
                        y=refer['y_data_list'],
                        dist_method="euclidean",
                        step_pattern=asymmetric,
                        keep_internals=True)
        # alignment.plot(type="twoway", offset=1, ylab=curr_node)
        # 将相似度分析结果加入simi_list
        node_simi = dict()
        node_simi['node_name'] = curr_node
        node_simi['simi_value'] = alignment.__getattribute__('distance')
        node_simi['x_data_list'] = refer['x_data_list']
        node_simi['y_data_list'] = refer['y_data_list']
        simi_list.append(node_simi)
    # 按相似度大小排序，只返回rank个最相似序列
    simi_list.sort(key=lambda e:e['simi_value'])
    simi_list = simi_list[:rank]

    # 将异常序列加入返回值（即加入表0）
    node_simi = dict()
    node_simi['node_name'] = "abnormal0"
    node_simi['x_data_list'] = simi_list[0]['x_data_list']
```

```
52      # 由于考虑波动提前/延后的情况，用于对比的序列截取时间首尾边界被拓宽
53      # 查询序列需要补全首尾，使序列长度相同
54      y_data_list=[-1,-1]
55      y_data_list.extend(query['y_data_list'])
56      y_data_list.extend([-1,-1])
57      node_simi['y_data_list'] = y_data_list
58      simi_list.append(node_simi)
59      return simi_list
60
61  # 获取相似度比对的数据表名
62  def get_similarity_node(filename="AIBMA-ES-CLUSTER-20230423"):
63      # 设置数据库
64      database_name = filename.replace("-", "_")
65      database_sql = "USE " + database_name
66      cursor.execute(database_sql)
67      db.commit()
68      res = []
69      # 查询表名
70      sql = "select table_name from information_schema.tables where table_schema='
71      cursor.execute(sql)
72      ret = cursor.fetchall()
73      for item in ret:
74          res.append(item[0])
75      return res
76
77  # 根据输入的表名获取相似度信息
78  def get_similarity_data(nodes_name:list, ab_node:str, start_time:str, end_time:s
79      database_sql = "USE AIBMA_ES_CLUSTER_20230423"
80      cursor.execute(database_sql)
81      db.commit()
82      res = dict()
83      # 根据需要查询的节点名称去查询
84      for node_name in nodes_name:
85          res[node_name] = dict()
86          datas = select_node_similarity_data(node_name, ab_node == node_name, sta
87          res[node_name]['x_data_list'] = datas[0]
88          res[node_name]['y_data_list'] = datas[1]
89      return res
90
91  # 从表中截取序列
92  def select_node_similarity_data(table_name:str, isQuery:bool, start_time:str, en
93      # 找到开始序列的首尾id
94      sql = "select id from " + table_name + " where date=\'" + start_time +"\' or
95      cursor.execute(sql)
96      datas = cursor.fetchall()
97      start_id, end_id = datas[0][0], datas[1][0]
98      # 查询所有表的数据
```

```
    sql = "select id,date,value from " + table_name + " where date!=''"
    cursor.execute(sql)
    datas = cursor.fetchall()
    x_data_list = []
    y_data_list = []
    # 按照异常的开始时间和结束时间截取数据，得到时间点x_data_list和对应值y_data_list
    # 对于用于对比的序列，考虑波动提前/延后的情况，将对比时间段边界放宽
    for data in datas:
        if(not isQuery and data[0] >= start_id-2 and data[0] < start_id):
            x_data_list.append(data[1])
            y_data_list.append(float(data[2]))
        elif(data[0] >= start_id and data[0] <= end_id):
            x_data_list.append(data[1])
            y_data_list.append(float(data[2]))
        elif(not isQuery and data[0] > end_id and data[0] <= end_id + 2):
            x_data_list.append(data[1])
            y_data_list.append(float(data[2]))
        elif(data[0] > end_id+2):
            break
    return x_data_list, y_data_list
```

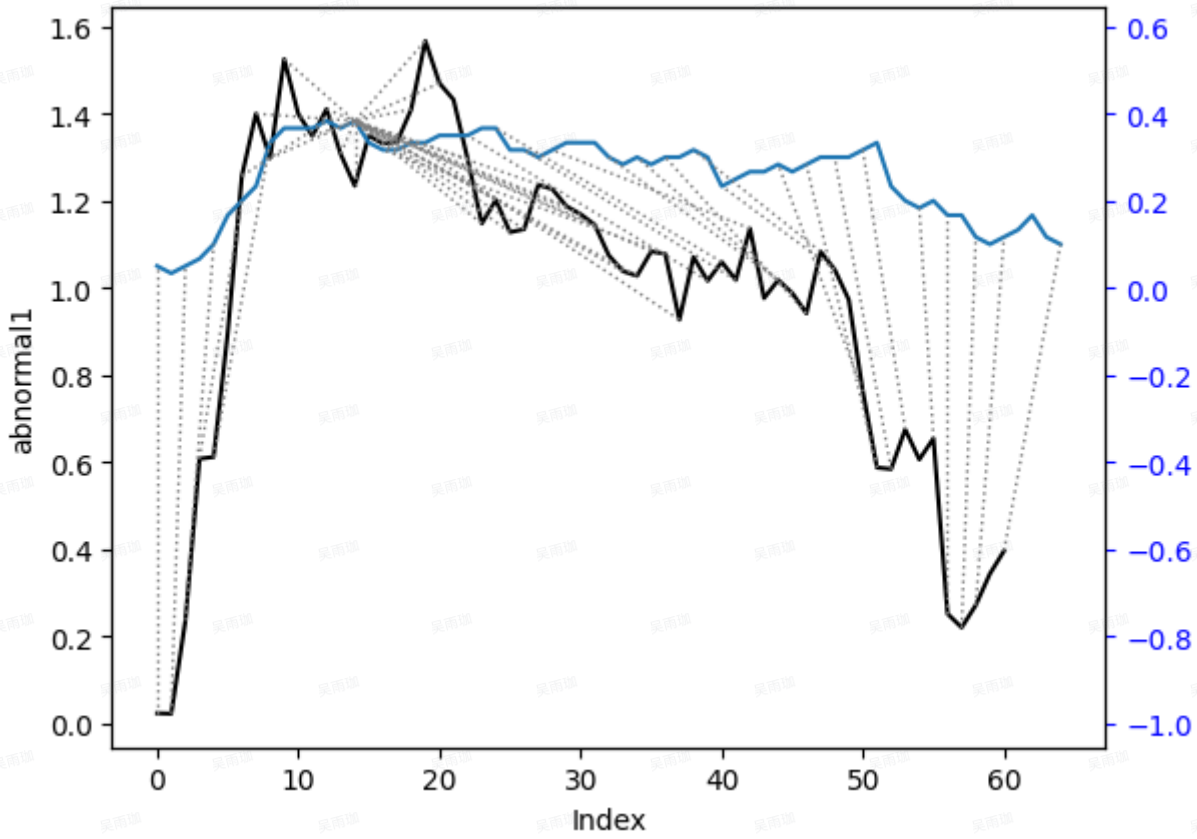
3、效果展示

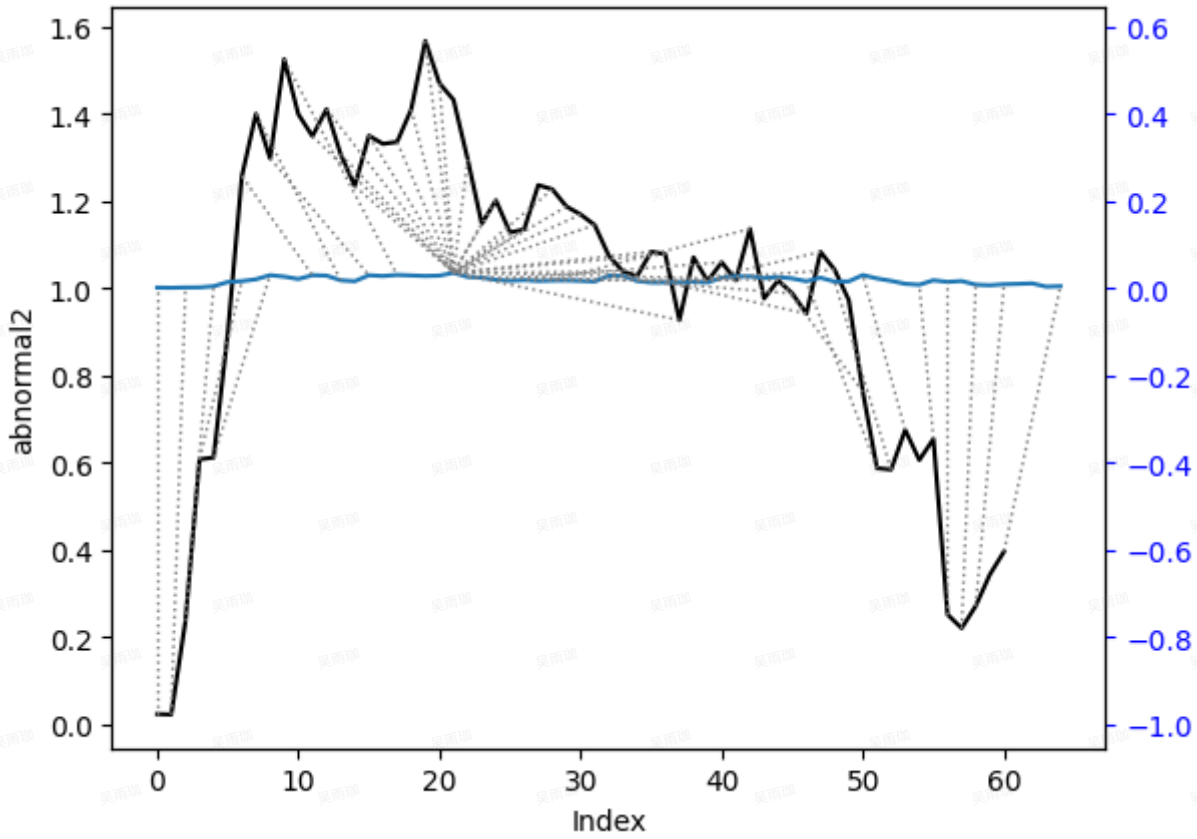
测试用例：

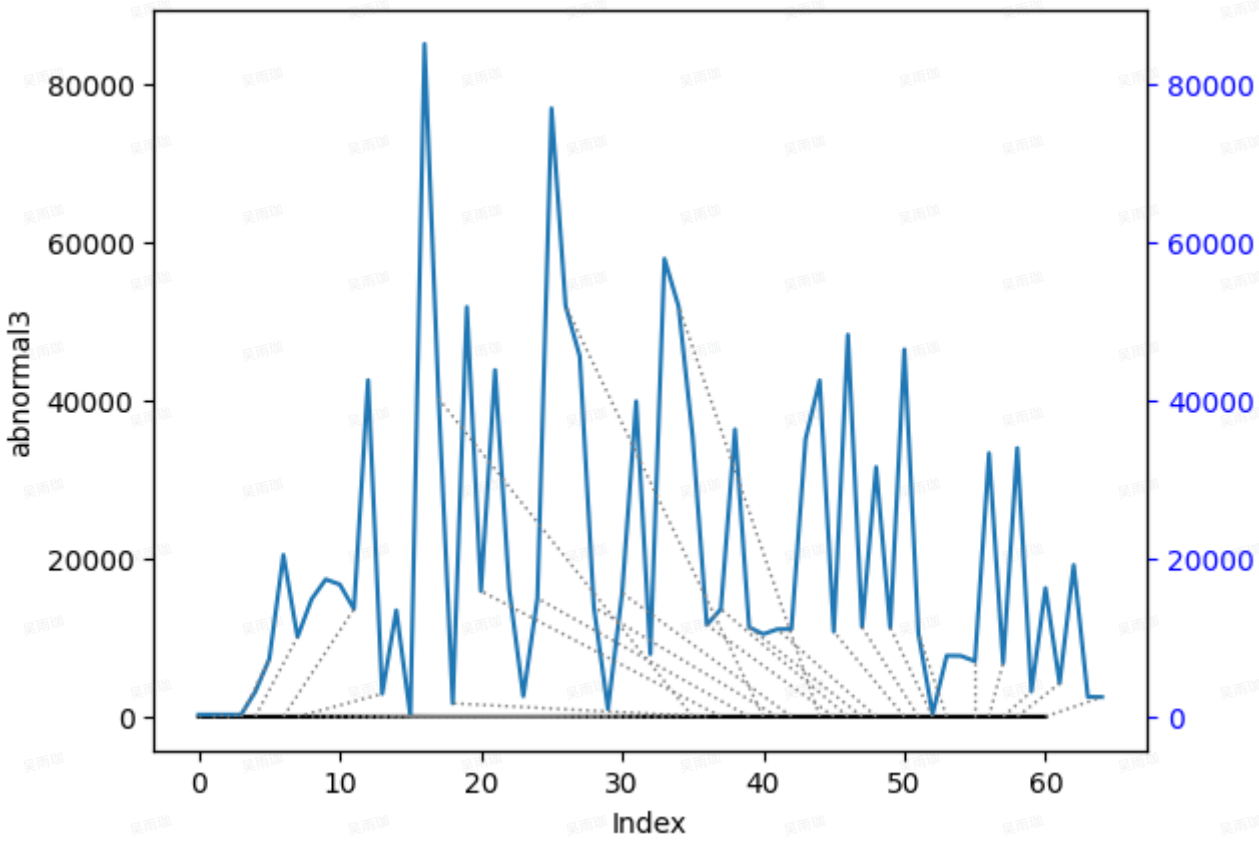
```
if __name__ == '__main__':
    process_similarity(20, 'abnormal0', '2023/4/23 15:20', '2023/4/23 17:20')
```

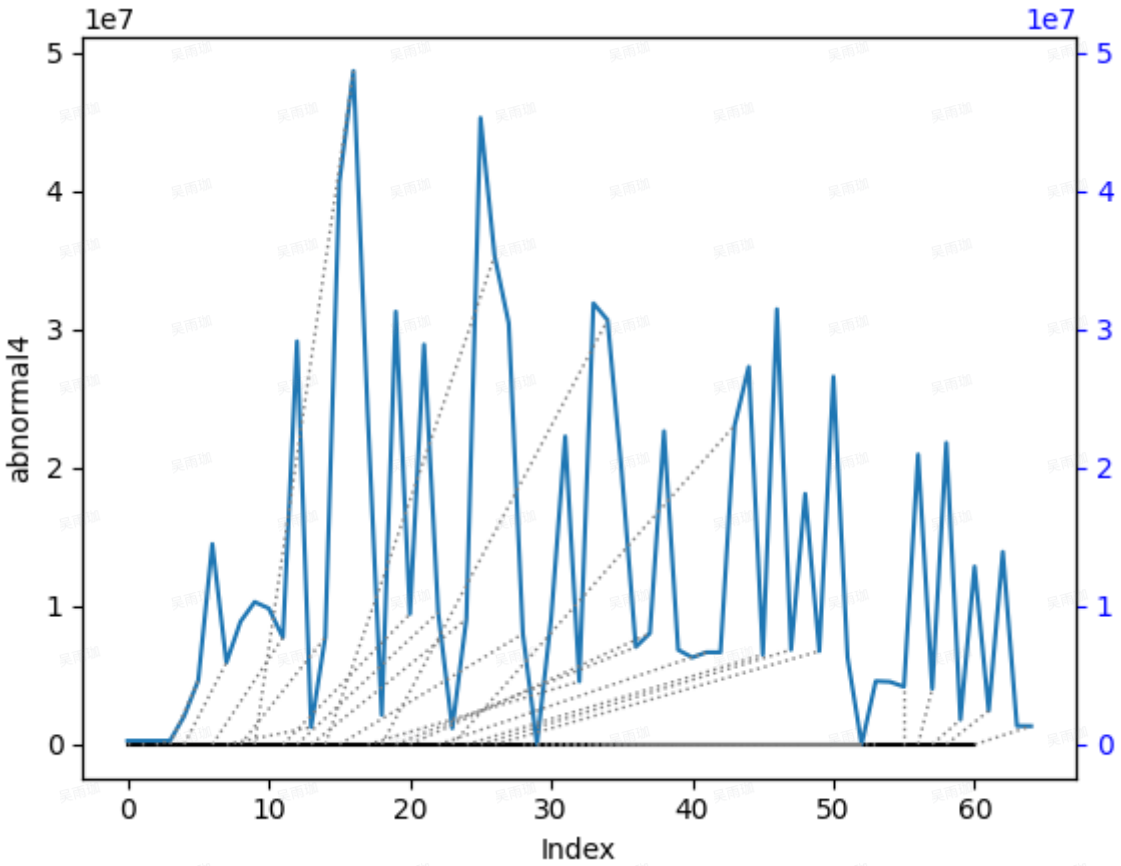
绘图结果：

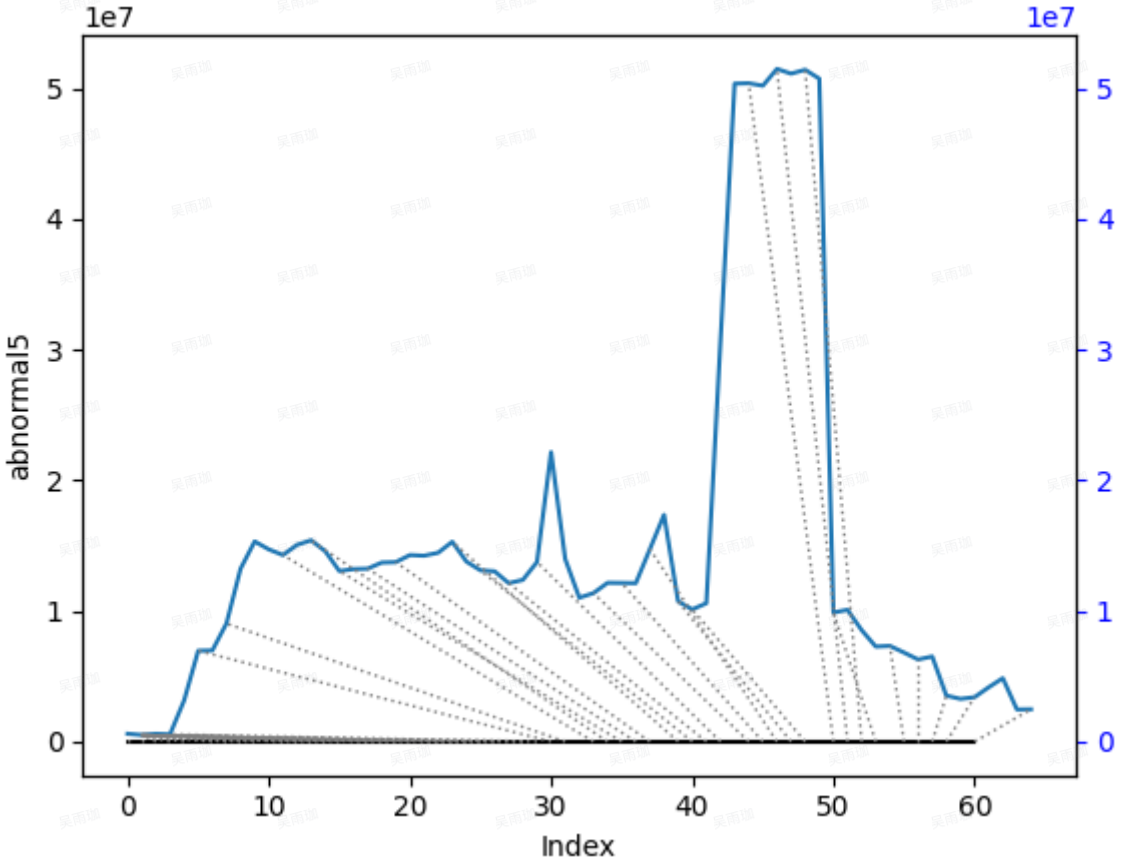
黑色的曲线表示0，蓝色曲线表示其他表（表名见左）
左边黑色刻度对应黑色曲线，蓝色刻度对应蓝色曲线

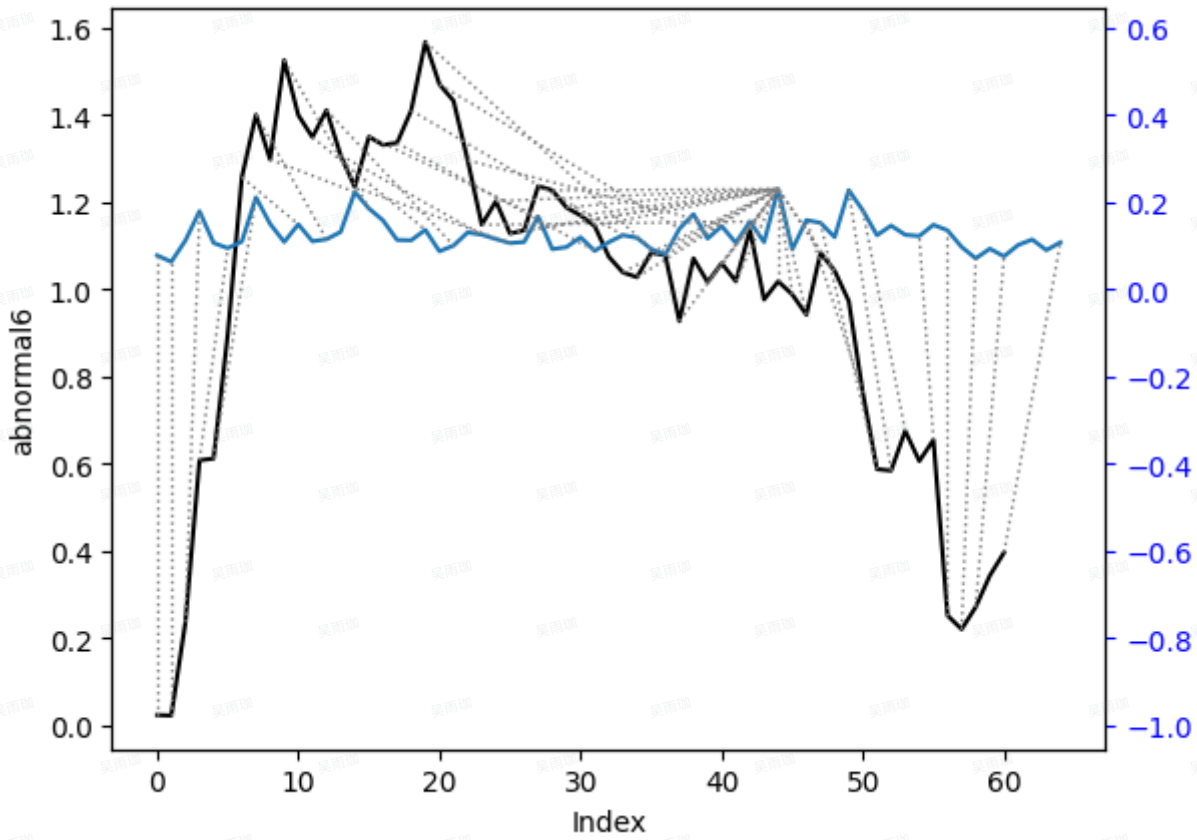

abnormal1
Index
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
10
20
30
40
50
60

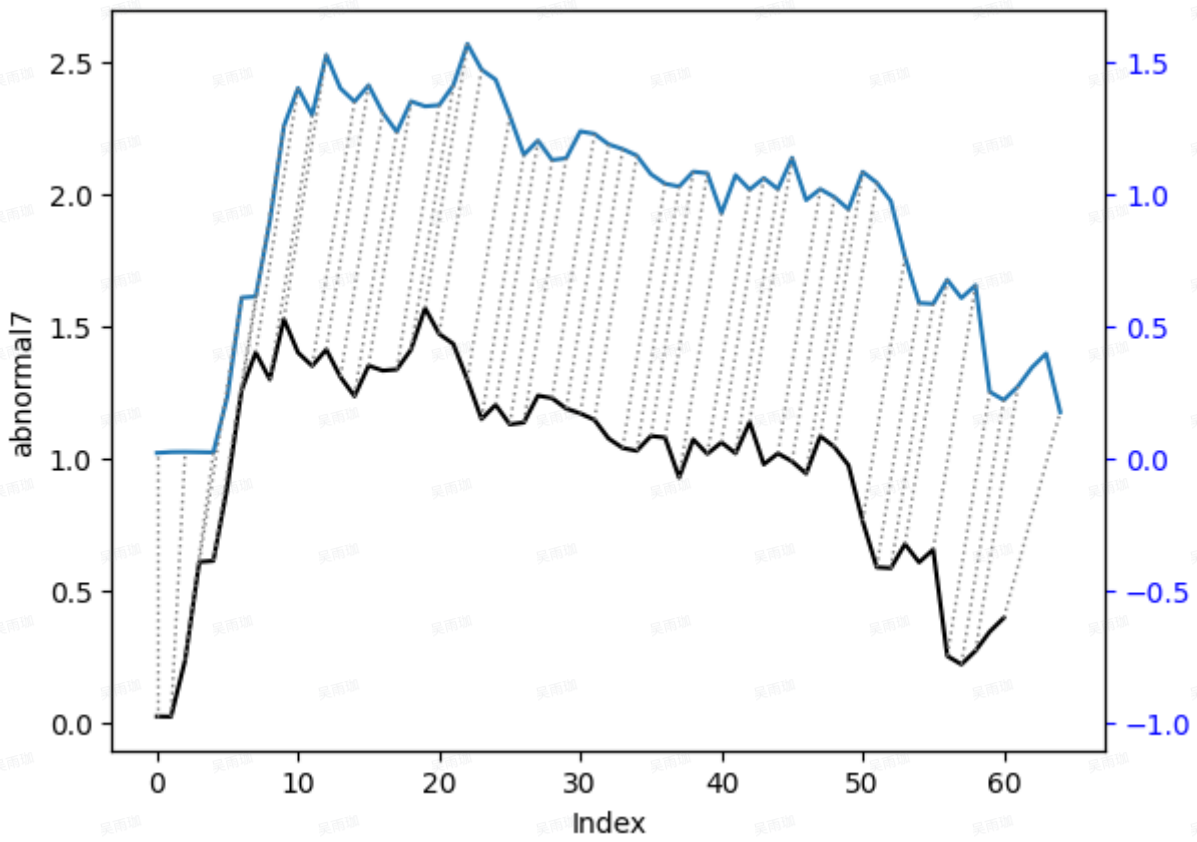

abnormal2
Index
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
10
20
30
40
50
60

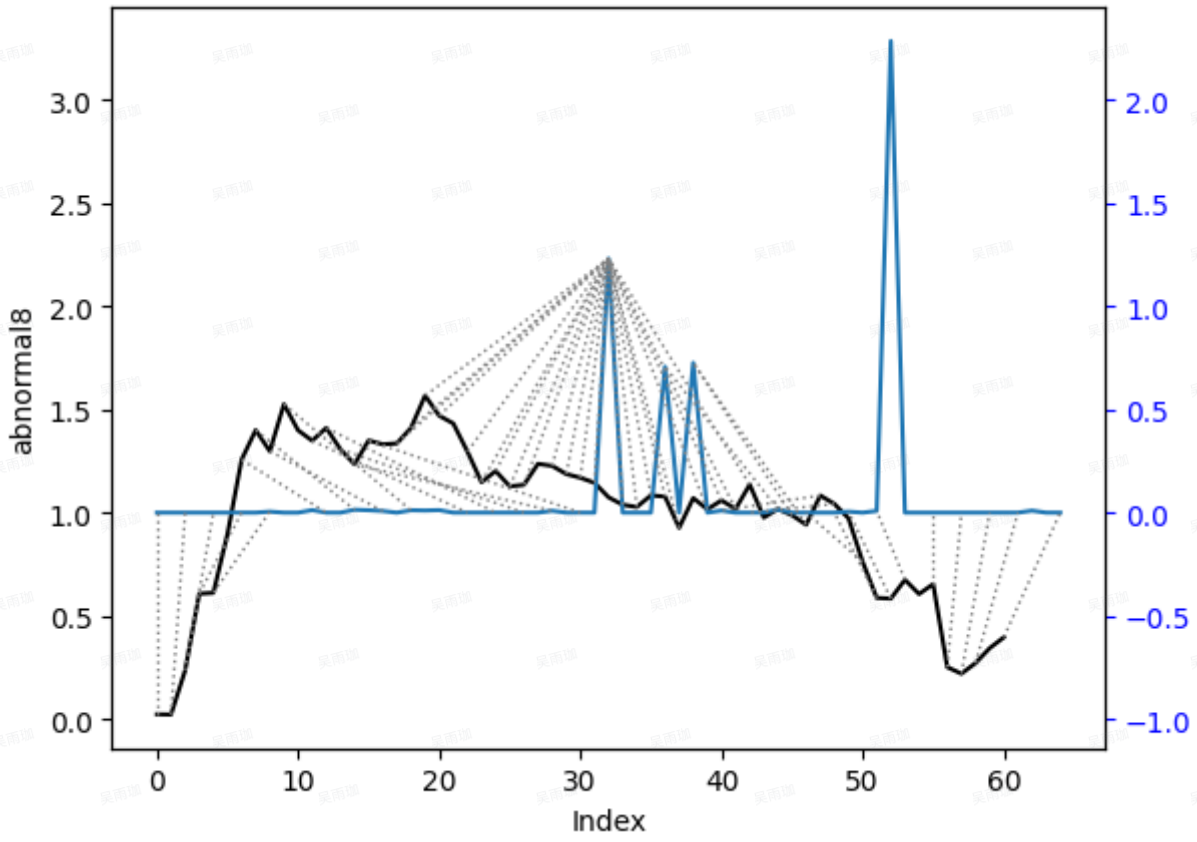

abnormal3
Index
0
20000
40000
60000
80000
0
10
20
30
40
50
60

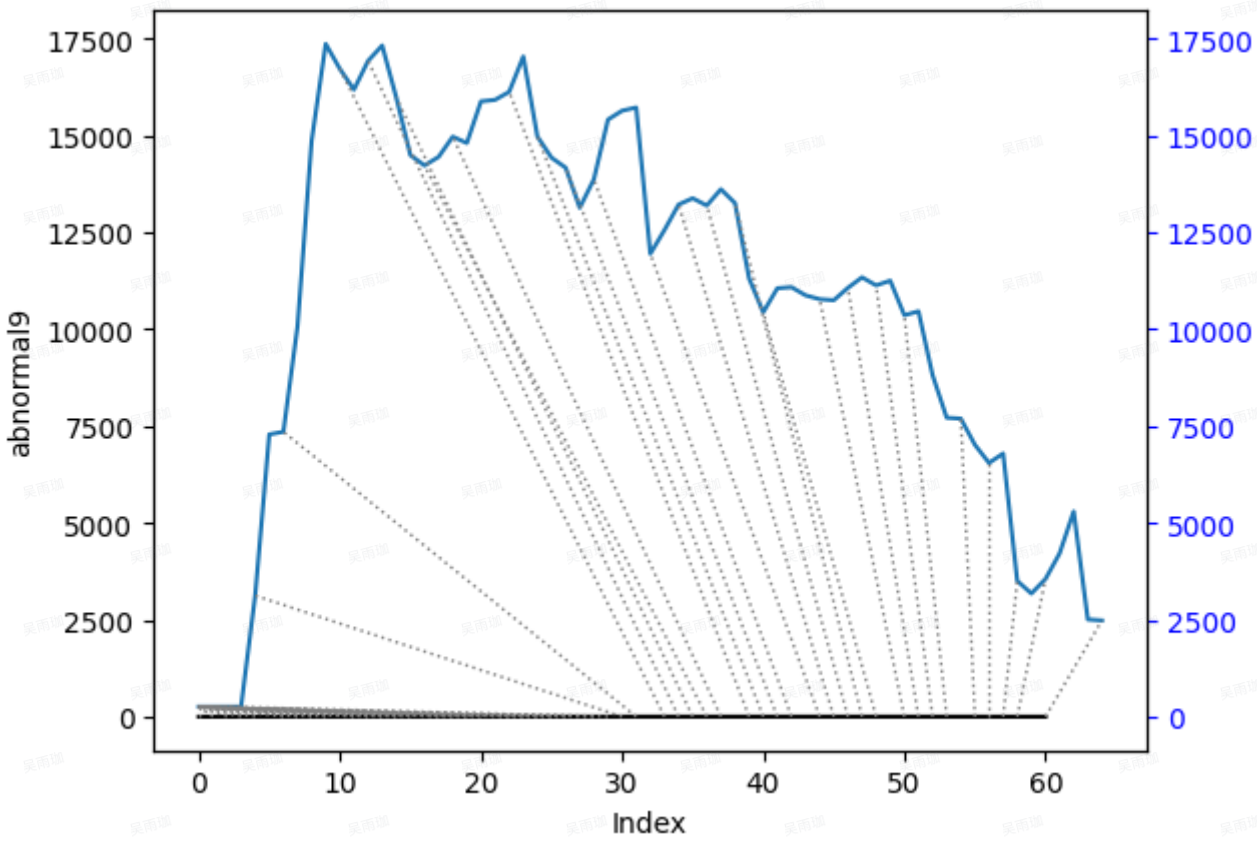

1e7
abnormal4
Index
0
1
2
3
4
5
0
10
20
30
40
50
60

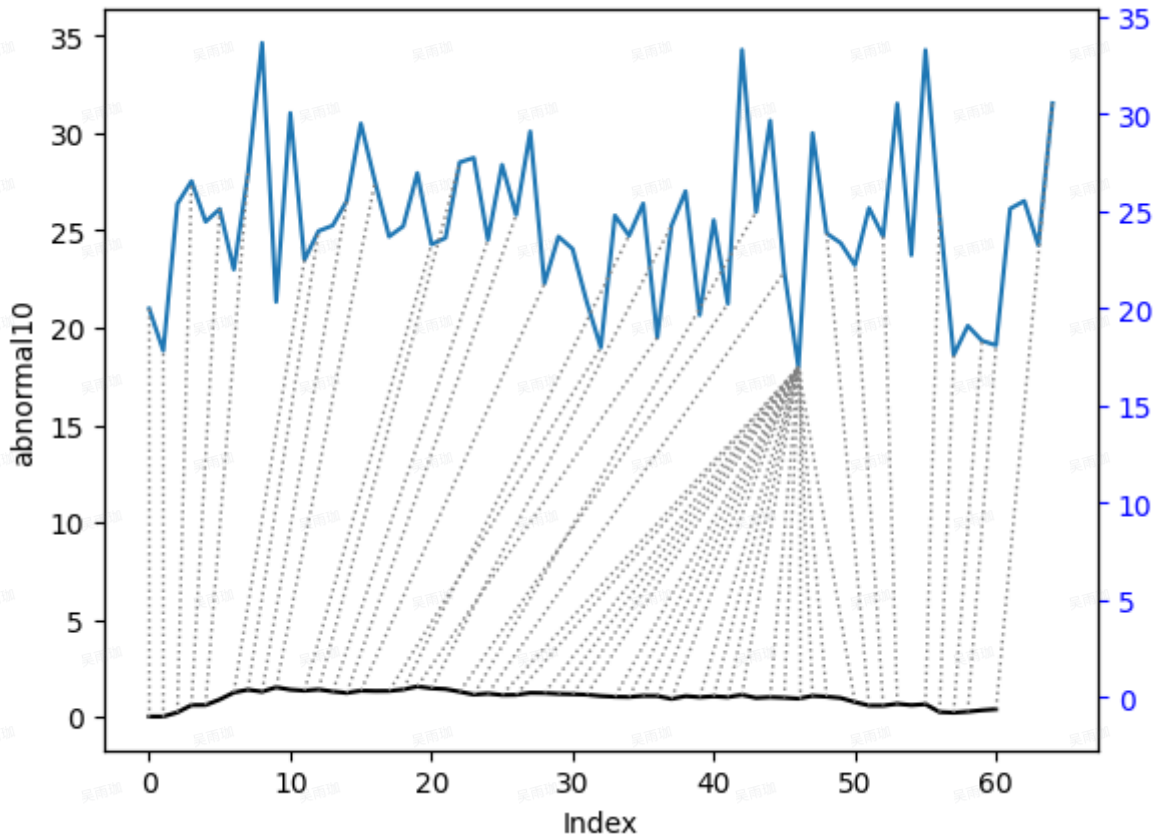

1e7
1e7
abnormal5
Index

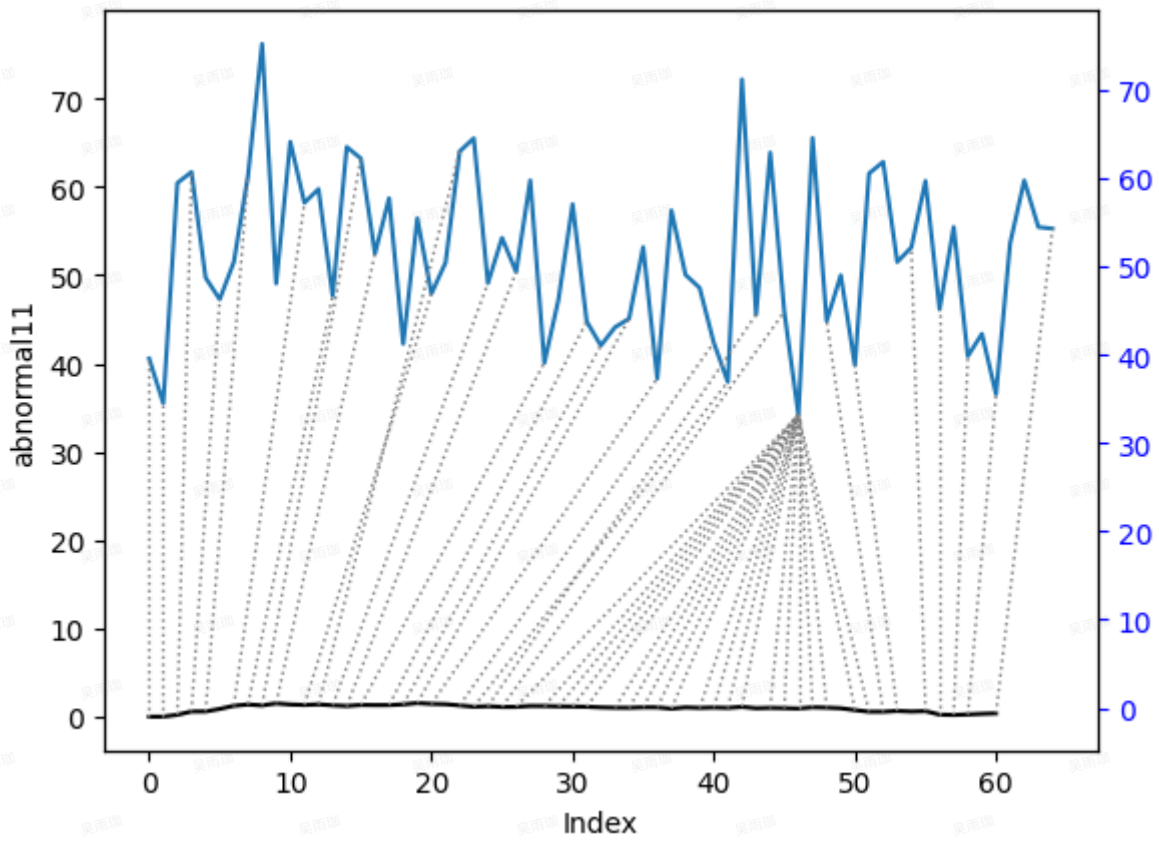

abnormal6
Index

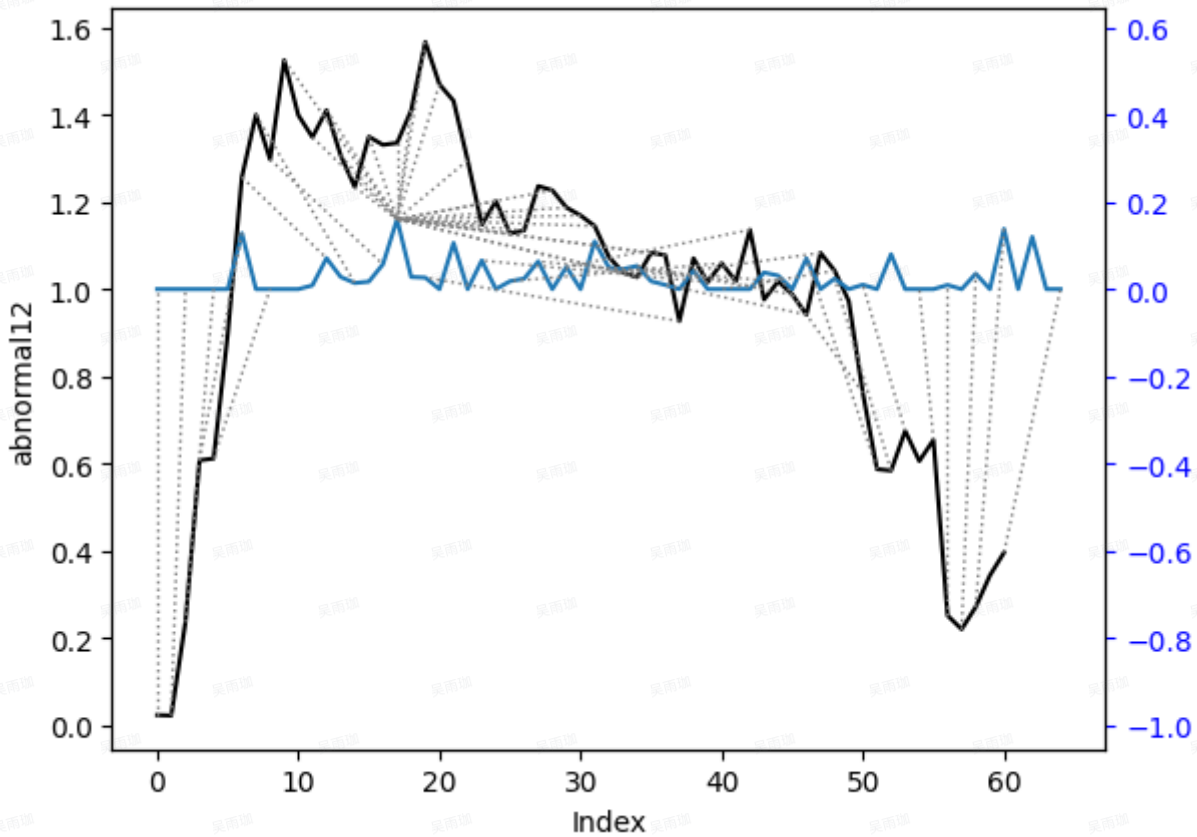

abnormal7
Index
0.0
0.5
1.0
1.5
2.0
2.5
−1.0
−0.5
0.0
0.5
1.0
1.5
0
10
20
30
40
50
60

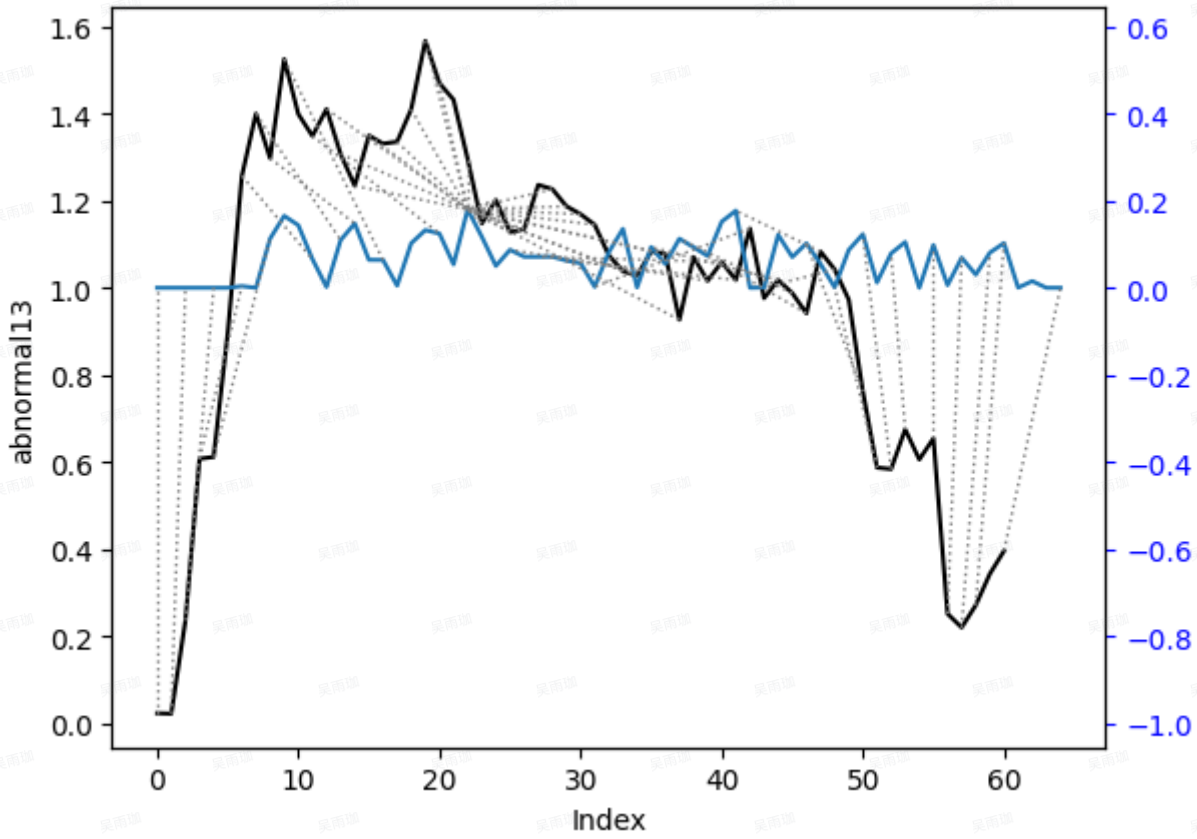

abnormal8
Index
0.0
0.5
1.0
1.5
2.0
2.5
3.0
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
0
10
20
30
40
50
60

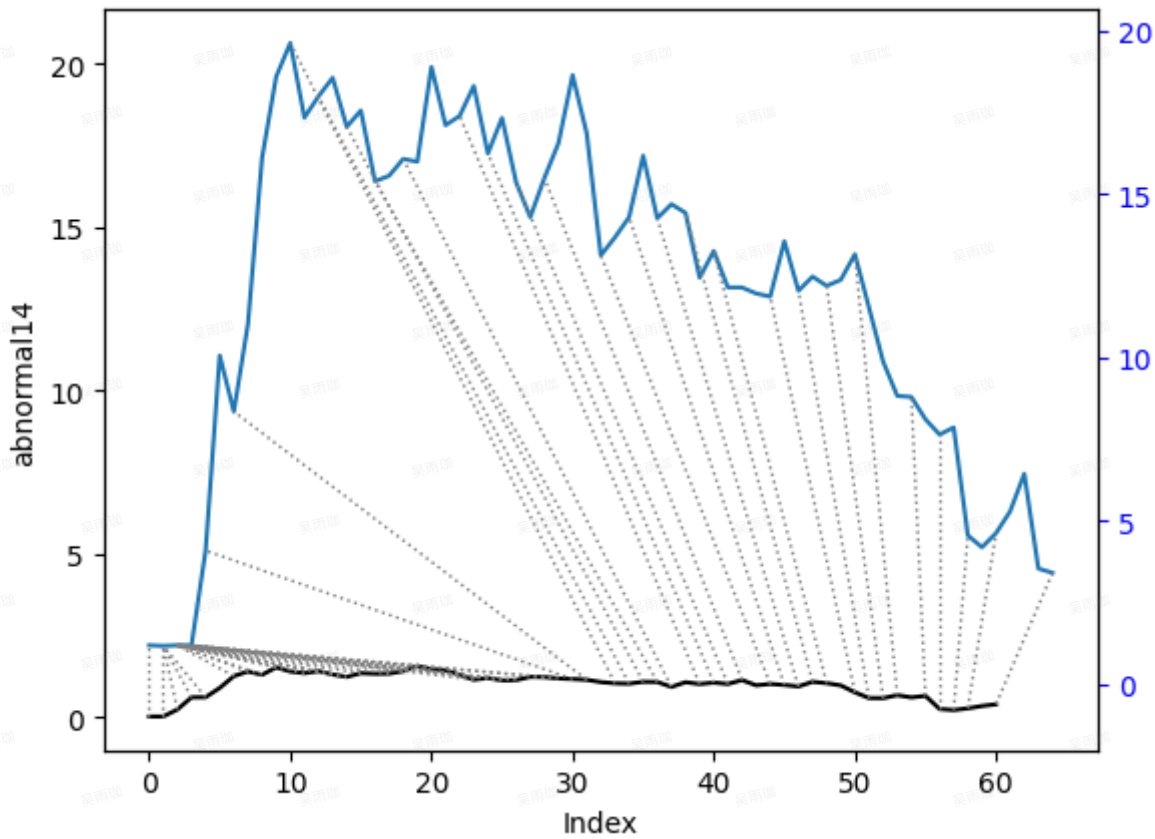

abnormal9
Index
17500
15000
12500
10000
7500
5000
2500
0
0
10
20
30
40
50
60

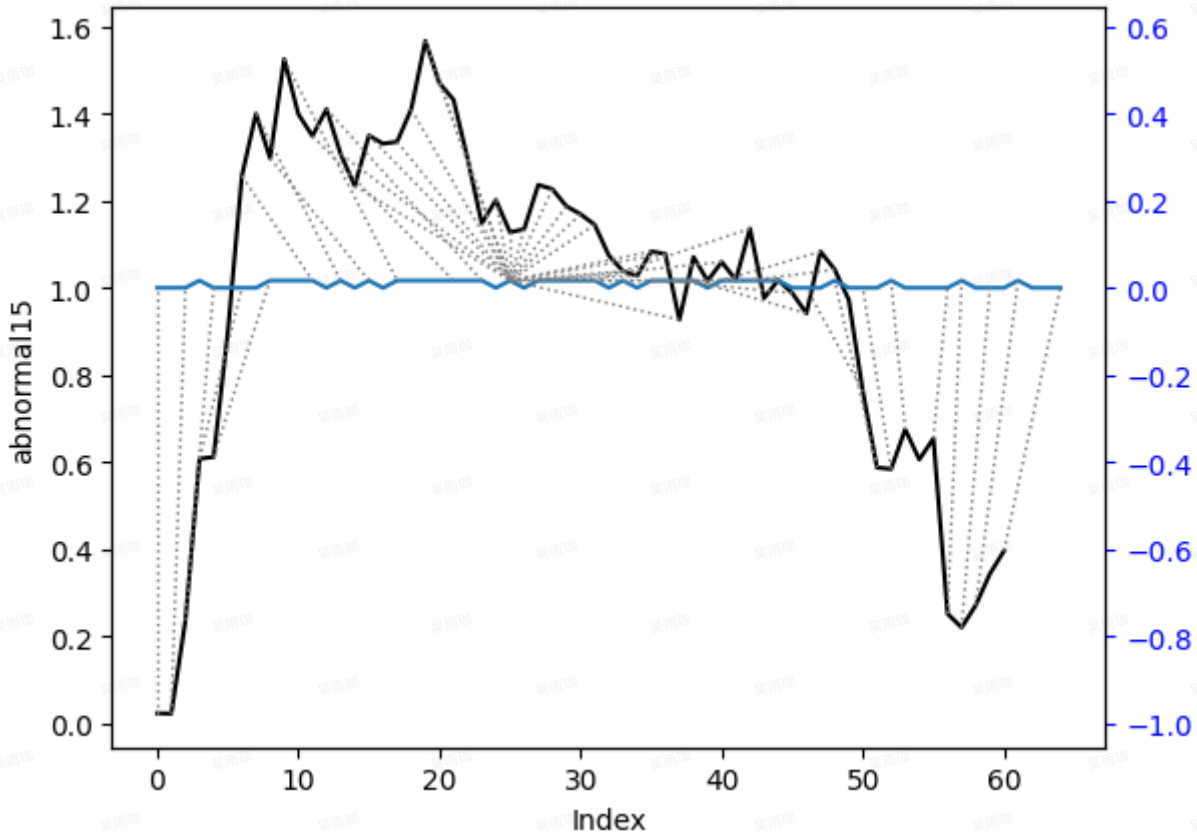

abnormal10
Index
35
30
25
20
15
10
5
0
0
10
20
30
40
50
60

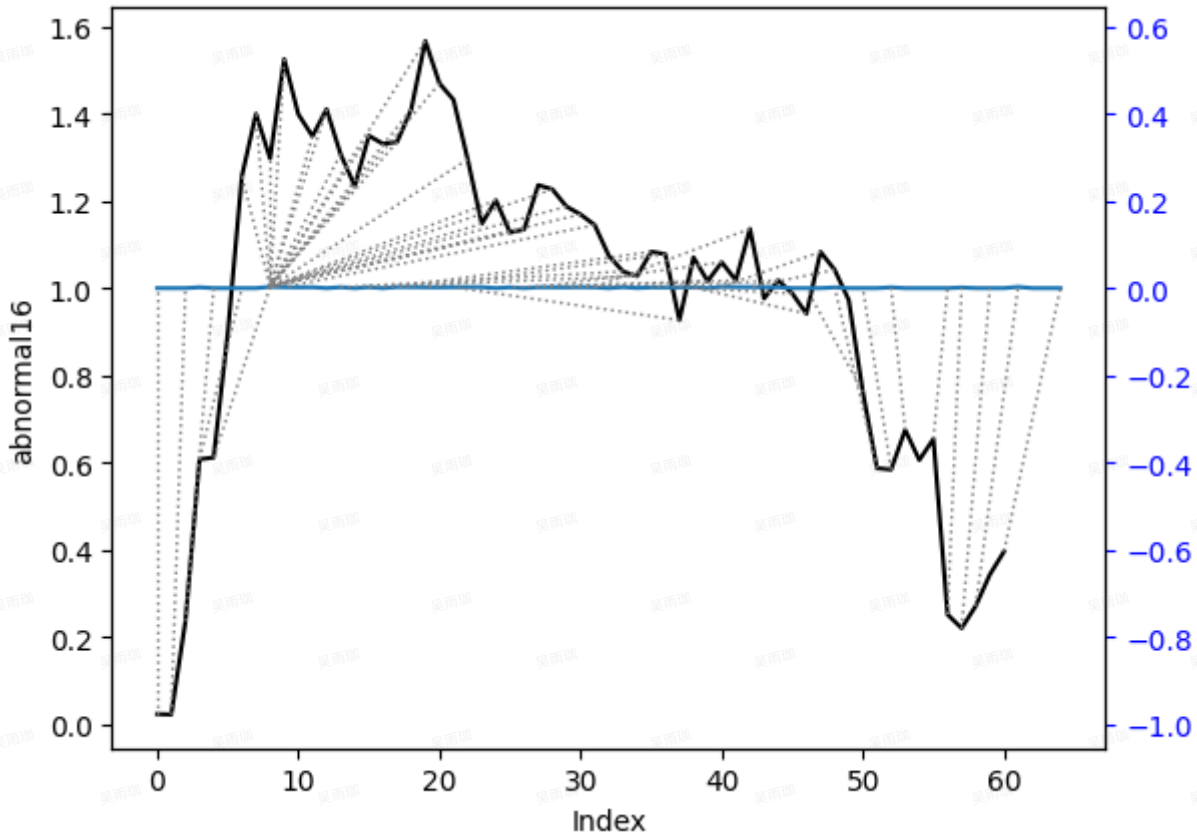

abnormal11
Index
0
10
20
30
40
50
60
70

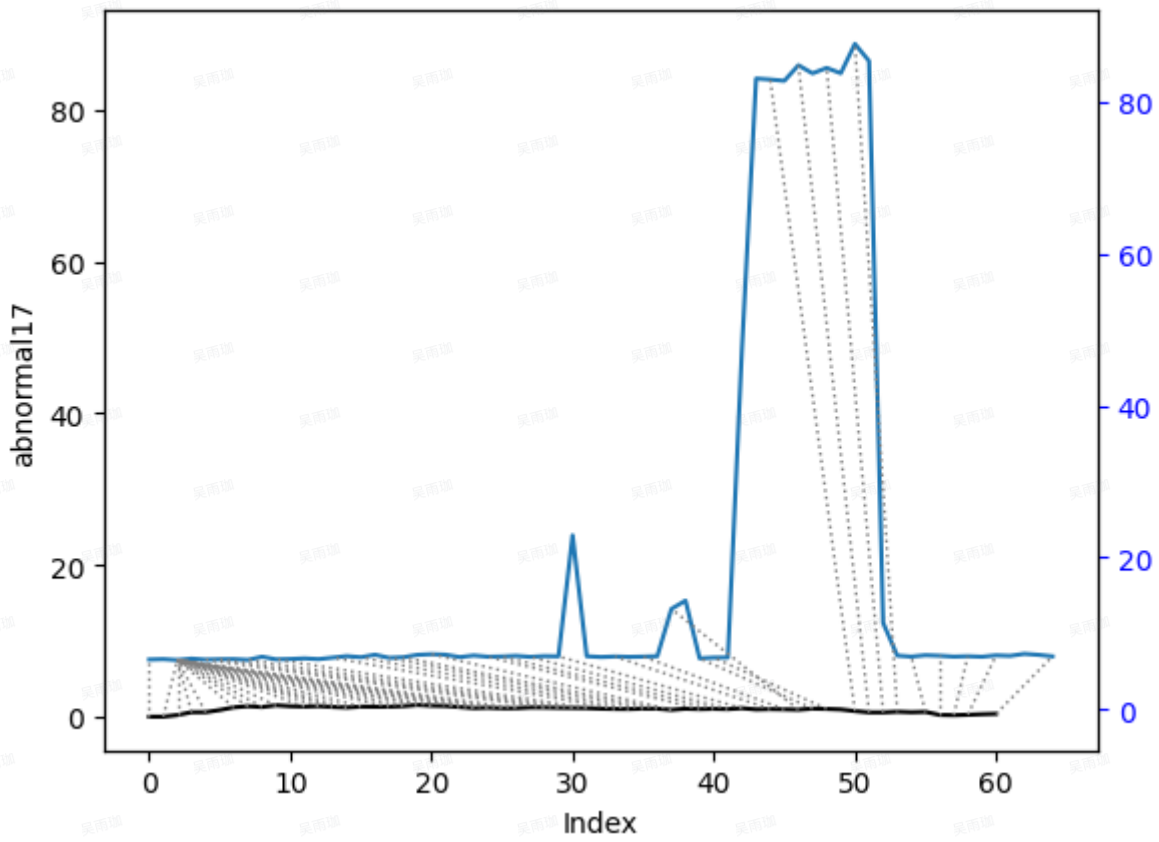

abnormal12
Index
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
−1.0
−0.8
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6

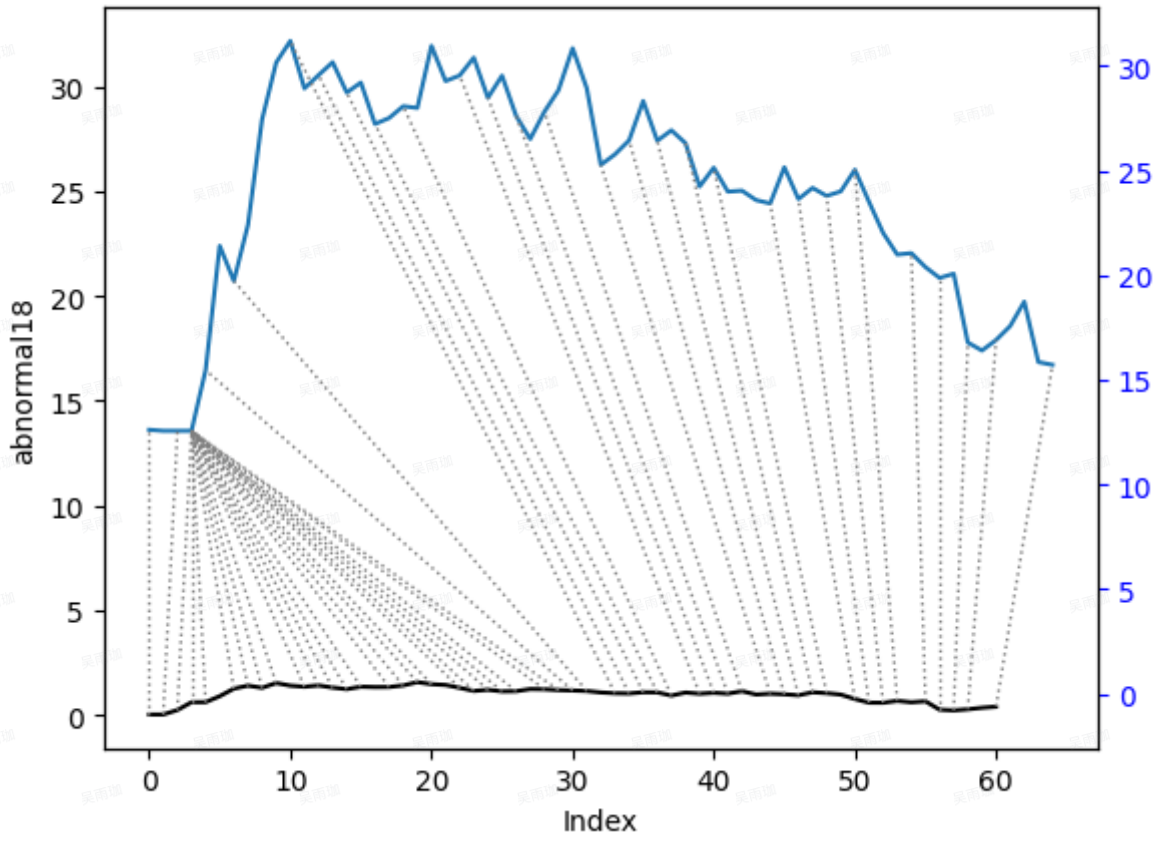

abnormal13
Index
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
−1.0
−0.8
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6
0
10
20
30
40
50
60

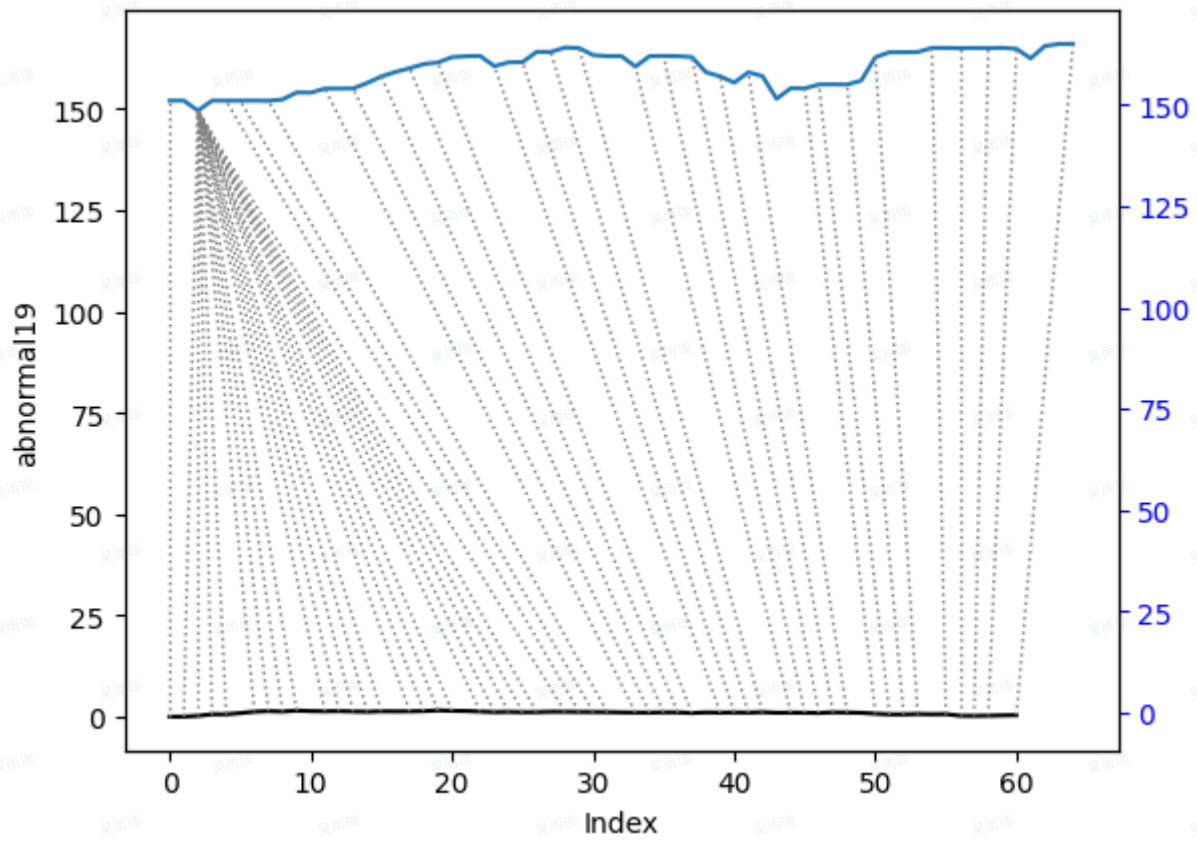

abnormal14
Index
0
5
10
15
20
0
5
10
15
20
0
10
20
30
40
50
60

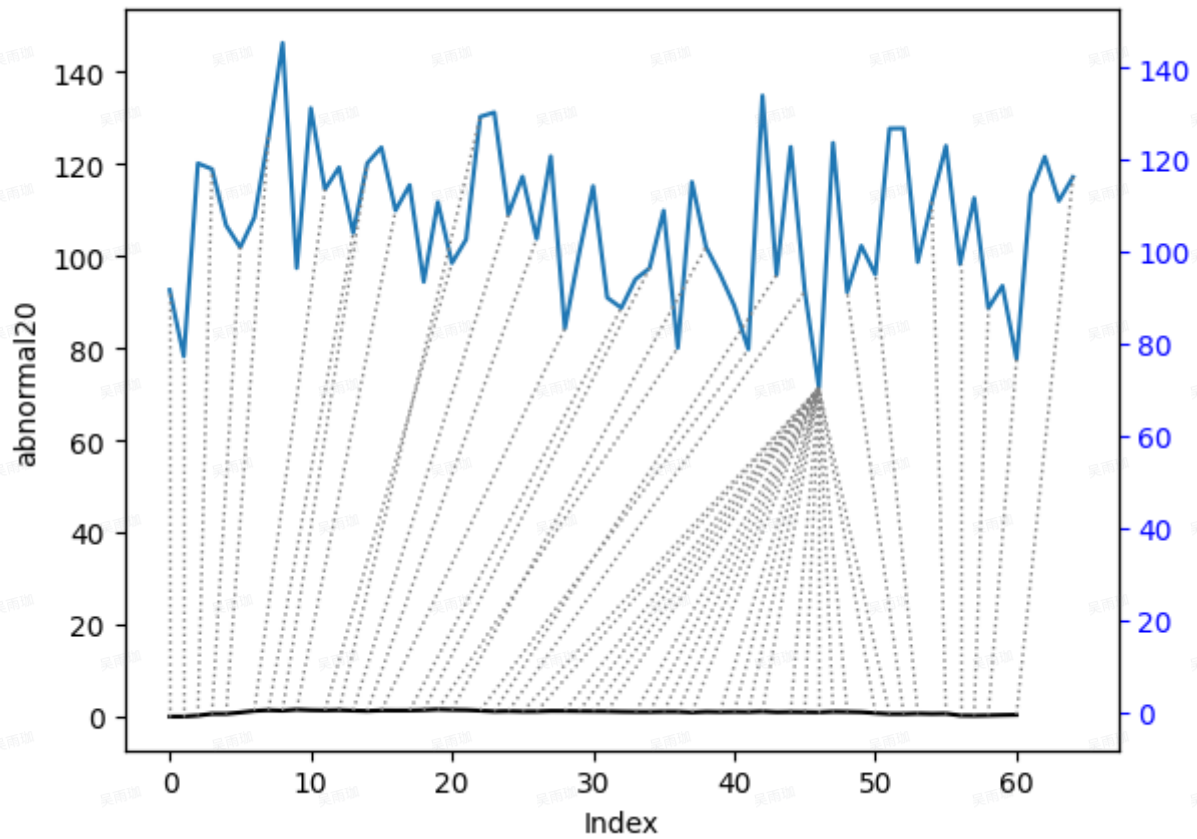

abnormal15
Index
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
10
20
30
40
50
60


abnormal16
Index
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
10
20
30
40
50
60


abnormal17
Index
0
10
20
30
40
50
60
0
20
40
60
80


abnormal18
Index
0
10
20
30
40
50
60
0
5
10
15
20
25
30





四、参考文献

- Pattern distance of time series -D. WangDa & RongGang
  https://www.witpress.com/Secure/elibrary/papers/DATA03/DATA03005FU.pdf

- Searching and Mining Trillions of Time Series Subsequences under Dynamic Time Warping - Thanawin Rakthanmanon, Bilson Campana, Abdullah Mueen, Gustavo Batista2 , Brandon Westover1 , Qiang Zhu, Jesin Zakaria, Eamonn Keogh https://www.cs.ucr.edu/~eamonn/SIGKDD_trillion.pdf
- 时间序列相似性度量综述 https://zhuanlan.zhihu.com/p/69170491
- Python中的dtw-python库 https://dynamictimewarping.github.io/python/