# 目录

# 一、文献调研

## Adtributor

| Data Center | Forecasted Revenue | Actual Revenue | Diff-erence |
|---|---|---|---|
| X | $94 | $47 | $47 |
| Y | $6 | $3 | $3 |
| Total | $100 | $50 | $50 |

Table 2: **Revenue by Data Center**

| Device Type | Forecasted Revenue | Actual Revenue | Diff-erence |
|---|---|---|---|
| A1 | $50 | $24 | $26 |
| A2 | $20 | $21 | -$1 |
| A3 | $20 | $4 | $16 |
| A4 | $10 | $1 | $9 |
| Total | $100 | $50 | $50 |

Table 3: **Revenue by Advertiser**

| Device Type | Forecasted Revenue | Actual Revenue | Diff-erence |
|---|---|---|---|
| PC | $50 | $49 | $1 |
| Mobile | $25 | $1 | $24 |
| Tablet | $25 | $0 | $25 |
| Total | $100 | $50 | $50 |

Table 4: **Revenue by Device Type**

$$Revenue\_Drop(DC == X) = \$47 \quad (1)$$

$$Revenue\_Drop(AD == A1 \vee AD == A3 \vee AD == A4) = \$51 \quad (2)$$

$$Revenue\_Drop(DT == Mobile \vee DT == Tablet) = \$49 \quad (3)$$

| Term | Notation | Example |
|---|---|---|
| Dimensions | $D = \{D_1, D_2, ..., D_n\}$ | {Advertiser,Data center,...} |
| Cardinality of Dimension $D_i$ | $C_i$ | 1000's for advertiser, 10's for Data center,... |
| Elements of Dimension $D_i$ | $E_i = \{E_{i1}, E_{i2}, ..., E_{iC_i}\}$ | {Flower123,...} for Advertisers |
| Measures | $M = \{m_1, m_2, ..., m_k\}$ | {Revenue, Number of Searches,...} |
| Forecasted and Actual Values of measure $m$ for element $E_{ij}$ | $F_{ij}(m), A_{ij}(m)$ | Revenue for Flowers123: forecast = $100, actual = $90 |
| Overall forecasted and actual values of measure $m$ | $F(m), A(m)$ | Total revenue: forecast = $1,000,000 and actual = $900,000 |

从表格可知: 总收入下降值为 50, 一个维度下会有不同的元素值, 例: 数据中心(DC)这个维度下有不同的数据中心 X、Y 作为这个维度下的元素。

**利用**

**① 对异常的解释程度(EP)**：在给定的一个维度下，利用某一个元素的真实值和预测值之间的差距和这个维度下所有元素真实值总和和预测值总和差值的比例大小，例:总的变化量是 50，DC=X 这个数据中心占 47，显然 DC=X 可能和根因有关;

$$EP_{ij} = (A_{ij}(m) - F_{ij})/(A(m) - F(m))$$

**②表述异常的简洁程度(P)**：（1）>（3）>（2）

解释程度超过阈值的元素组合，使用越少的元素去达到超过阈值的解释程度，该组合越"简洁"。

**③ 意外程度(S)：**

EP 偏向于从绝对大小的角度去衡量一个指标是否和异常有关，但是 S 偏向于从相对大小的角度去衡量。例：对于数据中心(DC)这个指标而言，X 是一个大体量的数据中心，而 Y 是一个小体量的数据中心。对 X 而言 94→47，绝对变化是 47，变了 50%，

但是对 Y 而言，虽然 6→3 的绝对变化大小只有 3，也同样变了 50%，因此通过 JS 散度来体现这种“相对异常”。

$$p_{ij}(m) = F_{ij}(m)/F(m), \forall E_{ij} \quad (5)$$

$$q_{ij}(m) = A_{ij}(m)/A(m), \forall E_{ij} \quad (6)$$

$$S_{ij}(m) = 0.5\ (p\ log(\frac{2p}{p+q}) + q\ log(\frac{2q}{p+q})) \quad (7)$$

**这三个指标去定位根因：**

首先计算每个维度下每个元素的“意外程序”，之后对每个维度进行循环遍历，在每个维度中，首先根据意外程度的大小，对每个维度中的元素从大到小进行排序，之后对计算每个元素的“对异常的解释程度”，如果超过阈值，则将这个元素加入这个维度对应的根因“候选”，直到这个维度候选区中所有的候选者“对意外的解释程度”的加和超过阈值，这样，就得到了这个维度下的最终的所有“候选元素”的集合，而且候选区中每个元素的“意外程度”相加可以得到这个维度下整个候选区的“意外程度”。最终，对所有维度的候选元素集合按照“意外程度”再排序，取前三个，就是最终的根因。

**此外，提供了一种根因分析算法应用在派生测量指标中的方法：**

| Adver-tiser | Forecasted Revenue | Actual Revenue | EP % |
|---|---|---|---|
| Overall | 100 | 90 | -10 |
| A1 | 50 | 10 | 400 |
| A2 | 0 | 0 | 0 |
| A3 | 40 | 70 | -300 |
| A4 | 10 | 10 | 0 |

Table 6: **Revenue**

| Adver-tiser | Forecasted Clicks | Actual Clicks | EP % |
|---|---|---|---|
| Overall | 500 | 580 | 16 |
| A1 | 100 | 20 | -100 |
| A2 | 200 | 360 | 200 |
| A3 | 100 | 100 | 0 |
| A4 | 100 | 100 | 0 |

Table 7: **Clicks**

| Adver-tiser | Forecasted Cost/Click | Actual Cost/Click | EP % |
|---|---|---|---|
| Overall | 0.2 | 0.155 | -22.5 |
| A1 | 0.5 | 0.5 | 125 |
| A2 | 0 | 0 | 106 |
| A3 | 0.4 | 0.7 | -131 |
| A4 | 0.1 | 0.1 | 0 |

Table 8: **Cost-per-click**

简单来说就是：一个维度下，如果要算一个派生元素“对异常的解释程度”，在算“预测值”的时候，使用这个元素的“预测值”，其他元素的真实值计算，例如：算 A1 的时候，A2-A4 同理：

60=10+0+40+10,

420=20+200+100+100,60/420=0.143,

0.2-0.143=0.057,

0.057/0.2=28.5%

A1-A4 的四个结果映射变化到比例加和为 100 的状态，就是最终结果（28.5%→125%）

该论文主要是提出了解释力和惊奇性来分析根因，首先假设根因都是由一维变化引起的，计算每一个维度的惊奇性(很大变化的维度比没有表现出这种变化的维度更有可能成为

根本原因)。确定根因维度，然后计算此维度中的各元素解释力，当某几个元素解释力(所有元素解释率之和应该为 1)大于预定阈值时，这些元素被视为根因。同时说明了这种算法的一些缺点：比如一维假设不现实，只适用于基本 KPI，依赖于整体 KPI 的表现。

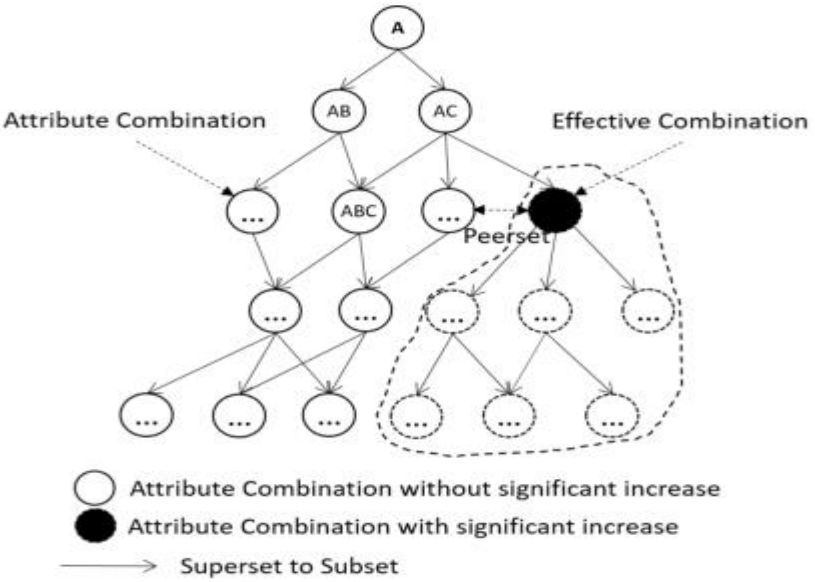
## iDice

**输入"issue"报告，一种时序的多维数据**

Table 1: Sample Issue Reports for a Microsoft Online Service System

| Time | TenantType | ProductFeature | ProductVersion | Package | DataCenter | Country | UserAgent | ClientOS | ... |
|---|---|---|---|---|---|---|---|---|---|
| 5-Dec 11:01 | Home | Admin | V15 RTM | Basic | DC1 | India | IE7.0 | Win8.1 | ... |
| 6-Dec 15:01 | Home | Modify | V15 RTM | Basic | DC1 | USA | IE6.0 | WinXP | ... |
| 7-Dec 10:22 | Edu | Edit | V16 SP1 | Lite | DC3 | UK | Firefox30.0 | Win8.1 | ... |
| 8-Dec 05:33 | Edu | Share | V16 SP2 | Pro | DC6 | India | IE7.0 | WinXP | ... |
| 8-Dec 15:04 | Enterprise | Share | V15 RTM | Lite | DC1 | Australia | IE11.0 | Win8.1 | ... |
| 8-Dec 15:16 | Edu | Admin | V16 SP1 | Ultimate | DC6 | India | IE11.0 | Win8.1 | ... |
| 8-Dec 15:26 | Edu | Edit | V16 SP2 | Pro | DC6 | India | Firefox31.0 | Win7 | ... |

**这种时序的多维数据中有多个维度，每个维度有多个属性，可以把这种多维数据按照属性集合的方式建模成树状结构，**一旦属性集合{AB}会导致异常（例：突增），那么它的下游集合{ABC}、{ABCD}…这些也可能会异常（但这个假设是有缺陷的），所以算法的目标就是在这棵树中找到{AB}这个集合，怎么找？一个标准就是，把 A、B 属性对应到时序数据中，对应的"issue"报告的数量就会有异常。但是其他属性对应的"issue"报告的数量的异常程度会小很多。AB 就是论文中所说的"Effective Combination"




**在搜索过程中，应用“剪枝策略”去减少搜索空间**

**“影响”小的剪掉**，（例：属性 A 对应的 issue 报告数量只涉及很少的用户量）

**变化量小的剪掉**（时序上无较大变化）

**可能导致冗余的剪掉**

其实论文这里的“冗余”主要针对的情景是树中的上下游集合该选哪个作为“effective combination”的问题，需要计算信息熵，一个基本思想就是：两个都有异常的数据集或者两个都没异常的数据集混起来的“熵”小于一个没异常和一个有异常的数据集混起来的“熵”。具体在建模出来的树状结构中，会依据“effective combination”把树分成两个部分，需要算这两个部分数据的“信息熵”，计算得到一个值，叫做“分隔能力”，"isolation power"

针对于上下游的集合：

$$X=\{Country=India\}$$
$$Y=\{Country=India;\ TenantType=Edu;\ DataCenter=DC6\}$$
$$Z=\{Country=India;\ TenantType=Edu;\ DataCenter=DC6;\ Package=Lite\}$$

如果 Y 的分隔能力大于 Z 和 X，那么最终保留的“effective combination”是 Y

最后可能选出来多个“effective combination”，需要根据类似于 Fisher Distance 的衡量标准来排个序

$$R = p_a * \ln \frac{p_a}{p_b} \qquad (2)$$

在一个异常点前后，可以把 issue 数据分为两个部分，前部分记为 a，后部分记为 b

p(a)表示这个“effective combination”中的属性对应的 issue 报告数量在总的报告数量中的占比，p(b)同理

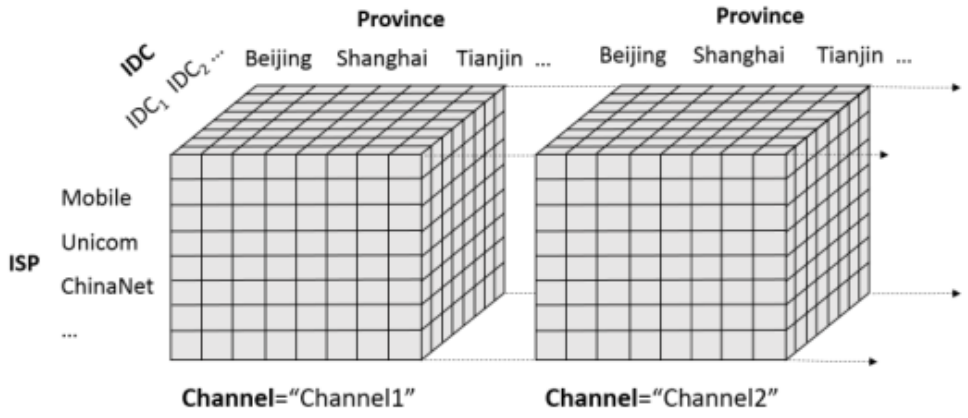
R 值越低，表明这个“effective combination”越不重要，可以剪掉。

算法主要思想是找到有效的属性组合，由于当属性过多时，组合数量呈指数型增长，因此采用剪枝的思想。提出 IP 的概念(基于信息熵，IP 越大表面变化越大)当某一个组合有效时，他的 IP 既大于超集也大于子集(比如组合{A,B}是有效组合，那{A}和{A,B,C}就无效)。此时可将他的超集和子集删去，减小工作量。同时还有一些其他方法来剪枝，比如某个属性变化不大则直接删去；某个组合一段时间内没有变化也删去。

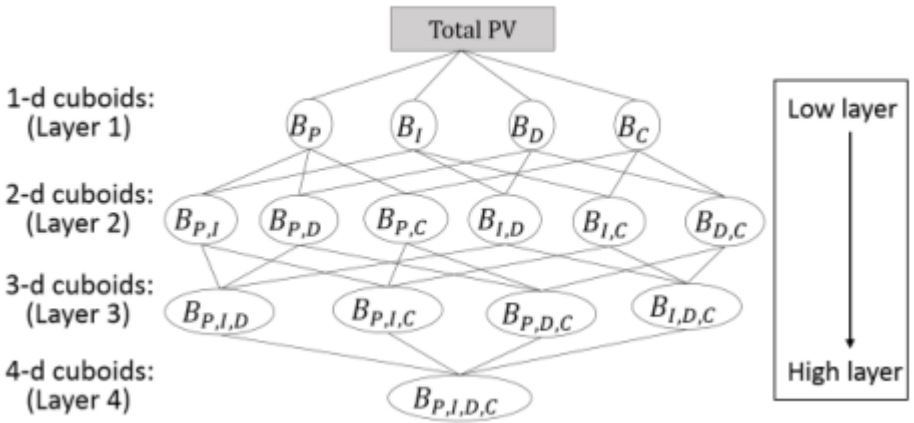
## HotSpot

| Term | Definition | Notation | Example |
|---|---|---|---|
| Attributes | The categories of the information of each PV record | — | Province(P), ISP(I), DC(D), Channel(C) |
| Attribute values | The candidate values of each attribute | — | {Beijing, Shanghai, Guangdong} for Province(P) |
| Element | A combination vector of distinct values of each attribute | $e=(p,i,d,c)$ | (Beijing,*,*,*), (*,Mobile,*,*), (Beijing, Mobile,*,*) |
| PV value | The number of access logs according to an element | $v(e_i)$ | $v$(Beijing,*,*,*) |
| Forecast value | Forecast PV value of an element using the historical values | $f(e_i)$ | $f$(Beijing,*,*,*) |
| Data cube | A data structure of multi-dimensional data | *n-d cube* | A 4-d data cube with the dimensions {P,I,D,C} |
| Cuboids | A cuboid is a data cube whose dimensions are in a subset of all given dimensions | $B_i$ | $\{B_P, B_{P,I}, B_{P,I,D},...\}$ for the 4-d data cube with the dimensions {P,I,D,C} |
| Potential Score | A concept of measuring the potential of a set of elements to be the root cause | $ps$ | ps(S), S={(Beijing,*,*,*), (*,Mobile,*,*)} |

首先，这篇文章的术语表中提到了一个新的术语 Cuboids，




但这只是对不同的维度用相互正交的方式，表示数据的一种表达方式和理解方式，维数可能远大于 3，所以也不一定是个立方体，所以理解为“正交体”更合适，本质上还是给定一个测量指标的实际值，可以投射到不同的维度上，本质上和 Adtributor 以及 iDice 的表达方式并没有很大区别，HotSpot 对数据的建模方式也是一种树状结构。

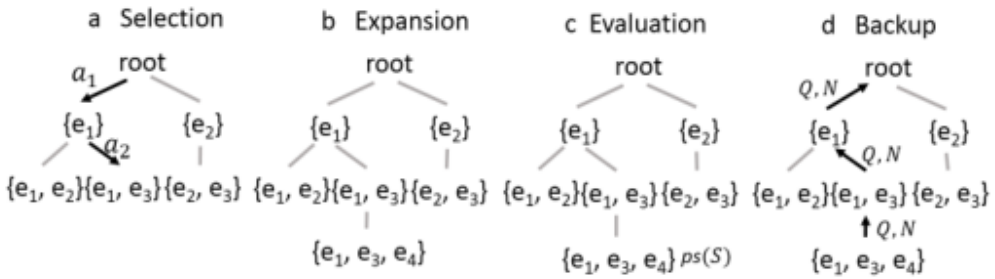



**蒙特卡洛树**

总的来说，HotSpot 把根因定位问题视为一个在巨大的解空间中的搜索问题。事实上，

蒙特卡洛树在 AlphaGo 围棋中的应用证明了这种搜索算法应用在海量空间中进行搜索的能力和效果（蒙特卡洛树就是一种启发式搜索算法）。

但是在用蒙特卡洛树搜索的时候需要根据自己的应用场景去自己定义一个“价值函数”




在蒙特卡洛树中，每一个节点代表一种状态，这个状态其实就是一种属性集合，算法需要干的事情就是不断地从一个节点状态移动到另一个新的节点状态，树中的边可以视为一种状态转移的“动作”。但是算法在执行动作的时候需要依据“价值”来判断接下来执行动作空间中的哪个状态以及把哪个元素加入新的节点中，这个“价值”怎么算就是基于涟漪效应去算一个潜在分数作为“价值”

**涟漪效应(ripple effect)→潜在分数**

| $f(p,i)\rightarrow v(p,i)$ | | Province($p$) | | | |
|---|---|---|---|---|---|
| | | Beijing | Shanghai | Guangdong | * |
| ISP ($i$) | Mobile | 20→8 | 15→15 | 10→10 | 45→33 |
| | Unicom | 10→4 | 25→25 | 20→20 | 55→49 |
| | * | 30→12 | 40→40 | 30→30 | 100→82 |

**8=20-18×(20/30)**

$$v(x_i') = f(x_i') - h(x) \times \frac{f(x_i')}{f(x)},\ (f(x) \neq 0).$$

“涟漪效应”的意图和 Adtributor 中的“对异常的解释程度”这个概念相似

下面式子中，a 代表用涟漪效应算出来的值组成的向量，f 表示预测值组成的向量，v 表示实际值组成的向量

需要注意的是，对于(*, *)这种或者(Beijing, * )这种, a=f 成立

对于(Beijing, Mobile)这种，a=f 并不恒成立

$$Potential\ Score = max(1 - \frac{d(\vec{v}, \vec{a})}{d(\vec{v}, \vec{f})}, 0) \qquad (6)$$

$$d(\vec{u}, \vec{w}) = \sqrt{\sum_i (u_i - w_i)^2}. \qquad (7)$$

**分层剪枝**

从上到下搜索的时候，如果高层的某个节点元素分数很低，那么它的下层分支子树中的节点分数“一般”也不高，所以剪掉。

该算法主要针对两个问题，一是如何找出导致 PV 异常的维度和元素，二是维度过多时组合过多如何搜索(与第二个算法类似)。他将数据切分成不同的 cuboid，在每个 cuboid 找出根因集，最后比较各个 cuboid 的根因集得分，得到最终根因。然后提出了基于 Ripple Effect 的根因判断方法，通过 Potential Score 判断节点与叶子节点满足此条件的程度，通过蒙特卡洛树搜索和分层的剪枝策略来降低复杂度便于搜索。

## Squeeze

**① 普适的涟漪效应(GRE)：**

这篇论文给出了涟漪效应其实在派生测量指标中也适用的简单证明过程，并且通过把

$$\frac{f(e) - v(e)}{f(e)} = \frac{f(S) - v(S)}{f(S)},$$

改写成

$$\frac{f(e) - v(e)}{f(e) + v(e)} = \frac{f(S) - v(S)}{f(S) + v(S)}$$

解决了涟漪效应不能应用在预测值为 0 的情况，虽然这种情况在实际应用中很少发生。

当然，把涟漪效应应用在派生测量指标中后，派生测量指标也可以计算相应的潜在分数(GPS)。

**② 首先自底向上做聚类,通过聚类的方式缩小搜索空间**

**③ 接着自顶向下做搜索**

聚类出来多个可能的属性集合，需要根据潜在分数去选择分数最高的几个属性组合作为最终的根因输出

此外，这篇论文给出了相关算法的优缺比较

TABLE III: Qualitative comparison of related works. Detailed reviews are in Section VII.

| Algorithms | Root Cause Assumption | Measure | Change Magnitude | Parameter Fine Tuning | Time Cost | Method |
|---|---|---|---|---|---|---|
| Adtributor [3] | single attribute (in one first-layer cuboid) | **fundamental & derived (quotient)** | significant | **no** | **very short** | top-down |
| R-Adtributor [4] | **none** | **fundamental & derived (quotient)** | significant | yes | **short** | top-down |
| iDice [5] | one or two attribute combinations | fundamental only | significant | **no** | **very short** | top-down |
| Apriori [6] | **none** | **fundamental & derived** | **any** | yes | always too long | bottom-up |
| HotSpot [7] | all attribute combinations of the root cause in one cuboid | fundamental only | significant | **no** | sometimes long | top-down |
| *Squeeze* | **those which cause the same changes are in one cuboid** | **fundamental & derived (quotient, product)** | **any** | **no** | **short** | bottom-up then top-down |

算法提出了广义涟漪效应（GRE）,捕获了根本原因属性组合与其“后代”属性组合之间的幅度关系,且经过改进可以适用于派生类型和 0 预测值。在 squeeze 算法中先采用自下而上的方法来剪掉正常的叶子节点，减小搜索空间，同时根据 GRE 将潜在的异常属性组合分组到聚类。再自上而下找到引起异常的根本原因。与上述算法不同的是他并不忽略异常量级不显著的属性组合。

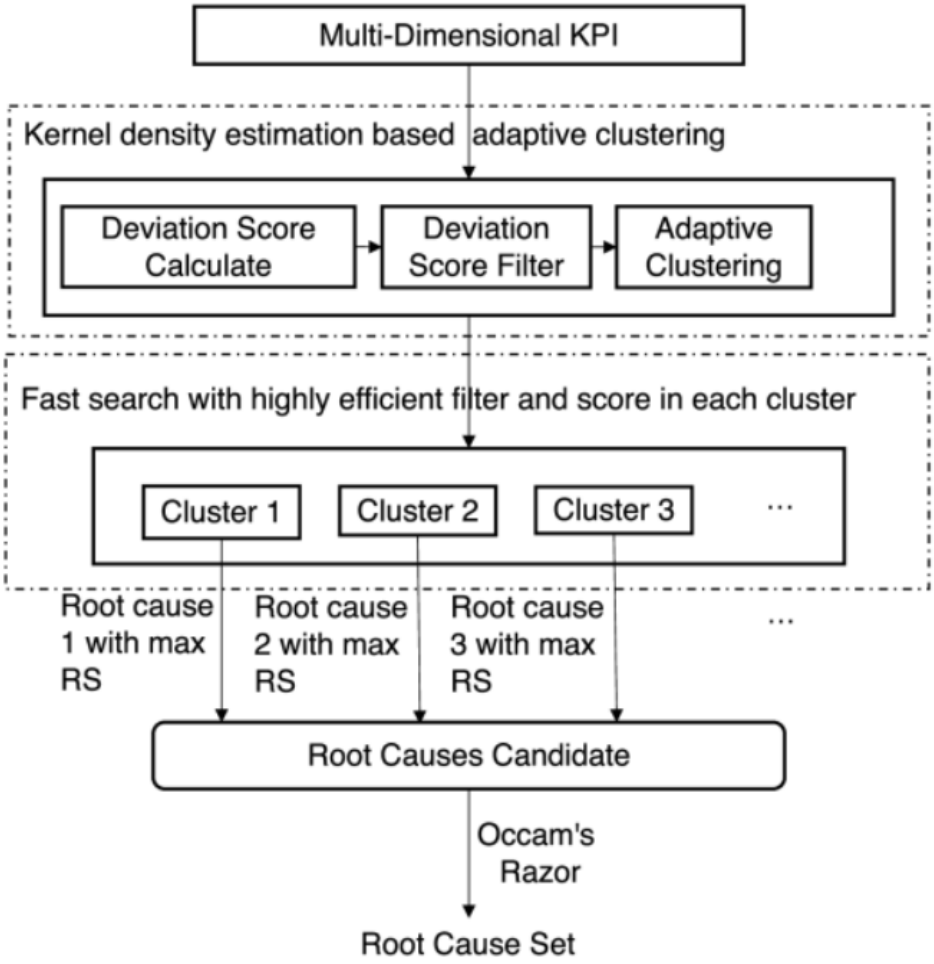
## AutoRoot

根据 GRE 原理，如果存在从 S 继承的根本原因 S 和叶节点 e，则应满足以下公式

$$\frac{f(e)-v(e)}{f(e)}=\frac{f(S)-v(S)}{f(S)}$$

f()和 v()是维度组合的预测值和实际值函数。使用 ARMA 算法来计算预测值 f().在实际情况下，在某些情况下，预测值可能为零。为了避免零分母，以前的工作 [11] 将 f 替换为 f+v/2。
在图 2 的第一个虚线框中，根据公式 3 计算所有叶节点的偏差分数，基于其实际值和预测值。

基于上述 GRE 原理，正常叶节点的偏差得分非常小，而异常叶节点的偏差得分非常大。因此，使用安全偏差分数阈值来过滤掉正常的叶节点。这可以在很大程度上加快根本原因定位的过程，而不会损失准确性（请参阅第 IV-B2 节中的详细信息）。进一步将核密度估计（KDE）与高斯核一起自动得到偏差得分数组的分布密度函数。然后根据分布密度函数将异常叶节点自适应分组为 K 个簇（详见第 IV-B3 节）。在图 2 的第二个虚线框中，搜索最可能的根本原因.max 每个集群中的根分数 （RS） 以获得 K 候选项（请参阅第 IV-D 节中的详细信息）。最后，使用奥卡姆剃刀合并和删除 K 个候选者，以获得一组简化的根本原因。




$$NPS = 1 - \frac{\text{avg}\left(\frac{|v(e_1)-a(e_1)|}{v(e_1)}\right) + \text{avg}\left(\frac{v(e_2)-f(e_2))}{v(e_2)}\right.}{\text{avg}\left(\frac{|v(e_1)-f(e_1)|}{v(e_1)}\right) + \text{avg}\left(\frac{v(e_2)-f(e_2))}{v(e_2)}\right)}$$

NPS 和 GPS 之间最重要的区别是使用所有叶节点的相对差异的平均值。进行此修改是因为差异很容易受到实际值的影响。

AutoRoot 是一种可以快速准确地定位通用多维根本原因的方法。提出了一种无参数密度聚类算法，根据数据本身自适应地选择合适的聚类参数。为了准确评估概率，提出了一种新的指标新潜在分数，采用相对差异。提出了两种新颖的过滤方案，有助于减少搜索空间，并大大加快根本原因定位的过程。通过案例验证此算法确实更有效率。

## PSqueeze

与第四个算法类似，在 GRE 算法的基础上。在自底向上的过程中，将问题分解为更小的问题(单一的根本原因)。这样可以减小搜索空间。然后在自顶向下的过程中，提出了广义潜在评分(GPS)，使用启发式搜索策略搜索最大化属性组合，同时可以判断是否存在外部根因。最后通过向数据集中注入故障证明此算法的有效性。

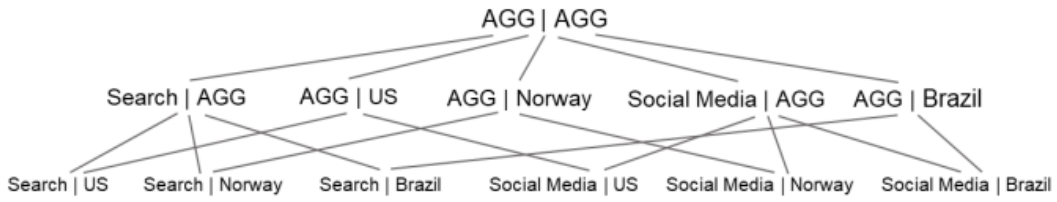
## ImpAPtr

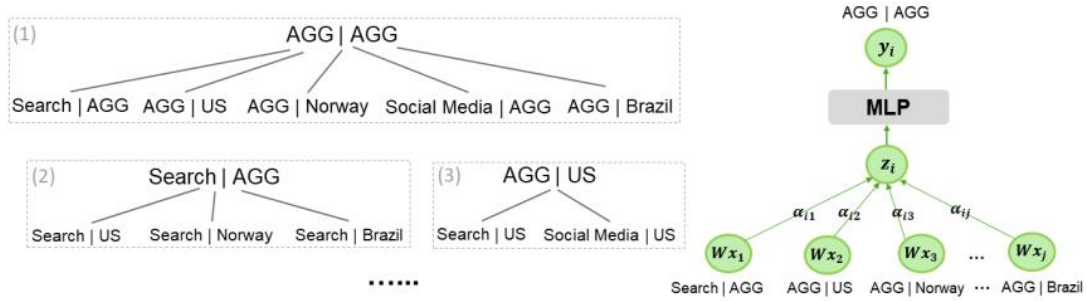
ImpAPtr 采用阈值驱动策略，即需要在服务调用成功率下降≥0.05%时触发，在实际应用中，可能小于此阈值的情况下也会引起极大错误。本文提出的方法去除了阈值的限制，而是加入时间因素，利用跨时间间隔的监控数据。在对三种算法 ImpAPTr+、r-attributor 和 Squeeze 进行了实验评估得出此算法的性能为最优。

## CMMD

### ① 图神经网络的应用

其实“涟漪效应”这个思路的潜台词在于建模不同测量指标之间的“关系”，只不过“涟漪效应”可以用显式的统计学方法去表示出来，CMMD 里直接用 GNN 去建模表示这种关系。




(a) Subtrees of the Dimension Tree　　(b) Graph Attention Network

CMMD 对多维数据的建模表示方式也是一种树状结构。

对于单个的子树而言，可以用单层图注意力机制去建模子节点和父节点之间的关系，这里的多层感知网络(MLP)其实就是用来拟合从子节点到父节点的函数关系（比如，除法计算每次点击的费用）

### ② 遗传算法求最优解

其实遗传算法和 HotSpot 中应用的蒙特卡洛树在应用上是一样的，都需要根据自己的应用场景去自己定义一个“价值函数”，或者在遗传算法中叫“适应度函数”

其实先用图神经网络推理计算得到不同测量指标之间的“关系”之后，把这种关系作为一个黑盒，给一个输入，得到一个输出，可以根据这个输出去构建遗传算法的“适应度函数”。

之后进行编码，把根因候选属性编码成一个向量(数组)，数组中的每一个元素就是一个 0 或者 1，表示这个维度下对应的属性是否被选择成为根因候选选项。

之后就可以用适应度函数去控制遗传算法中交叉变异、增加后代这一整个进化过程，最终拿到最优解

其实可以对比下面这个公式和 HotSpot 中公式(6)在结构上的相似

$$\text{Fitness score} = \frac{|\mathcal{G}^L(\boldsymbol{X}(s)) - f_{\text{root},m}|}{|v_{\text{root},m} - f_{\text{root},m}|} + \beta||s||_1.$$

MID

**① 首先把根因定位过程编码成组合优化问题**

**对于搜索空间：**

对每一个属性都有多个对应的值，建模成（属性—值）元组，

$$t = (a_i, v_i^k)$$

同时每个属性可以有多个值

$$T_i = \{a_i\} \times V_i,$$

$$T = \bigcup_{i=1}^{n} T_i.$$

最终整个搜索空间可以表示成(P(T)表示 T 的超集)：

$$S = \{s \in P(T) \mid \forall T_i : \ \text{card}(s \cap T_i) \leq 1\}$$

**对于搜索的目标函数**

**另外，这篇文章用的目标函数和 iDice 应该是相同的**

$$R(x) = p_{a(x)} \ln \frac{p_{a(x)}}{p_{b(x)}}$$

**② 之后用贪婪模型+随机模型的元启发式搜索算法去解决组合优化问题**
**在搜索的时候定义了 4 种搜索操作：**

增加一个元组

替换一个元组

替换一个元组中的值（属性不变）

删除一个元组

**之后每次迭代计算都是用贪婪模型+随机模型**

贪婪搜索时，这篇论文借鉴禁忌算法去避免重复搜索之前已经搜索过的路径

随机模型的主要作用其实就是增加一些随机扰动，避免贪婪搜索陷入局部最优解，平衡在有的“很有前途”的候选搜索空间中进行搜索和扩展现有的搜索范围这两者。

# 二、数据集溯源

**3.1 讨论 2019AIOps 挑战赛数据集和 Squeeze A 之间的关系。**

**观点一：2019 年挑战赛数据集与 Squeeze A 不同源**

论文《Constructing Large-Scale Real-World Benchmark Datasets for AIOps》中明确提出 2019 年挑战赛数据集(文中 dataset B)与 Squeeze B0-B4 同源。

而在《Generic and Robust Localization of Multi-Dimensional Root Causes》（Squeeze)中得出 A 与 B0-B4 不同源

由此得出 2019 年挑战赛数据集与 Squeeze A 不同源。

**观点二：2019 年挑战赛数据集与 Squeeze A 同源**

从论文来看：
《Constructing Large-Scale Real-World Benchmark Datasets for AIOps》中

说明 2019 年挑战赛数据集来源于苏宁。而 Squeeze 中提到数据集 L1 来自于一家线上购物平台(推断为苏宁)，从下方表格可以看出 L1 正是数据集 A。

从数据集来看:
属性：2019 年挑战赛数据集与 Squeeze A 都具有五个属性且命名相同，都为 i,e,c,p,l 而 Squeeze B 只有四个属性且命名为 a,b,c,d。
时间：Squeeze A 数据集的时间戳处于 2018 年 9 月和 10 月，B、D 数据集的时间戳处于 2015 年 12 月，而 2019AIOps 挑战赛数据集的时间戳处于 2018 年、2019 年

从数据特征来看:

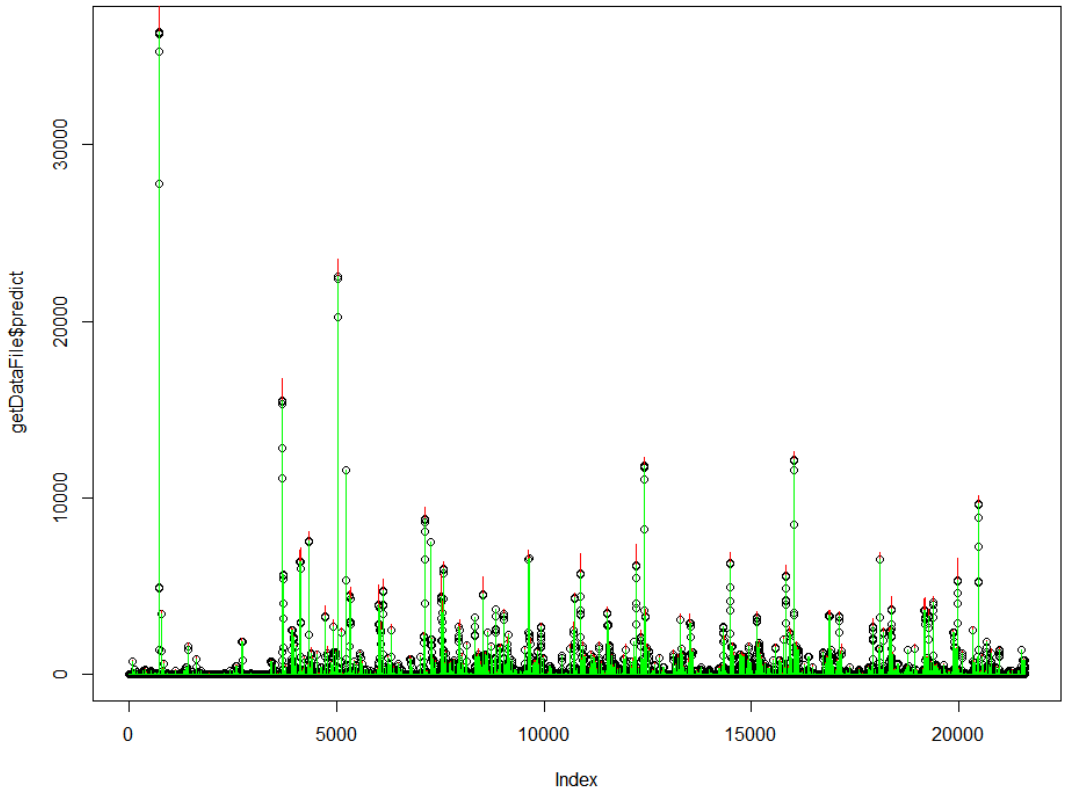
**备注:**

**1.凡是下图中的颜色，绿色表示预测值，红色表示实际值，绿色覆盖红色越多，表明预测值和实际值差距越小**

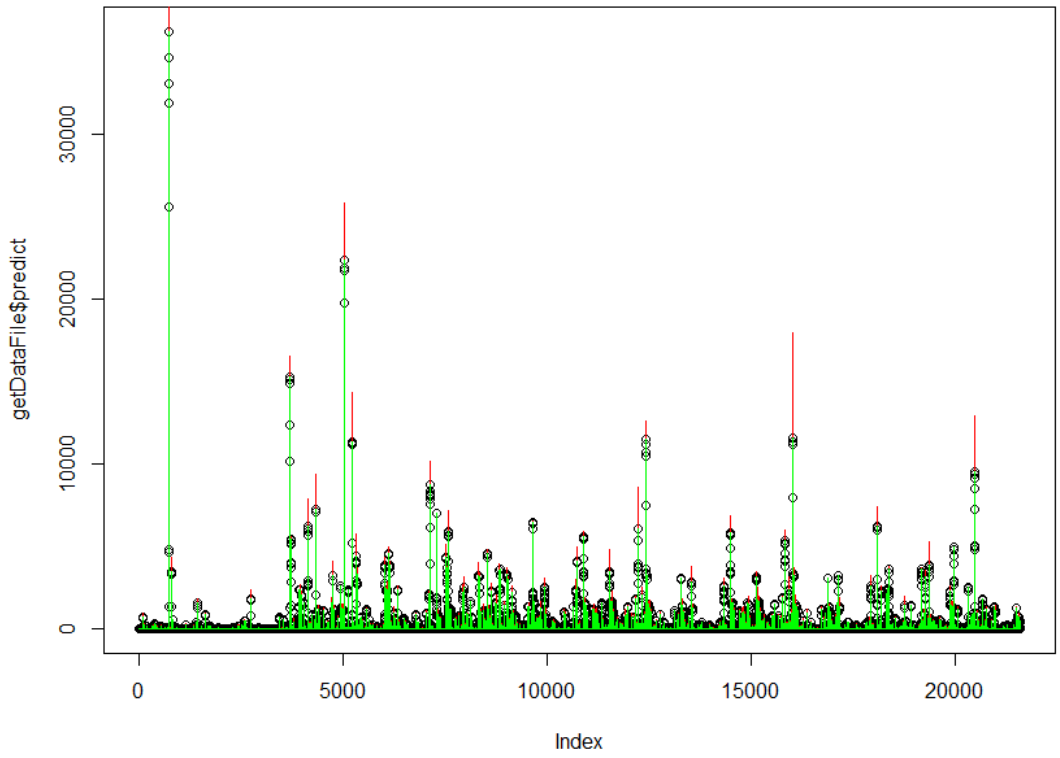
**2.图片含义：选定某个时间戳，横轴表示叶节点属性组合的行数 index，纵轴表示相应属性组合的实际值、预测值**

选定 Squeeze B、D 中的某个时间戳，**典型数据分布**如下:

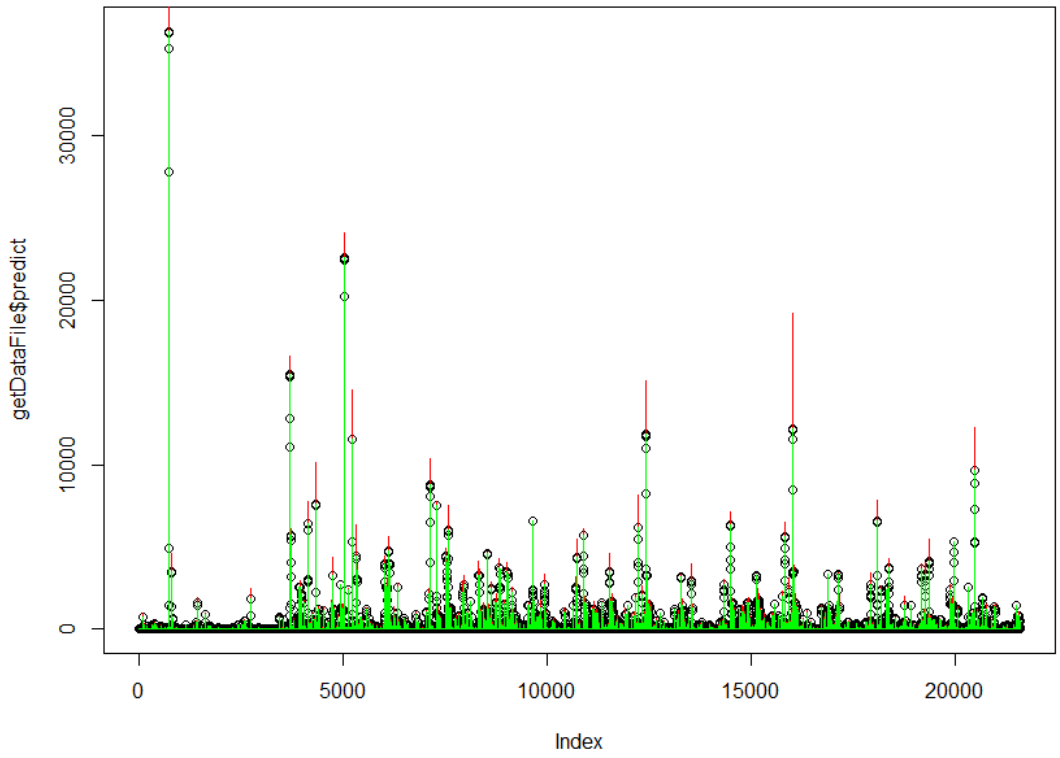
(例：B3/B_cuboid_layer_3_n_ele_3/1450760100.csv)




(例：D/B_cuboid_layer_3_n_ele_3/1450653900.a.csv)

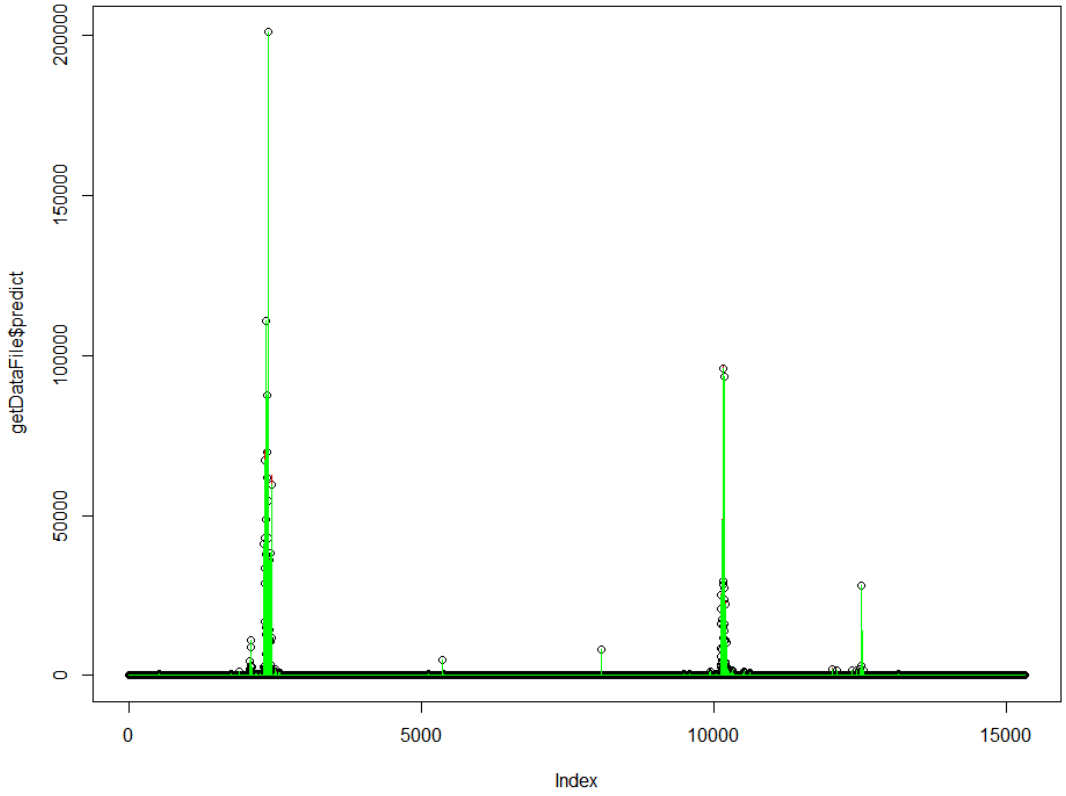



(例：D/B_cuboid_layer_3_n_ele_3/1450653900.b.csv)



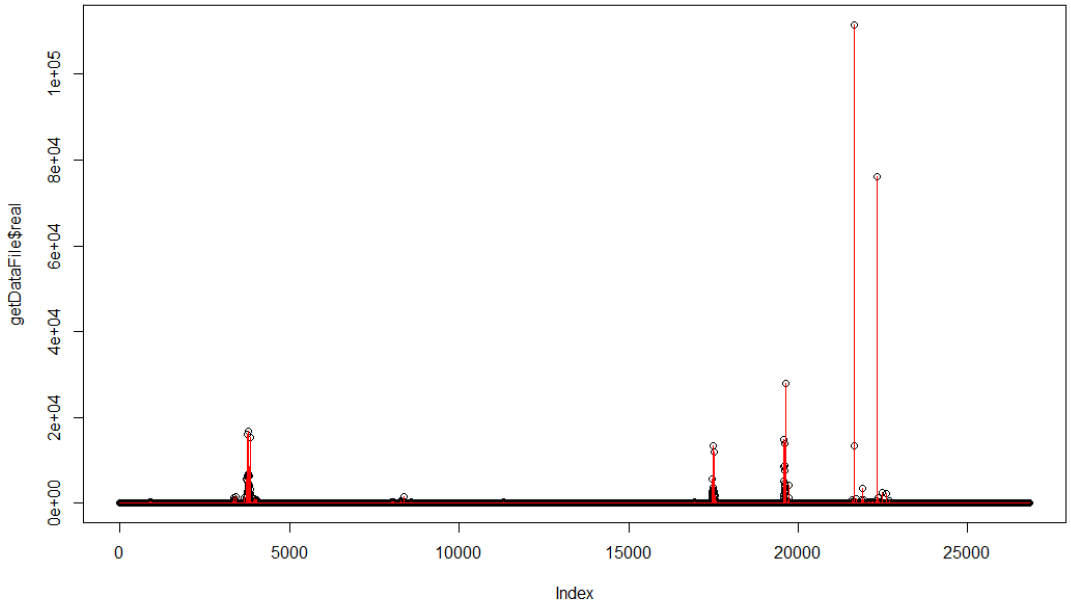

选定 Squeeze A 中的某个时间戳，**典型数据**分布如下：

(例：A/new_dataset_A_week_12_n_elements_3_layers_3/1535837700.csv)




选定 2019AIOps 数据集中的某个时间戳，**典型数据分布**如下：

(例：1544728500000.csv，注意，无预测值）




**从数据分布情况来看，Squeeze A 和 2019AIOps 挑战赛数据集更为相似。**

④ 综上所述

支持观点一：2019AIOps 数据集和 Squeeze B 同源的支撑点

论文《Constructing Large-Scale Real-World Benchmark Datasets for AIOps》的表述

支持观点二：2019AIOps 数据集和 Squeeze A 同源的支持点

1.Squeeze 原论文的表述

2.Squeeze A 和 2019AIOps 数据集时间戳相同

3.Squeeze A 和 2019AIOps 数据集属性标签相同

4.Squeeze A 和 2019AIOps 数据集数据特征相同

总结：论文《Constructing Large-Scale Real-World Benchmark Datasets for AIOps》中的表述有误，2019AIOps 数据集和 Squeeze A 应当同源，而非 Squeeze B。

# 三、算法复现结果

Squeeze

**运行结果：**本地运行结果和论文数据相符合，虽然具体数据并不完全相同，但**总体情况一致**：

除了(n_element,cuboid_layer)=(3, 3)这项数据和原论文相比低 0.1025，差距超过 0.1 之外，其他数据项的偏差均在 0.07 以内，(n_element,cuboid_layer)=(1, 1)这项数据甚至比原论文数据高出 0.04，且 F1 值超过 0.9

在 Squeeze A 数据集上的具体运行结果如下：

| | (1, 1) | (1, 2) | (1, 3) | (2, 1) | (2, 2) | (2, 3) | (3, 1) | (3, 2) | (3, 3) |
|---|---|---|---|---|---|---|---|---|---|
| 论文数据 | 0.8632 | 0.7827 | 0.4932 | 0.7584 | 0.6361 | 0.4097 | 0.6441 | 0.5145 | 0.3618 |
| 本地数据 | 0.9048 | 0.7772 | 0.4350 | 0.7822 | 0.5899 | 0.3436 | 0.6825 | 0.4933 | 0.2593 |
| 论文→本地 | +0.0416 | -0.0055 | -0.0582 | +0.0238 | -0.0462 | -0.0661 | +0.0384 | -0.0212 | -0.1025 |

此外，Squeeze 计算 tp 数量时采用了**严格的标准**，如下：

```
timestamp:1451220600
label:a=a4&b=b15&c=c3;a=a2&b=b24&d=d8;a=a3&b=b31&c=c1
pred :a=a4&b=b15&c=c3;a=a2&b=b24&d=d8;a=a3&b=b31&c=c1&d=d5
tp: 2, fp: 1, fn: 1
```

只有当真值和预测值的叶节点属性组合完全一致时才会标记为 tp，

当真值 label=(a=a3&b=b31&c=c1)，预测值 pred=(a=a3&b=b31&c=c1&d=d5)

虽然 pred 已经包含了 label，但 Squeeze 并不将其计为 tp，当且仅当 lebel 和 pred 完全一致时，才会记为 tp

## 四、关于 Squeeze 在 A 数据集(3, 3)条件效果不佳的原因分析

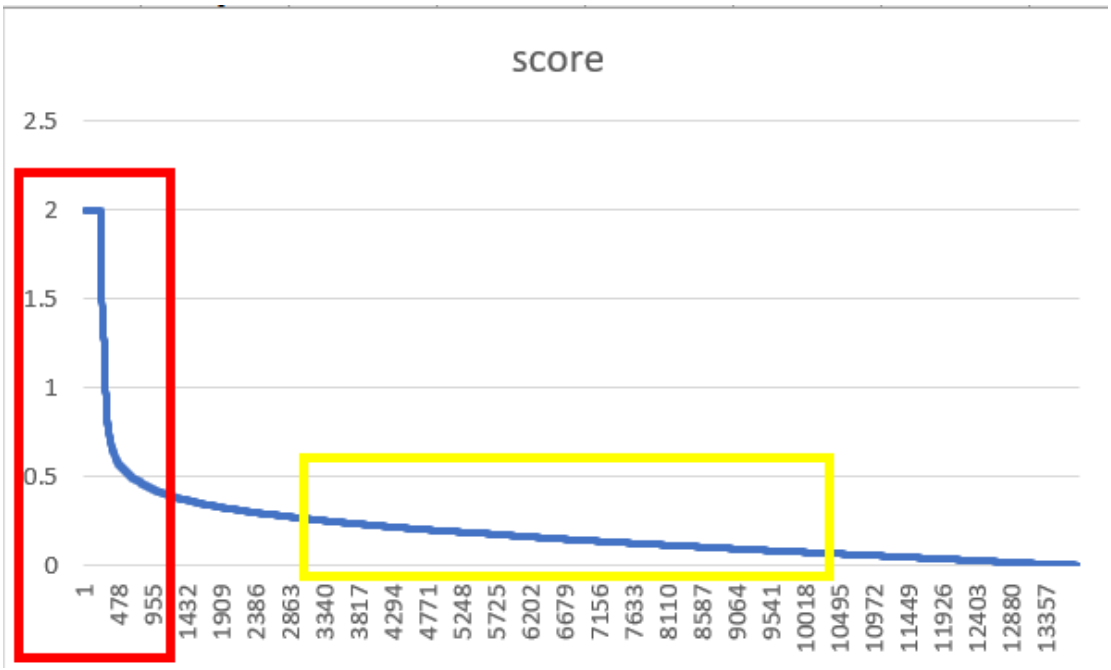
① 涟漪效应太过理想化，存在缺陷：

结合常识直观感知，涟漪在传播过程中必然是会遇到阻力，慢慢消散；很不幸，起码从 Squeeze A、B、D 数据来看，在根因分析中，涟漪的衰减速度几乎是指数级别的速度，这一点从 Squeeze 原论文中自己提出来的 Deviation Score 的衰减速度就能看出来，“异常”并没有按照理想情况进行持续传播；

(备注，涟漪的指数衰减其实也解释了为什么上次开会讨论的时候 A 数据集中画出来的曲线会出现很突兀的两根竖线，看似不符合涟漪效应，但是论文中却明确说明在错误注入的时候是严格按照 GRE 来的)

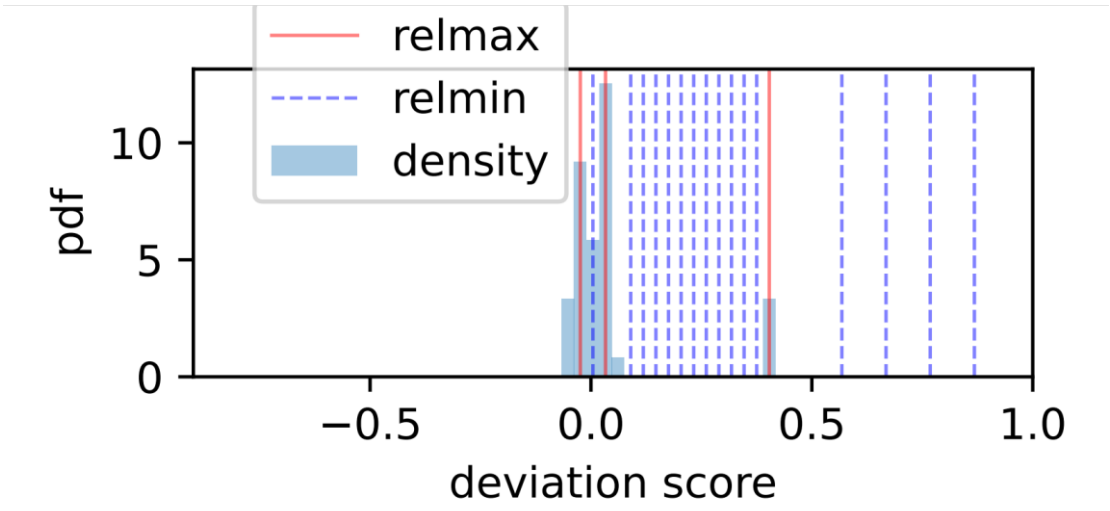
② Squeeze 论文中基于涟漪效应的假设做基于密度的聚类，

真正有用的信息在红色区域，但是 Squeeze 做基于密度的聚类会导致搜索空间定位在黄色区域，南辕北辙。涟漪指数衰减的过程中对应的红色区域往往只有小比例的叶节点熟悉集合数量，大量的叶节点属性集合拥挤在指数衰减之后，也就是黄色区域，当然密度会很大。

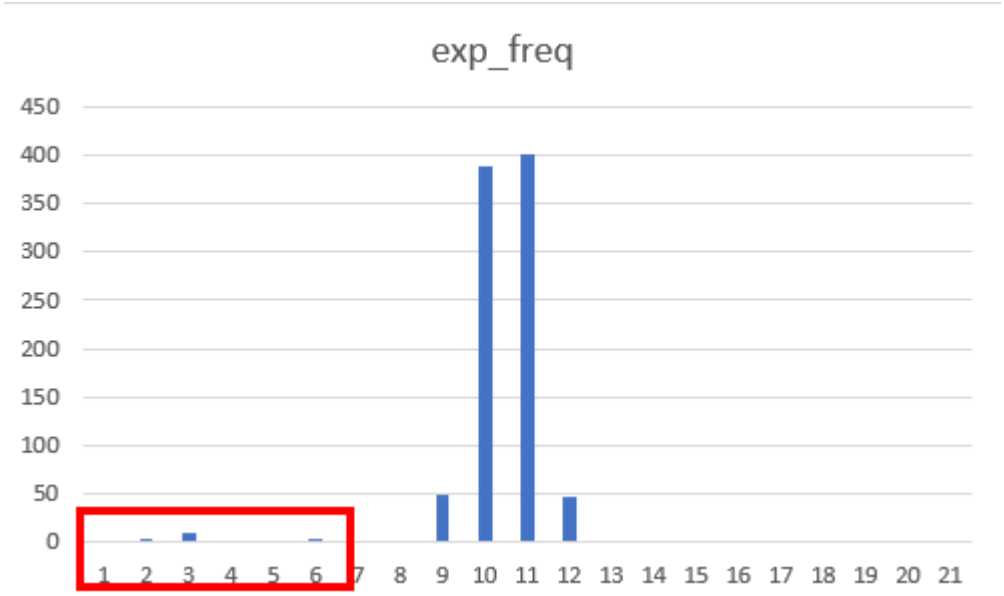



**下面是 Squeeze 的聚类可视化和 GT 的对比，显然聚类方向有误**

算法聚类位置:




GT 位置(红色框区域)




后续

已经商用的 CMMD 并不开源，但可以借鉴 DejaVu 项目中的 MLP、图注意力机制等诸多相似之处，曲线救国，尝试复现 CMMD。

## 五、算法改进思路调研

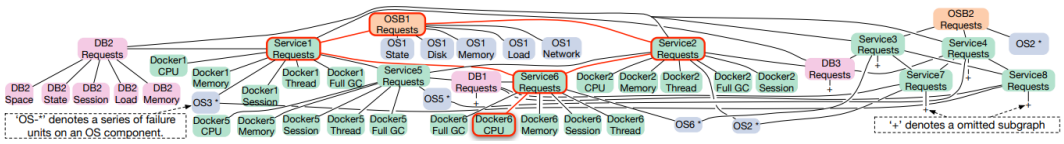
**5.1DejaVu**
**输入：**

① faults 数据，包含[timestamp, root_case_node]，例：

| | A | G |
|---|---|---|
| 1 | timestamp | root_cause_node |
| 2 | 1586534700 | docker_003 CPU |
| 3 | 1586536500 | docker_002 |
| 4 | 1586538600 | docker_001 |

（备注：虽然数据集中除了 timestamp 和 root_case_node 之外还有其他数据项，但不入模）

② graph 数据，表示(故障)组件节点之间的依赖关系，可视化例：



(备注：上图中的每个节点代表一个 **failure unit**，包含的信息为故障位置+故障类型)

③ metrics 数据, graph 中的每个 node 都会对应多个 metric, 每个 metric 的格式为[name, timestamp, value]:

| | name | timestamp | value | metric_type |
|---|---|---|---|---|
| 0 | db_003##ACS | 1590685860 | -0.333333 | ACS |
| 1 | db_003##ACS | 1590685920 | 0.000000 | ACS |
| 2 | db_003##ACS | 1590685980 | 0.000000 | ACS |
| 3 | db_003##ACS | 1590686040 | 0.000000 | ACS |
| 4 | db_003##ACS | 1590686100 | 0.000000 | ACS |
| ... | ... | ... | ... | ... |
| 1380339 | db_009##tnsping_result_time | 1586555640 | 0.000000 | tnsping_result_time |
| 1380340 | db_009##tnsping_result_time | 1586555700 | 0.000000 | tnsping_result_time |
| 1380341 | db_009##tnsping_result_time | 1586555760 | 1.000000 | tnsping_result_time |
| 1380342 | db_009##tnsping_result_time | 1586555880 | 0.500000 | tnsping_result_time |
| 1380343 | db_009##tnsping_result_time | 1586555940 | 0.000000 | tnsping_result_time |

[1380344 rows x 4 columns]

## 输出：

左侧：真值

右侧：预测所得的置信度排名前三的可能根因

在程序中，若 rank-1 的根因与真值一致，则预测成功，否则判定为预测错误

| timestamp | root cause | rank-1 | rank-2 | rank-3 |
|---|---|---|---|---|
| 2020-05-30T04:13:00+08:00 | docker_002 CPU | docker_002 CPU | db_003 Session | db_009 Session |
| 2020-05-29T03:41:00+08:00 | docker_001 CPU | docker_001 CPU | docker_008 | docker_007 |
| 2020-05-23T00:05:00+08:00 | docker_004 CPU | docker_004 CPU | db_009 Session | os_021 Network |
| 2020-05-27T05:09:00+08:00 | docker_001 CPU | docker_001 CPU | os_020 Network | db_003 Session |
| 2020-05-27T01:23:00+08:00 | docker_006 CPU | docker_006 CPU | docker_006 | db_009 Session |
| 2020-05-26T05:15:00+08:00 | docker_002 CPU | docker_002 CPU | os_017 Network | db_003 Session |
| 2020-04-11T04:40:00+08:00 | docker_008 CPU | docker_008 CPU | docker_008 | db_003 Session |
| 2020-05-28T00:47:00+08:00 | docker_001 CPU | docker_001 CPU | os_021 Network | os_018 Network |
| 2020-05-23T05:20:00+08:00 | docker_005 | docker_005 | os_021 Network | os_019 Network |

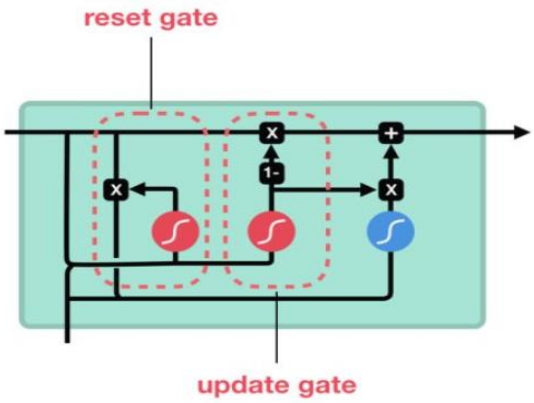
## 网络结构

总过程：抽特征+做分类，具体过程如下：

① 构建错误单元矩阵(Metrics of a failure unit):
w 方向：表示时间维度，每个 failure unit 节点对应的时序数据；

m 方向：表示空间维度，每个 failure unit 节点都对应着多个测量指标；

② 利用 GRU 抽取时间维度上的特征，得到 feature_map_1，其中，GRU 类似于 LSTM 网络，也存在门机制



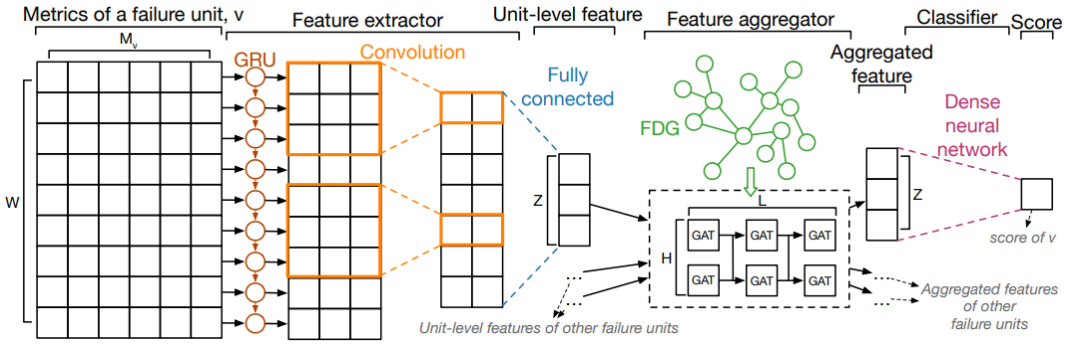

③ 利用 CNN 等方式进一步抽取空间维度上的特征，得到 feature_map_2

④ 经过全连接层将 feature_map_2 变换到一个 Unit-level 特征向量

⑤ 把从多个不同 failure unit 提取的 Unit-level 特征向量和 FDG 作为输入一起输入 GAT 网络
(备注：GAT，Graph Attention Network，图注意力网络，在 GCN 的基础上添加了一个隐藏的 self-attention 层，在卷积过程中将不同的重要性分配给邻域内的不同节点，**同时处理**不同大小的邻域。)

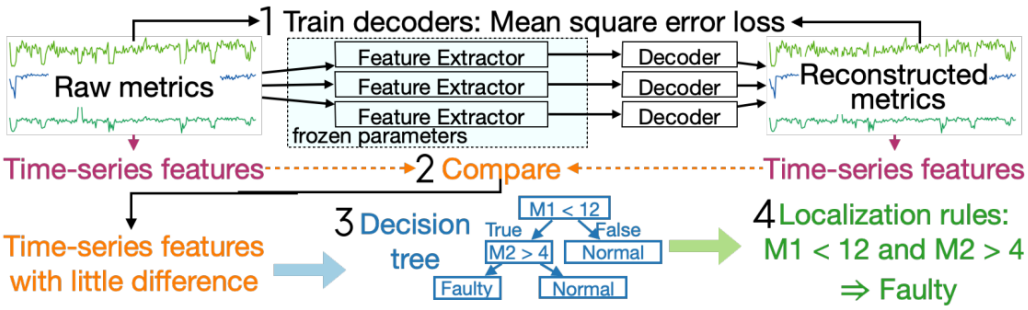
⑥ 每个 failure unit 都能得到一个对应的，经过 GAT 处理得到的聚合特征，经过分类网络之后，每个 failure unit 都能得到一个对应的分数

⑦ 把得分最高的三个 failure unit 进行输出




## 关于模型的可解释性

在实际工作中，如果只是把 DejaVu 作为一个黑盒供运维人员使用的话，难以令人理解和信服，因此需要额外提供一些增强模型可解释性的措施：




将上述已经训练好的用于“抽特征”的网络结构视为一个编码器，后接一个解码器，输入原始数据，输出重建之后的数据。对比输入数据和输出数据，基于它们的差别构建一颗决策树，模拟模型的决策过程，供运维人员理解。

但是，需要额外注意的是：

① 决策树仅仅只是对模型分类过程的一个模拟，并不是模型决策过程本身，论文中也明确提到，决策树的准确度是低于模型的，例如：模型的输出结果是 A、B、C 三个根因节点，但是决策树可能输出的是 C、D、E 这三个节点；

② 虽然论文没有明确提“自编码器”这个术语，但是图中的结构本质上就是一个自编码器，极端理想条件下，自编码器的输入应当完全等同于输出，原始数据和重建数据没有区别，这个时候决策树是失效的。不过在客观实际条件下，解码器输出的重建数据是基于隐空间特征信息的重建数据，也就是说，相比于原始数据，重建数据是滤去噪声的数据。从这一点来讲，论文中所提出的这种辅助解释模型分类路径的方式也是有一定道理的。

## 5.2 Riskloc

## 介绍：

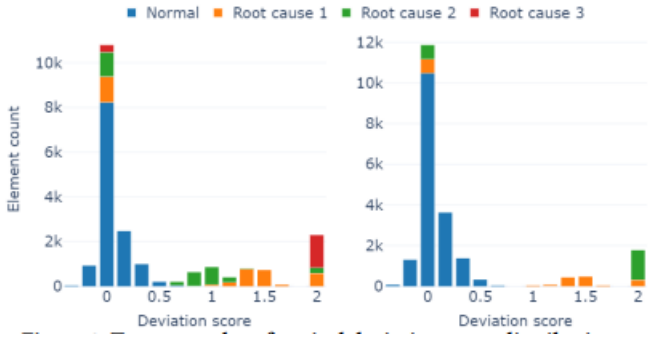
RiskLoc 应用了一种 2-way 划分方案，并分配了与与划分点的距离线性增加的元素权重。 一个风险分数被分配给整合了两个因素的每个元素，1）它在异常分区内的加权比例 ， 2 )偏差分数的相对变化经过涟漪效应属性的调整。

## 算法：

计算每个元素的偏差分数

$$ds(e) := 2 \cdot \frac{f(e) - v(e)}{f(e) + v(e)},$$

偏差分数较小的叶元素可以被认为是正常的 ， 因为它们的相对预测残差较小，而偏差分数进一步偏离零可能是异常的。




**计算加权后的叶子节点集合**

**Input:** leaf element set $\mathcal{E}$

$\mathcal{D} = \{ds(e) \mid e \in \mathcal{E}\}$
Remove outliers in $\mathcal{D}$
**if** $|\min \mathcal{D}| < |\max \mathcal{D}|$ **then**
  $t = -\min \mathcal{D}$
  $E_n = \{(e, |t - ds(e)|, 0) \mid e \in \mathcal{E}, ds(e) < t,$
    $f(e) \neq 0 \vee v(e) \neq 0\}$
  $E_a = \{(e, |ds(e)|, 1) \mid e \in \mathcal{E}, ds(e) \geq t\}$
**else**
  $t = -\max \mathcal{D}$
  $E_n = \{(e, |t - ds(e)|, 0) \mid e \in \mathcal{E}, ds(e) > t,$
    $f(e) \neq 0 \vee v(e) \neq 0\}$
  $E_a = \{(e, |ds(e)|, 1) \mid e \in \mathcal{E}, ds(e) \leq t\}$
**end if**
$E_z = \{(e, 0, 0) \mid e \in \mathcal{E}, f(e) = 0, v(e) = 0\}$
$\mathcal{E}_w = E_n \cup E_a \cup E_z$
$\mathcal{E}_w = \{(e, \min\{w, 1\}, p) \mid (e, w, p) \in \mathcal{E}_w\}$

**Output:** Weighed leaf element set $\mathcal{E}_w$

根据 D 的最小值和最大值，将叶子节点集合 E 分成三个子集 En、Ea 和 Ez，分别代表 ds(e)小于、大于和等于阈值 t 的叶子节点。

对于每个叶子节点 e，计算它的权重 w(e)。如果 f(e)不等于 0 或 v(e)不等于 0，则 w(e)为|t-ds(e)|。否则，w(e)为 0。最后标准化到 0-1 之间

**元素搜索**

输入是加权后的叶子节点集合 E'，风险阈值 tr 和解释能力阈值 tep

**Algorithm 2** Element Search (ES)

**Input:** weighed leaf element set $\mathcal{E}_w$, risk threshold $t_r$, explanatory power threshold $t_{ep}$

**for** $l = 1$ **to** $d$ **do**
  $candidates = \{\}$
  **for** $C \in C$ in layer $l$ **do**
    prune elements in $ES(C)$
    **for** $e_r \in ES(C)$ **do**
      $risk$ = apply equation 12 on $e_r$
      **if** $risk \geq t_r$ **and** $ep(e_r) \geq t_{ep}$ **then**
        $candidates = candidates \cup \{e_r\}$
      **end if**
    **end for**
  **end for**
  **if** $candidates$ is not empty **then**
    **return** $e_r \in candidates$ with maximum $ep(e_r)$
  **end if**
**end for**
**return** null

对于 C 中的每个叶子节点 er，计算它的风险 risk 和解释能力 ep

$$ep(e) := \frac{v(e) - f(e)}{v(M) - f(M)}.$$

$$w_a = \sum_{(w,1)\in S} w, \qquad w_n = \sum_{(w,0)\in S} w.$$

$$r_n = \sum_{e\in LD(e_r)} 2\left|\frac{a(e) - v(e)}{a(e) + v(e)}\right|, \qquad r_d = \sum_{e\in LD(e_r)} |ds(e)|.$$

$$r_1 = \frac{w_a}{w_n + w_a + 1}.$$

$$r_2 = \frac{r_n}{r_d}.$$

$$risk = r_1 - r_2.$$

如果 er 的 risk 值大于等于风险阈值 tr 并且 ep 值大于等于解释能力阈值 tep，则将 er 加入候选解释叶子节点集合 candidates。如果候选解释叶子节点集合 candidates 非空，则返回其中 ep 值最大的叶子节点；否则返回空值 null。

**输出根因集合**

**Input:** leaf element set $\mathcal{E}$, risk threshold $t_r$, proportional explanatory power threshold $t_{pep}$

Obtain $\mathcal{E}_w$ using Algorithm 1
$R = \{e \mid (e, w, p) \in \mathcal{E}_w, p = 1\}$
**if** $ep(R) < 0$ **then**
  $ep(\mathcal{E}_w) = -ep(\mathcal{E}_w)$ ▷ Negate $ep$ for all elements
**end if**
$t_{ep} = t_{pep} \cdot ep(R)$
$RS = \{\}$
**while** $ep(R) \geq t_{ep}$ **do**
  $e_r = \mathrm{ES}(\mathcal{E}_w, t_r, t_{ep})$
  **if** $e_r$ is null **then**
    **break**
  **end if**
  $\mathcal{E}_w = \{(e, w, p) \mid (e, w, p) \in \mathcal{E}_w, e \notin LD(e_r)\}$
  $R = \{e \mid (e, w, p) \in \mathcal{E}_w, p = 1\}$
  $RS = RS \cup \{e_r\}$
**end while**

**Output:** Set of root causes $RS$

使用 Algorithm 1 将叶子节点集合 E 转换为加权后的叶子节点集合 Ew。

从 Ew 中筛选出所有原因节点，即 p=1 的叶子节点，构成集合 R。

如果 R 的解释能力 ep(R)小于 0，则将所有叶子节点的解释能力 ep 取负数。

根据比例解释能力阈值 tpep 和 R 的解释能力 ep(R)，计算出最终的解释能力阈值 tep。

初始化一个空集合 RS，用于存储被筛选出的根因节点。

当 R 的解释能力 ep(R)大于等于 tep 时，重复以下步骤： a. 使用 ES(Ew, tr, tep)算法获取符合条件的最优解释能力叶子节点 er。 b. 如果 er 为空，则跳出循环。 c. 从 Ew 中删除 er 的所有祖先节点。 d. 更新 R 为 Ew 中所有权重为 1 的叶子节点。 e. 将 er 加入 RS 中。

返回 RS 作为根因集合 R。

**结果比较**

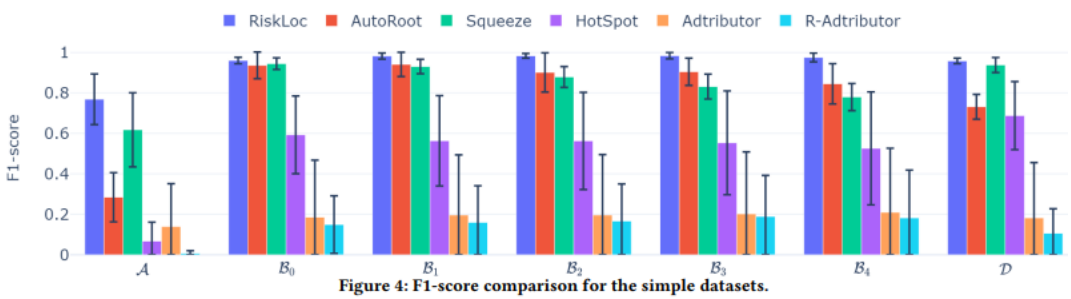



**Figure 4: F1-score comparison for the simple datasets.**

Table 3: F1-scores on the more complex datasets.

| Dataset | Algorithm | | | | | |
|---|---|---|---|---|---|---|
| | Adtributor | R-Adtributor | HotSpot | Squeeze | AutoRoot | RiskLoc |
| $\mathcal{A}_\star$ | 0.0650 ± 0.12 | 0.00003 ± 0.00 | 0.0127 ± 0.02 | 0.2951 ± 0.16 | 0.1802 ± 0.06 | 0.4640 ± 0.10 |
| $\mathcal{S}$ | 0.0589 | 0.0066 | 0.1740 | 0.1283 | 0.4478 | 0.6350 |
| $\mathcal{L}$ | 0.0000 | 0.0089 | 0.1848 | 0.3524 | 0.5266 | 0.6767 |
| $\mathcal{H}$ | 0.0235 | 0.0000 | 0.1109 | 0.0511 | 0.3492 | 0.4906 |

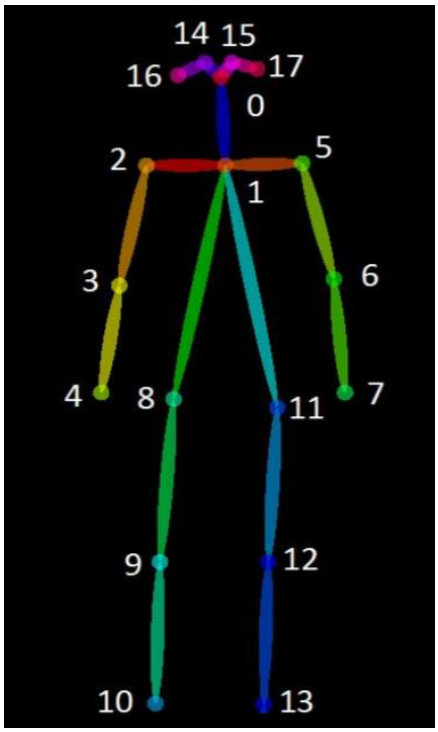
在 riskloc 算法中，F1 得分比前面提到的算法有改善。

| Dataset | Folder | TP | FP | FN | Time | F1-score |
|---|---|---|---|---|---|---|
| A | layer_1_el | 1392 | 0 | 0 | 231.19262 | 1 |
| A | layer_1_el | 2691 | 238 | 1027 | 446.14174 | 0.8096886 |
| A | layer_1_el | 3120 | 532 | 2348 | 538.10031 | 0.6842105 |
| A | layer_1_el | 3198 | 923 | 4014 | 679.76234 | 0.5643695 |
| A | layer_1_el | 508 | 199 | 907 | 113.35341 | 0.4787936 |
| A | layer_2_el | 1774 | 281 | 101 | 1907.9078 | 0.902799 |
| A | layer_2_el | 2812 | 678 | 935 | 2047.8437 | 0.7771176 |
| A | layer_2_el | 3331 | 1009 | 2271 | 2293.773 | 0.6700865 |
| A | layer_2_el | 3336 | 1192 | 3821 | 2300.9937 | 0.5709884 |
| A | layer_2_el | 669 | 334 | 1310 | 482.31628 | 0.4486922 |
| A | layer_3_el | 1526 | 439 | 349 | 7638.8207 | 0.7947917 |
| A | layer_3_el | 2451 | 1096 | 1300 | 5470.4839 | 0.6716909 |
| A | layer_3_el | 2862 | 1492 | 2190 | 5420.4208 | 0.6085477 |
| A | layer_3_el | 895 | 609 | 1057 | 1349.4958 | 0.5179398 |
| A | layer_3_el | 203 | 151 | 297 | 313.86786 | 0.4754098 |
| A | layer_4_el | 862 | 586 | 1013 | 16528.597 | 0.5188083 |
| A | layer_4_el | 1031 | 1416 | 2461 | 14021.656 | 0.3471965 |
| A | layer_4_el | 564 | 1186 | 1971 | 6106.9587 | 0.2632439 |

# 六、图注意力网络

## 原理总述

**“图”中存在着两种特征：边特征和节点本身的特征。**下面举一个计算机视觉中抽取人体骨骼关键点图结构特征的直观例子，通过类比的方式来对根因分析中的“图”建立一个基本认知：




实际应用中，显然经典的 CNN 卷积常常被用来通过 3×3 大小的卷积核来处理规整的方形图像这样的欧几里得结构，但并不能直接对人体骨骼关键点这样的非欧几里得图像结构进行卷积。因此，对于人体骨骼关键点图像，做如下变换：

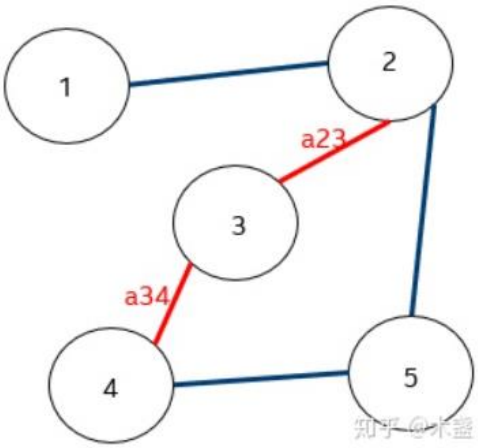
1.利用对称矩阵 A 表示人体骨骼关键点构成的无向图，例：人体左侧大腿部位对应于对称矩阵中 A[11,12]=A[12,11]=1，人体手和脚之间不直接存在骨骼连接，因此 A[13,7]=A[7,13]=0，对于上图所示的 18 节点无向图，可以构成 18×18 的对称矩阵，**体现“图”中“边”的特征**；

2.假设输入一段包含 T 帧的视频，视频中存在 M 个人的骨骼关键点，每个人体骨骼关键点的格式和上图中的 18 骨骼关键点一致，那么对于 batch_size 为 N 的情况，可以得到如下尺寸的张量[N×M，C，T，V]，其中 C=(x, y, confidence)

代表在 T 帧图像中，1 个人体骨骼关键点在图像中的坐标值和置信度。类比 H×W 大小的 RGB 图像进入 CNN 做卷积时，[N，3，H，W]这样的经典张量尺寸，可以看出，**C 就能够体现“图”中每个“点”的特征，对应一个三元组**。

**对应 DejaVu 的根因分析场景，**FDG 便是需要处理的“图”。其中，“边”的关系由组件之间的调用依赖关系体现，因此 FDG 中每个节点，也就是 Failure Unit 节点需要一个对应的“节点特征”，这也就是 DejaVu 中通过 GRU 等“Feature Extractor”过程得到多个“Unit-level Feature”并对应到每个 Failure Unit 节点的目的所在。

## “注意力”机制




在有向图中，a23 和 a32 分别代表着节点 2 对节点 3 的重要性和节点 3 对节点 2 的重要性，这个“重要性”由网络通过训练得出。这个“重要性”就是所谓的注意力。具体而言：

$$\alpha_{ij} = softmax(\sigma(\vec{a}^T[W\overrightarrow{h_i}||W\overrightarrow{h_j}]))$$

其中，W 表示权重矩阵，hi 表示 i 节点的节点特征，双竖线表示张量的拼接。例如：DejaVu 经过“Feature Extractor”过程得到的“Unit-level Feature”是 3×1 维的，那么 W 自然是 3×3 维，二者相乘得到 3×1 维的张量，经过拼接得到 6×1 维的张量。而向量 a 相当于一个 attention kernel，为 6×1 维，经过转置之后相乘得到 1 个实数，经过激活函数的变换，得到 i 节点对于 j 节点得到注意力系数。其实，这里的权重矩阵 W 和代表 attention kernel 的向量 a 就可以类比视为图像 CNN 卷积时需要自行学习的卷积核参数。

$$\overrightarrow{h_i} = \sigma(\sum_{j \in N_i} \alpha_{ij} \mathbf{W} \overrightarrow{h_j})$$

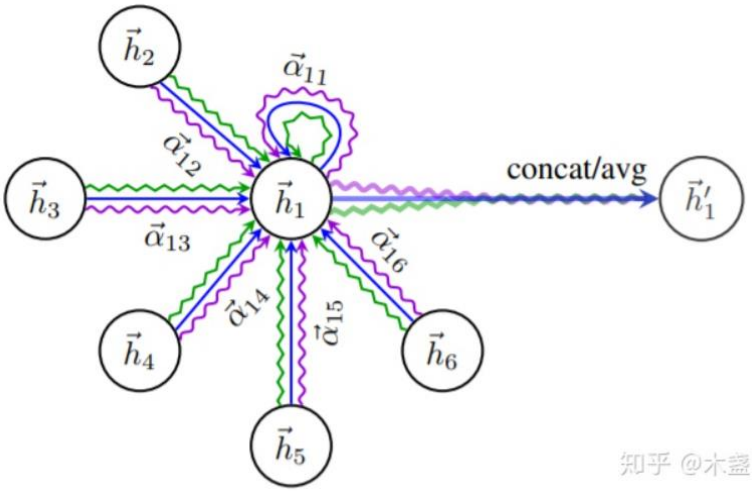
进一步如上式所示，Ni 表示节点 i 的邻接节点，经过累计求和以及 softmax 函数之后，得到的 hi 就是最终节点 i 的输出特征。事实上由公式中的累计求和可以看出，

GAT 是将邻居节点的特征聚合到中心节点上，其实就是一种特征聚合运算。

以上所述构成了 GAT 的基本组成组件，GAL，图注意力层。

## 多头注意力机制

对于 GAL 而言，它可以完全仿照 CNN 的操作：CNN 中对于每一层特征图的卷积核，其实可以有多个，而且每个卷积核相互独立，从而使得输出特征图具有更多的 channel。GAL 经过这样的操作之后，组合成为了 GAT：

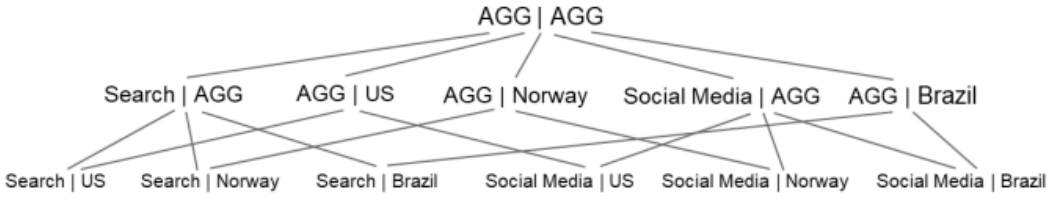



每个节点到节点 1 都有 3 条波浪线。这 3 条波浪线就代表 3 个独立的 attention 系数，独立学习，并且有着独立的注意力系数矩阵。相应的每个节点的输出对应的计算公式变换如下：

$$\overrightarrow{h_i'} = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in N_i}\alpha_{ij}^{k}\mathbf{W}^{k}\overrightarrow{h_j}\right)$$

**GAT 在 CMMD 中的应用和复现**

建图方式和原论文一致：




① 方案一

省略 DejaVu 中的 Feature Extractor 阶段，直接用 Squeeze 中的 Deviation Score 作为每个节点对应的特征值，此时 W 权重矩阵退化为一个实数。选择得分最高的 n 个节点或者超过阈值的节点作为最终的根因。

这样做的优点是操作相对简单，对 CMMD 原论文的算法进行了简化，但是最终效果如何难以估计。

②方案二
严格按照 CMMD 原论文算法步骤，利用剔除异常值之后的数据训练 GAT，利用 GAT 拟合测量指标节点之间的函数关系，利用这种函数关系替代 GRE，之后按照 CMMD 中所设定的适应度函数使用遗传算法选出最终根因。

# 七、NMS 算法尝试

**基本认知：**

根因分析里为什么要基于预测值和实际值的差距搜索根因？这个“差值”包含的信息是什么？

① 事实上，预测值和实际值的差值也是涟漪效应的基础，但涟漪效应对差值信息的利用可能并不合适
② 实际上，理解这个问题应当从统计学中时序分析的角度出发：做预测的方法有很多：MA 模型、AR 模型、ARMA 模型、ARIMA 模型、卡尔曼滤波……都能做预测，但是“预测”和“滤波”有什么关系？时序分析到底要分析什么？
③ 凡是时序分析，都是要把输入的一段时序序列所包含的信息分成两部分：一部分是噪声，可以理解成在时间上完全没有规律的随机值；另一部分是时序特征，也就是一段时序数据分离出噪声之后所剩下的特征。有了这个时序特征，能做从已知推导未知，也就是做预测；而把噪声从时序数据中心“过滤”出去的这个时序分析过程，就是“滤波”的过程，毕竟“波”也是一种时序数据。
④ 所以，另一个问题，被“滤波”过滤出去的噪声就是完全没用的吗？“事故”或者“异常”是归属于特征呢还是一种噪声呢？
显然，答案是后者，而且可以做一个合理的假设：导致“事故”或者“异常”的时候，往往是“噪声”比较大的时候。而预测值和实际值之间的差值，就是最直观的表达“噪声”的描述方式。

⑤ 所以，在论文 HotSot 中所提出的涟漪效益无异于要求一个有限的数字在庞大的“噪声背景”中持续稳定传播，这种假设显然太过理想化，而更接近实际的状态是，一个有

限大小的数字在庞大的“噪声背景”中传播时，会以指数的速度衰减。

| $f(p,i)\rightarrow v(p,i)$ | | Province($p$) | | | |
|---|---|---|---|---|---|
| | | Beijing | Shanghai | Guangdong | * |
| ISP ($i$) | Mobile | 20→8 | 15→15 | 10→10 | 45→33 |
| | Unicom | 10→4 | 25→25 | 20→20 | 55→49 |
| | * | 30→12 | 40→40 | 30→30 | 100→82 |

## 原理基础：

① 导致“事故”或者“异常”的时候，往往是“噪声”比较大的时候

② Squeeze 论文中所提出的 Deviation Score

虽然 Squeeze 原论文中提出的 Deviation Score 是用于解决基于涟漪效应的聚类问题，

但是，可以对这样的 Deviation Score 在原理①的基础上重新解释并发现其良好的数学

性能，当然，乘 2 并不是必要的。

对 Deviation Score 的重新解释：

原论文公式：

$$d(e) = 2 * \frac{f(e) - v(e)}{f(e) + v(e)}$$

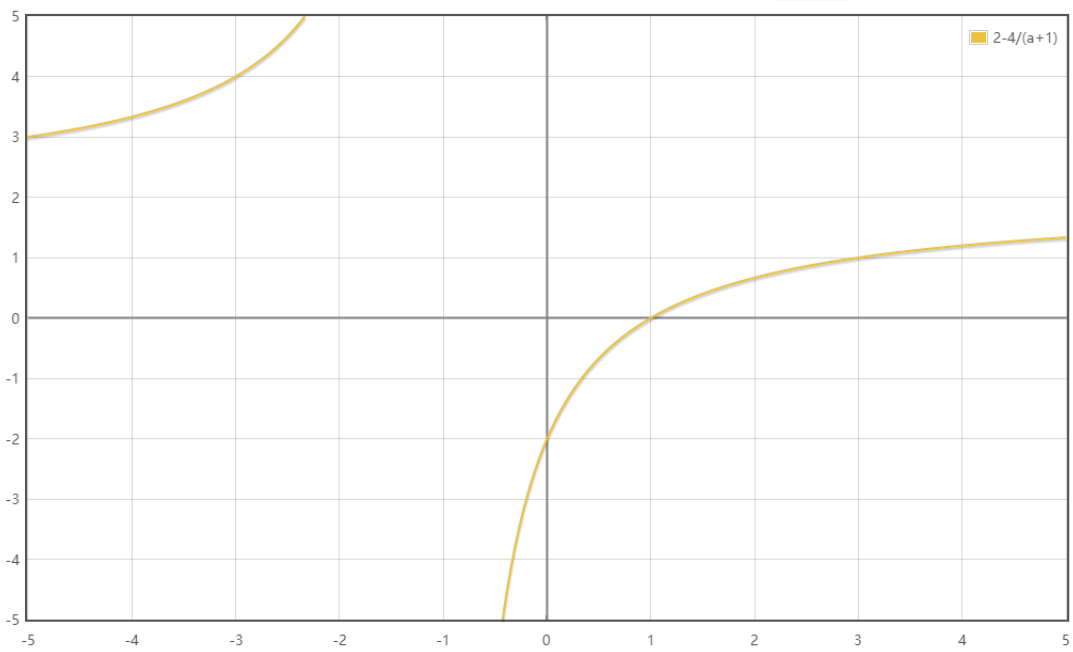
设：

$$f(e) = a \times v(e)$$

则上式可变化为（0 除外，单独考虑）：

$$d(e) = 2 - \frac{4}{a+1}$$

在实际应用中只需要关注这个反比例函数的下半部分：




注意，上图中，下半部分反比例函数的渐近线在 y=2，同时要保证 a>-1

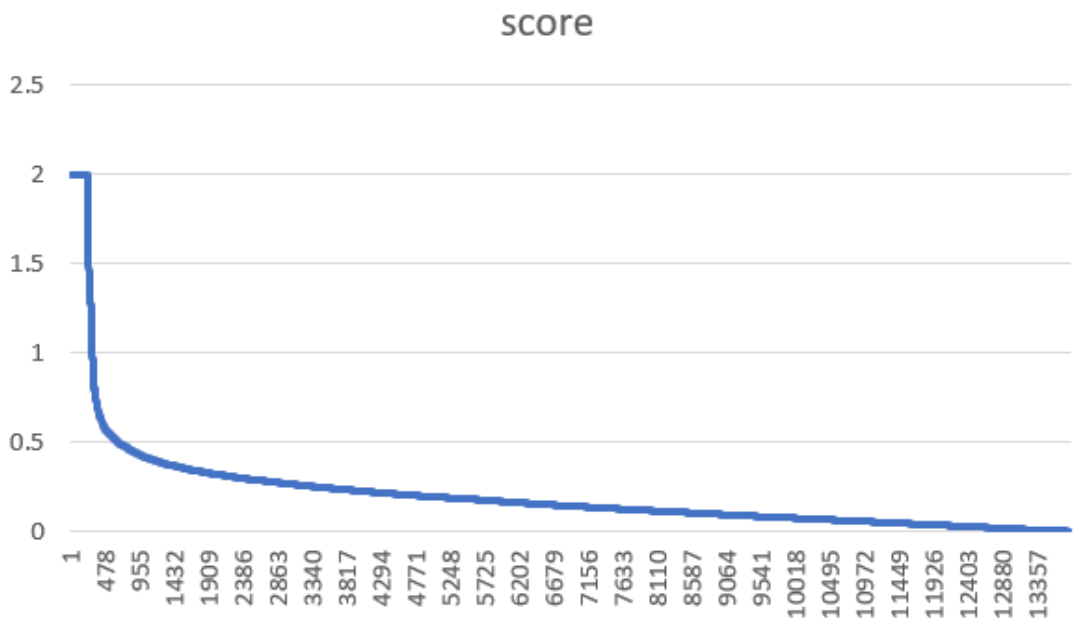
当 d(e)的值域取[-1, 1]时，a 对应的范围在[1/3, 3]，意味着预测值是实际值的 1/3 到 3 倍

从实际经验来看，这已经足够容纳绝大多数预测值和实际值之间的差距，同时这段区间是曲线斜率变化较为适中的阶段，既不至于斜率变化过快导致 score 扎堆拥挤，又不至于斜率变化太慢导致 score 分布太过稀疏。适中的斜率对后续的步骤提供了便利和保障。

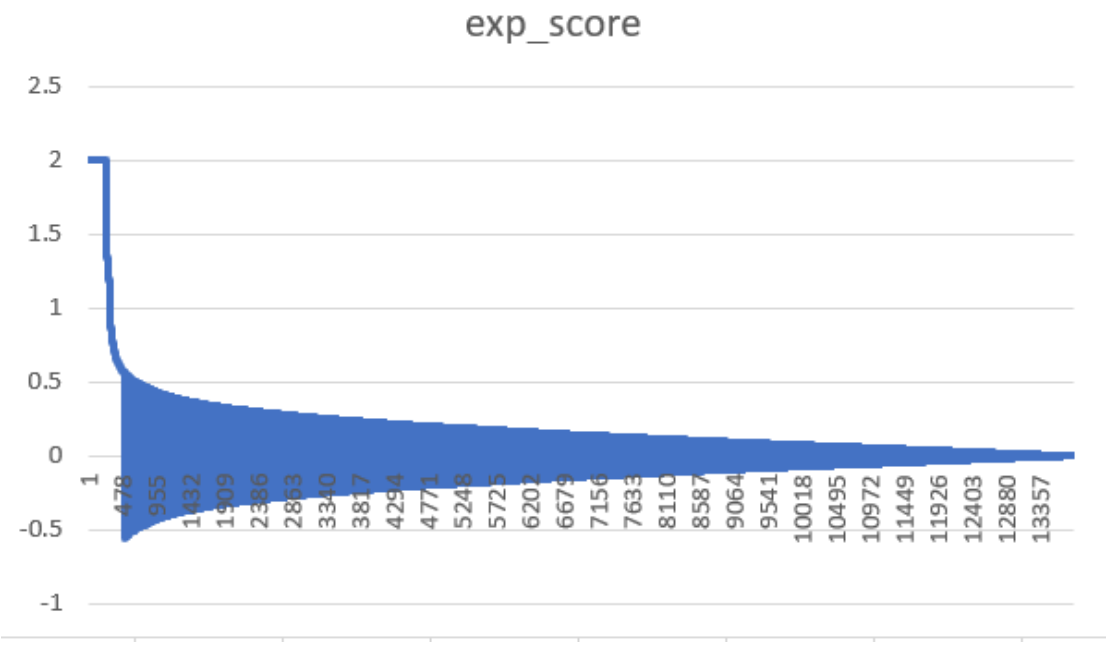
**不过，需要对 Deviation Score 做适当的修改：**

① 数学 1 公式中的乘 2 并不是必须的

② 更希望求实际值和预测值之间差值的绝对值而不是直接的差值，因为：
绝对值下，deviation score 的衰减情况是这样的：




直接求差值时，deviation score 的衰减情况是这样的：




**基于原理①，只需要关注绝对值大小即可，而不需要像涟漪效应那样关注差值的正负。因此，最终新方法所采用的 Deviation Score 如下：**

$$d(e) = \frac{|f(e) - v(e)|}{f(e) + v(e)}$$

# 算法步骤：

① 给定一个时间戳，计算该时间戳下所有叶节点属性集合的 Deviation Score，并基于 Deviation Score 进行排序；

② 分类讨论：

1.若实际值和预测值都为 0，不存在异常，但会导致计算 Deviation Score 时分母为 0，这一种情况在步骤①计算之前就应该处理；

2.若(实际值为 0，预测值不为 0)或(实际值不为 0，预测值为 0)，此时 Deviation Score 的计算结果恒为 2，所以是否存在异常一方面要看预测值和实际值之差的绝对值是否较大，另一方面要参考 NMS 非极大值抑制的思路，查看当前的叶节点属性集合中的元素是否和 Deviation Score 排名靠前的其他叶节点属性集合的交集属性数量是否较多：

若交集不多，则应当舍去，因为即使没有异常，例如当服务器重启时，其关联的流量也会从 0 开始增长，而只要实际值为 0，Deviation Score 就不可避免地会恒为 2；

若交集数量较多，则应当以预测值和实际值之间差值绝对值最大的为基准，对其余的叶节点属性集合实行 NMS

例：D23-1450689300.csv

| A | B | C | D | E | F | G | I | J |
|---|---|---|---|---|---|---|---|---|
| a | b | c | d | real | predict | diff | add | score |
| a4 | b17 | c1 | d1 | 0 | 2.26447 | 2.26447 | 2.26447 | 2 |
| a4 | b17 | c3 | d1 | 0 | 1.031837 | 1.031837 | 1.031837 | 2 |
| a4 | b17 | c4 | d1 | 0 | 1.306052 | 1.306052 | 1.306052 | 2 |
| a4 | b17 | c2 | d1 | 0 | 3158.651 | 3158.651 | 3158.651 | 2 |
| a4 | b2 | c5 | d1 | 0 | 15.42557 | 15.42557 | 15.42557 | 2 |
| a4 | b2 | c6 | d1 | 0 | 486.1622 | 486.1622 | 486.1622 | 2 |
| a4 | b2 | c1 | d1 | 0 | 492.3097 | 492.3097 | 492.3097 | 2 |
| a4 | b2 | c3 | d1 | 0 | 5758.436 | 5758.436 | 5758.436 | 2 |
| a4 | b2 | c7 | d1 | 0 | 526.7207 | 526.7207 | 526.7207 | 2 |
| a4 | b2 | c8 | d1 | 0 | 10.92685 | 10.92685 | 10.92685 | 2 |
| a4 | b2 | c4 | d1 | 0 | 326.418 | 326.418 | 326.418 | 2 |
| a4 | b2 | c2 | d1 | 0 | 38.43027 | 38.43027 | 38.43027 | 2 |
| a4 | b18 | c2 | d1 | 0 | 6.400092 | 6.400092 | 6.400092 | 2 |
| a4 | b19 | c5 | d1 | 0 | 545.0874 | 545.0874 | 545.0874 | 2 |
| a4 | b19 | c6 | d1 | 0 | 3115.044 | 3115.044 | 3115.044 | 2 |
| a4 | b19 | c1 | d1 | 0 | 23328.84 | 23328.84 | 23328.84 | 2 |
| a4 | b19 | c3 | d1 | 0 | 2146.283 | 2146.283 | 2146.283 | 2 |
| a4 | b19 | c7 | d1 | 0 | 2105.122 | 2105.122 | 2105.122 | 2 |
| a4 | b19 | c8 | d1 | 0 | 21854.65 | 21854.65 | 21854.65 | 2 |
| a4 | b19 | c9 | d1 | 0 | 14.81829 | 14.81829 | 14.81829 | 2 |
| a4 | b19 | c4 | d1 | 0 | 5223.934 | 5223.934 | 5223.934 | 2 |
| a4 | b19 | c2 | d1 | 0 | 12226.41 | 12226.41 | 1222 | 2 |
| a4 | b6 | c5 | d1 | 0 | 1.198228 | 1.198228 | 1.198228 | 2 |
| a4 | b6 | c6 | d1 | 0 | 36.48049 | 36.48049 | 36.48049 | 2 |
| a4 | b6 | c1 | d1 | 0 | 1480.211 | 1480.211 | 1480.211 | 2 |
| a4 | b6 | c3 | d1 | 0 | 3582.065 | 3582.065 | 3582.065 | 2 |
| a4 | b6 | c7 | d1 | 0 | 199.7156 | 199.7156 | 199.7156 | 2 |
| a4 | b6 | c8 | d1 | 0 | 796.7236 | 796.7236 | 796.7236 | 2 |
| a4 | b6 | c4 | d1 | 0 | 253.7502 | 253.7502 | 253.7502 | 2 |



以标红的叶节点属性集合为基准，对其他叶节点属性集合进行 NMS，

最终保留的 root cause 为 a4&d1

3.若实际值和预测值均不为 0，则正常执行 NMS 即可

不过此时，需要探讨一个问题：

3.1 若人为设置一个 score 阈值，score 小于阈值的直接舍弃不予考虑，对不小于阈值的叶节点属性结合执行 NMS

但，就 Squeeze A、B、D 数据集的初步验证情况来看，不同数据集的阈值差别很大，生硬的设置阈值会导致无用的调参；

3.2 人为设置一个最终希望得到的根因集合个数的阈值和 IOU 阈值，配合交集属性数量自动考虑合适终止 NMS，这样，当实际根因集合的数量多于人为设置的阈值时，程序即使得到了预期数量的根因，但仍然可以通过依旧存在交集属性数量较多的这一条件，适当多输出几个根因集合；当实际根因集合的数量少于人为设置的阈值时，程序即使没

有得到预期数量的根因，但仍然可以通过目前存在交集的属性数量已经低于阈值的这一条件，结束 NMS 计算过程。之所以不是只设置一个 IOU 阈值或者只是设置一个预期得到的根因集合数量是因为，在 Squeeze A、B、D 数据集中进行初步验证的时候观察到，单纯基于 IOU 阈值，最终的 NMS 操作的叶节点属性集合远多于人为控制预期根因数量的情况，但仅仅依靠人为设定预期的根因集合数量，又可能导致漏检或多检。

# 初步验证：

利用 excel 表格进行了简单的实验验证(代码还没写)

**事实上，这个可能有用的新方法执行起来比较简单，给定一个时间戳的 CSV 文件，单纯依靠 excel 和直接观察，就可以在 10 分钟差不多的时间确定根因。**

选择了下面 4 个文件

A33-1535731200.csv

B3-33-1450653900.csv

B4-33-1451030340.csv

D-23-1450689300.csv

进行了验证，算法步骤①直接利用 excel 的功能即可实现，算法执行 NMS 步骤时，其实就 Squeeze A、B、D 数据集而言，在执行算法步骤①之后，已经可以直接观察到明显的根因集合了。

**1.A33-1535731200.csv**

按照 NMS 的执行逻辑，显然：

pred cause = i38&e10&p15; i06&p05&l2; i47&c1&p35&l3

GT cause = i38&e10&p15; i06&p05&l2; i47&c1&p35

| 1 | i | e | c | p | l | real | predict | diff | add | score |
|---|---|---|---|---|---|---|---|---|---|---|
| 12983 | i06 | e09 | c3 | p05 | l2 | 0 | 1 | 1 | 1 | 2 |
| 12984 | i38 | e10 | c1 | p15 | l3 | 9798 | 23868 | 14070 | 33666 | 0.8358581 |
| 12985 | | | | | | | | | | |
| 12986 | i38 | e10 | c1 | p15 | l2 | 77 | 186 | 109 | 263 | 0.8288973 |
| 12987 | i38 | e10 | c1 | p15 | l4 | 1124 | 2644 | 1520 | 3768 | 0.8067941 |
| 12988 | i06 | e07 | c5 | p05 | l2 | 9 | 21 | 12 | 30 | 0.8 |
| 12989 | i38 | e10 | c4 | p15 | l3 | 518 | 1207 | 689 | 1725 | 0.7988406 |
| 12990 | i06 | e11 | c5 | p05 | l2 | 165 | 379 | 214 | 544 | 0.7867647 |
| 12991 | i06 | e08 | c5 | p05 | l2 | 714 | 1637 | 923 | 2351 | 0.7851978 |
| 12992 | i06 | e10 | c5 | p05 | l2 | 60 | 135 | 75 | 195 | 0.7692308 |
| 12993 | i38 | e10 | c4 | p15 | l4 | 43 | 95 | 52 | 138 | 0.7536232 |
| 12994 | i38 | e10 | c1 | p15 | l1 | 65 | 141 | 76 | 206 | 0.7378641 |
| 12995 | i06 | e09 | c5 | p05 | l2 | 32 | 68 | 36 | 100 | 0.72 |
| 12996 | i38 | e10 | c4 | p15 | l1 | 20 | 42 | 22 | 62 | 0.7096774 |
| 12997 | i38 | e10 | c4 | p15 | l2 | 2 | 4 | 2 | 6 | 0.6666667 |
| 12998 | i47 | e01 | c1 | p35 | l3 | 182 | 290 | 108 | 472 | 0.4576271 |
| 12999 | i47 | e04 | c1 | p35 | l3 | 210 | 333 | 123 | 543 | 0.4530387 |
| 13000 | i47 | e09 | c1 | p35 | l3 | 196 | 298 | 102 | 494 | 0.4129555 |
| 13001 | i14 | e11 | c5 | p29 | l3 | 3 | 4 | 1 | 7 | 0.2857143 |
| 13002 | i38 | e10 | c4 | p01 | l2 | 4 | 5 | 1 | 9 | 0.2222222 |
| 13003 | i16 | e04 | c5 | p15 | l3 | 5 | 6 | 1 | 11 | 0.1818182 |
| 13004 | i17 | e08 | c5 | p35 | l3 | 5 | 6 | 1 | 11 | 0.1818182 |
| 13005 | i38 | e09 | c1 | p28 | l2 | 5 | 6 | 1 | 11 | 0.1818182 |
| 13006 | i38 | e11 | c4 | p04 | l2 | 6 | 5 | 1 | 11 | 0.1818182 |
| 13007 | i02 | e08 | c1 | p13 | l3 | 202 | 241 | 39 | 443 | 0.1760722 |
| 13008 | i06 | e09 | c5 | p08 | l2 | 25 | 21 | 4 | 46 | 0.173913 |
| 13009 | i34 | e03 | c5 | p21 | l3 | 48 | 57 | 9 | 105 | 0.1714286 |
| 13010 | i02 | e09 | c1 | p24 | l3 | 11 | 13 | 2 | 24 | 0.1666667 |
| 13011 | i38 | e10 | c1 | p02 | l1 | 13 | 11 | 2 | 24 | 0.1666667 |
| 13012 | i06 | e12 | c5 | p12 | l3 | 492 | 417 | 75 | 909 | 0.1650165 |
| 13013 | i38 | e06 | c1 | p10 | l3 | 40 | 47 | 7 | 87 | 0.1609195 |
| 13014 | i38 | e10 | c1 | p26 | l2 | 21 | 18 | 3 | 39 | 0.1538462 |
| 13015 | i34 | e03 | c5 | p32 | l3 | 12 | 14 | 2 | 26 | 0.1538462 |

而 Squeeze 的计算结果为：


```
{
    "timestamp": 1535731200,
    "elapsed_time": 5.176909685134888,
    "root_cause": "i=i38&p=p15"
},
```


GT cause = i38&e10&p15; i06&p05&l2; i47&c1&p35

按照 Squeeze 的计算逻辑：

新算法： TP=2, FP=1, FN=1

Squeeze: TP=0, FP=1, FN=3

**2.B3-33-1450653900.csv**

按照 NMS 的执行逻辑，显然：

pred cause = b8&c5&d5; a4&b4&d5; a1&b25&c4

GT cause = b8&c5&d5; a4&b4&d5; a1&b25&c4

| | A | B | C | D | E | F | G | I | J |
|---|---|---|---|---|---|---|---|---|---|
| 1 | a | b | c | d | real | predict | diff | add | score |
| 7808 | a5 | b8 | c5 | d5 | 0 | 70 | 70 | 70 | 2 |
| 7809 | a6 | b8 | c5 | d5 | 0 | 579 | 579 | 579 | 2 |
| 7810 | a4 | b8 | c5 | d5 | 0 | 77 | 77 | 77 | 2 |
| 7811 | a1 | b8 | c5 | d5 | 0 | 216 | 216 | 216 | 2 |
| 7812 | a3 | b8 | c5 | d5 | 0 | 218 | 218 | 218 | 2 |
| 7813 | a2 | b8 | c5 | d5 | 0 | 143 | 143 | 143 | 2 |
| 7814 | a4 | b4 | c3 | d5 | 640.0209 | 2234 | 1593.979 | 2874.021 | 1.109233 |
| 7815 | a4 | b4 | c5 | d5 | 0.984827 | 3 | 2.015173 | 3.984827 | 1.011423 |
| 7816 | a4 | b4 | c2 | d5 | 16.53405 | 48 | 31.46595 | 64.53405 | 0.975173 |
| 7817 | a4 | b4 | c4 | d5 | 337.5368 | 960 | 622.4632 | 1297.537 | 0.959454 |
| 7818 | a4 | b4 | c1 | d5 | 162.0015 | 454 | 291.9985 | 616.0015 | 0.948045 |
| 7819 | a4 | b4 | c6 | d5 | 7.130581 | 18 | 10.86942 | 25.13058 | 0.865035 |
| 7820 | a4 | b4 | c7 | d5 | 48.89232 | 120 | 71.10768 | 168.8923 | 0.842048 |
| 7821 | a4 | b4 | c8 | d5 | 31.28719 | 71 | 39.71281 | 102.2872 | 0.776496 |
| 7822 | a1 | b35 | c3 | d7 | 0.671664 | 1.405 | 0.733336 | 2.076664 | 0.706263 |
| 7823 | a1 | b2 | c1 | d2 | 575.6987 | 1140.445 | 564.7463 | 1716.144 | 0.658157 |
| 7824 | a1 | b25 | c4 | d2 | 479.2498 | 888 | 408.7502 | 1367.25 | 0.597916 |
| 7825 | a6 | b3 | c3 | d5 | 24.03307 | 42.825 | 18.79193 | 66.85807 | 0.562144 |
| 7826 | a5 | b26 | c2 | d3 | 6.809128 | 12 | 5.190872 | 18.80913 | 0.551952 |
| 7827 | a2 | b22 | c3 | d5 | 60.21907 | 105.005 | 44.78593 | 165.2241 | 0.542124 |
| 7828 | a3 | b9 | c5 | d7 | 0.580486 | 1 | 0.419514 | 1.580486 | 0.530868 |
| 7829 | a5 | b10 | c3 | d7 | 0.58223 | 1 | 0.41777 | 1.58223 | 0.528077 |
| 7830 | a1 | b25 | c4 | d3 | 216.7962 | 365 | 148.2038 | 581.7962 | 0.50947 |
| 7831 | a5 | b6 | c5 | d8 | 0.880145 | 1.4775 | 0.597355 | 2.357645 | 0.506739 |
| 7832 | a1 | b25 | c4 | d8 | 159.4966 | 266 | 106.5034 | 425.4966 | 0.500608 |
| 7833 | a1 | b25 | c4 | d6 | 532.7104 | 887 | 354.2896 | 1419.71 | 0.499101 |
| 7834 | a3 | b30 | c1 | d2 | 466.0767 | 759 | 292.9233 | 1225.077 | 0.478212 |
| 7835 | a2 | b15 | c3 | d1 | 2.144052 | 3.4875 | 1.343448 | 5.631552 | 0.477115 |
| 7836 | a1 | b25 | c4 | d4 | 222.4414 | 361 | 138.5586 | 583.4414 | 0.47497 |

1450653900

而 Squeeze 的计算结果为：


```
{
    "timestamp": 1450653900,
    "elapsed_time": 25.20356512069702,
    "root_cause": "a=a1&b=b25&c=c4&d=d2;a=a4&b=b4&d=d5;b=b8&c=c5&d=d5"
},
```


GT cause = b8&c5&d5; a4&b4&d5; a1&b25&c4

按照 Squeeze 的计算逻辑：

新算法： TP=3, FP=0, FN=0

Squeeze: TP=2, FP=1, FN=1

**3.B4-33-1451030340.csv**

pred cause = a4&b31&c1; a2&b7&c4;a3&b13&c1

GT cause = a4&b31&c1; a2&b7&c4;a3&b13&c1

按照 Squeeze 的计算逻辑:

新算法: TP=3, FP=0, FN=0

| | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| 1 | a | b | c | d | real | predict | diff | add | score |
| 7832 | a4 | b31 | c1 | d1 | 0 | 1496 | 1496 | 1496 | 1 |
| 7833 | a4 | b31 | c1 | d4 | 0 | 1507 | 1507 | 1507 | 1 |
| 7834 | a4 | b31 | c1 | d9 | 0 | 2736 | 2736 | 2736 | 1 |
| 7835 | a4 | b31 | c1 | d5 | 0 | 115 | 115 | 115 | 1 |
| 7836 | a4 | b31 | c1 | d3 | 0 | 1505 | 1505 | 1505 | 1 |
| 7837 | a4 | b31 | c1 | d10 | 0 | 45 | 45 | 45 | 1 |
| 7838 | a4 | b31 | c1 | d6 | 0 | 114 | 114 | 114 | 1 |
| 7839 | a4 | b31 | c1 | d7 | 0 | 1431 | 1431 | 1431 | 1 |
| 7840 | a4 | b31 | c1 | d2 | 0 | 114 | 114 | 114 | 1 |
| 7841 | a4 | b31 | c1 | d8 | 0 | 1333 | 1333 | 1333 | 1 |
| 7842 | a2 | b7 | c4 | d2 | 720.456 | 4806 | 4085.544 | 5526.456 | 0.73927 |
| 7843 | a2 | b7 | c4 | d8 | 1767.921 | 11627 | 9859.079 | 13394.92 | 0.736031 |
| 7844 | a2 | b7 | c4 | d6 | 796.7771 | 4786 | 3989.223 | 5582.777 | 0.714559 |
| 7845 | a2 | b7 | c4 | d5 | 908.1844 | 4788 | 3879.816 | 5696.184 | 0.681125 |
| 7846 | a2 | b7 | c4 | d10 | 227.0984 | 1114 | 886.9016 | 1341.098 | 0.661325 |
| 7847 | a2 | b7 | c4 | d9 | 188.5934 | 823 | 634.4066 | 1011.593 | 0.627136 |
| 7848 | a2 | b7 | c4 | d4 | 3070.598 | 13370 | 10299.4 | 16440.6 | 0.626462 |
| 7849 | a2 | b7 | c4 | d1 | 3186.524 | 13597 | 10410.48 | 16783.52 | 0.62028 |
| 7850 | a2 | b7 | c4 | d7 | 3021.1 | 12695 | 9673.9 | 15716.1 | 0.615541 |
| 7851 | a2 | b7 | c4 | d3 | 3325.199 | 13455 | 10129.8 | 16780.2 | 0.603676 |
| 7852 | a3 | b13 | c1 | d5 | 1923.52 | 5742 | 3818.48 | 7665.52 | 0.498137 |
| 7853 | a3 | b13 | c1 | d6 | 2003.01 | 5795 | 3791.99 | 7798.01 | 0.486277 |
| 7854 | a3 | b13 | c1 | d8 | 5776.625 | 16036 | 10259.38 | 21812.62 | 0.470341 |
| 7855 | a3 | b13 | c1 | d1 | 7196.404 | 19459 | 12262.6 | 26655.4 | 0.460042 |
| 7856 | a3 | b13 | c1 | d10 | 1690.521 | 4567 | 2876.479 | 6257.521 | 0.459683 |
| 7857 | a3 | b13 | c1 | d7 | 7181.623 | 18561 | 11379.38 | 25742.62 | 0.442044 |
| 7858 | a3 | b20 | c3 | d2 | 950.0492 | 2427.35 | 1477.301 | 3377.399 | 0.437408 |
| 7859 | a3 | b10 | c2 | d7 | 42.67362 | 104.395 | 61.72138 | 147.0686 | 0.419677 |
| 7860 | a2 | b4 | c6 | d3 | 6.177944 | 14.75 | 8.572056 | 20.92794 | 0.409599 |

**4.D-23-1450689300.csv**

pred cause = a4&d1; b16&d3

GT cause = a4&d1; b16&d3; a5&b34

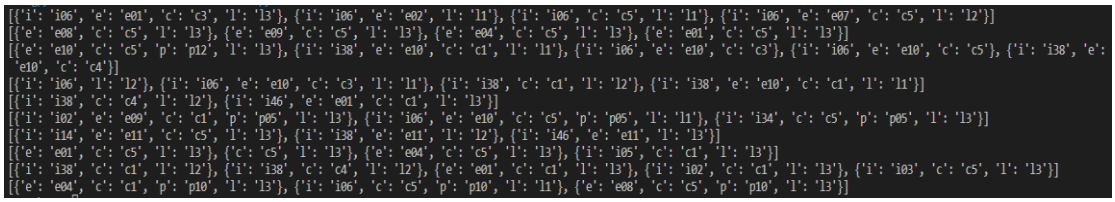
按照 Squeeze 的计算逻辑:

新算法: TP=2, FP=0, FN=1

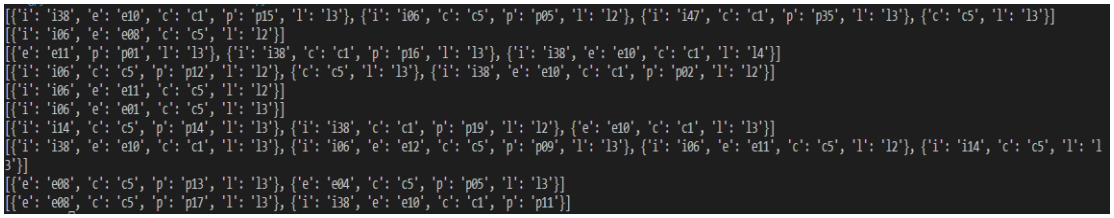
**算法实现**

按照 IOU>=0.3,重复次数>=2

(1,1)



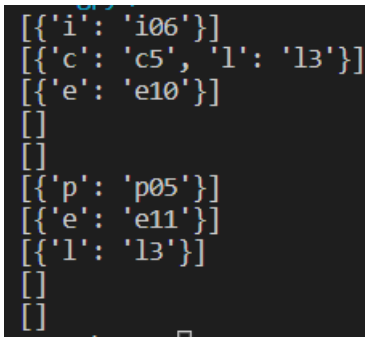

(3,3)



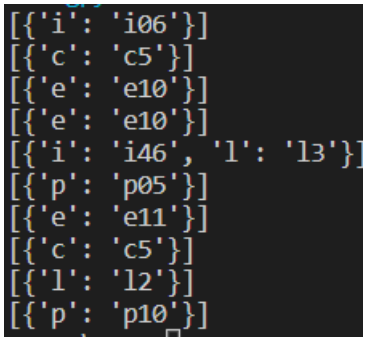

2,理想情况

对(1,1)而言，IOU>=0.3 并不合适，对于同一根因引起的很容易被归结为另一个新的根因。

=2 也不合适，对于不是根因的属性 l 来说只有 1-4　4 个值，取前 25 个值后 p>=2 很容易被归于根因。取 IOU>=0.1,p>=20 后效果有提升




没有推断出根因的原因是计算 Deviation Score 时 f(e)和 v(e)有一方为 0(比如(0,1),可能不是根因被取入了前 25 个值)当这样的数据过多时无法得到根因，因此大胆假设 |f(e)-v(e)|<=2 时的数据不是根因。

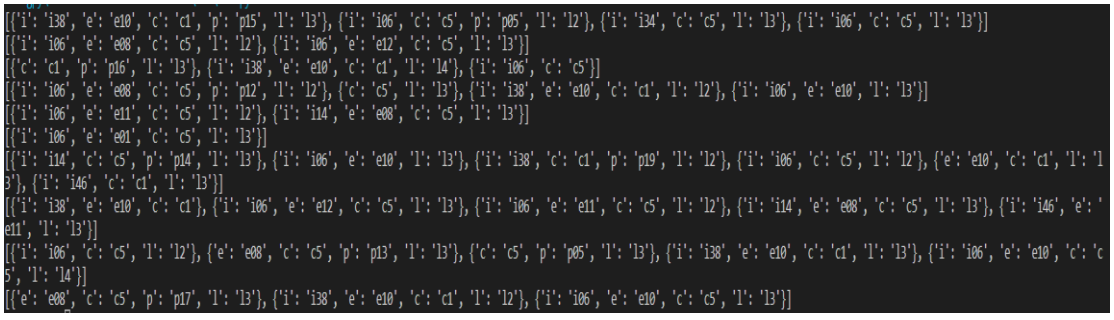



与实际根因相比效果不错

| | |
|---|---|
| i=i06 | 1535731200 |
| c=c5 | 1535733900 |
| e=e10 | 1535736000 |
| e=e10 | 1535738100 |
| i=i46 | 1535740200 |
| p=p05 | 1535742300 |
| e=e11 | 1535744700 |
| c=c5 | 1535748600 |
| l=l2 | 1535750400 |
| p=p10 | 1535753400 |

(3,3)

IOU>=0.3,取前 70 个值， p>4



| | | |
|---|---|---|
| 2 | i=i06&p=p05&l=l2;i=i38&e=e10&p=p15;i=i47&c=c1&p=p35 | 1535731200 |
| 3 | c=c5&p=p08&l=l4;e=e08&c=c5&l=l2;i=i06&e=e12&l=l3 | 1535733900 |
| 4 | e=e10&c=c1&l=l4;c=c1&p=p16&l=l3;e=e11&p=p01&l=l3 | 1535736000 |
| 5 | i=i06&p=p12&l=l2;i=i38&c=c1&p=p02;e=e11&c=c5&p=p19 | 1535738100 |
| 6 | i=i06&p=p35&l=l3;i=i06&e=e11&c=c5;i=i14&c=c5&l=l3 | 1535740200 |
| 7 | e=e10&c=c1&l=l3;i=i14&p=p14&l=l3;i=i06&c=c5&l=l3 | 1535742300 |
| 8 | i=i14&c=c5&p=p14;i=i06&p=p14&l=l3;c=c1&p=p19&l=l2 | 1535744700 |
| 9 | i=i38&p=p28&l=l3;i=i06&p=p09&l=l3;e=e11&p=p13&l=l2 | 1535748600 |
| 10 | c=c5&p=p13&l=l3;c=c5&p=p05&l=l3;i=i06&e=e09&p=p19 | 1535750400 |
| 11 | c=c5&p=p17&l=l3;e=e10&c=c1&p=p11;i=i38&p=p21&l=l3 | 1535753400 |

效果有提升，有些虽不是根因，但输出的结果基本包含根因。

想法：IOU 的取值，对根因不同的维度数量 IOU 取不同的值效果会更好，但无法实现。设想若 IOU 固定为 0.3，加大数据的选取同时加大 p 值，可能会有不错的效果

p 值的选取，从(3,3)来看，输出的根因基本包含实际根因，为什么不是根因的维度会输出，从数据集中可以看出，i 的取值范围是 1-147，e 是 1-13，c 是 1-7，p 是 1-35，l 是 1-4。

对于取值范围很小的属性，比如 l，即使他不是根因，由于随机性，取前 25 个数据 l 不是根因平均也会出现 6 次, 因此会错误的把他归于根因, 可以确定 p 值的时候分类确定，比如 i01 出现次数 3 次就可以归为根因时，l1 要出现 6 次才可以归为根因，可以会有更好的效果。

# 八、基于 deviation score 迅速衰减构造的两种方法

$$score = \frac{|predict - real|}{predict + real}$$

设置 score 阈值(例如：0.2)，按照 score 从高到低排序，取 score 大于阈值且不等于 1 的叶节点，之后分为两种方法：

## 1、建树法

### 基本流程

1.以通过 score 阈值筛选之后的叶节点作为基础层, 通过组合数来逐级向上形成父节点，

并求取每个节点的 score

例如：通过 score 阈值筛选之后得到的叶节点如下：

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 12986 | i38 | e10 | c1 | p15 | l2 | 77 | 186 | 109 | 263 |
| 12987 | i38 | e10 | c1 | p15 | l4 | 1124 | 2644 | 1520 | 3768 |
| 12988 | i06 | e07 | c5 | p05 | l2 | 9 | 21 | 12 | 30 |
| 12989 | i38 | e10 | c4 | p15 | l3 | 518 | 1207 | 689 | 1725 |
| 12990 | i06 | e11 | c5 | p05 | l2 | 165 | 379 | 214 | 544 |
| 12991 | i06 | e08 | c5 | p05 | l2 | 714 | 1637 | 923 | 2351 |
| 12992 | i06 | e10 | c5 | p05 | l2 | 60 | 135 | 75 | 195 |

对于 e=(i38, e10, c1, p15, l2)，通过组合数逐级形成其所有的父节点

{

(e10, c1, p15, l2)、(i38, c1, p15, l2)、(i38, e10, p15, l2)、(i38, e10, c1, l2)、(i38, e10, c1, p15)、

(c1, p15, l2)、(e10, p15, l2)、(e10, c1, l2)、(e10, c1, p15)、(i38, p15, l2)、

(i38, c1, l2)、(i38, c1, p15)、(i38, e10, l2)、(i38, e10, p15)、(i38, e10, c1)、

(i38, e10)、(i38, c1)、(i38, p15)、(i38, l2)、(e10, c1)、

(e10, p15)、(e10, l2)、(c1, p15)、(c1, l2)、(p15, l2)、

(i38)、(e10)、(c1)、(p15)、(l2)

}

非叶节点分数的计算方式为：

当前非叶节点对应的所有叶节点的 real 值累加，作为当前非叶节点的 real 值；

当前非叶节点对应的所有叶节点的 predict 值累加，作为当前非叶节点的 predict 值；

之后用当前非叶节点的 real 值和 predict 值计算 score

###这一步骤的出发点是，通过模拟建树的过程，从叶节点中罗列出所有可能的根因组合

2.每个节点对应的 score 分数都要乘 2 个因子：

①当前节点中组成元素的个数，例如：节点(e10, p15)对应的个数为 2

②当前节点对应的叶节点的数目，例如：在上表中，节点(e10, p15)对应的次数为 3

score×个数×次数=最终的分数

###这一步骤乘“次数”和“个数”这两个因子的出发点是，出现次数更多的节点更有可能是异常的根因节点，或者从“树”的角度来看，根因节点会具有跟多的入度，但是为了防止无法区分根因集合和其子集何方是根因，例如，真实根因为(i38, e10,p15)，若只乘次数，

则其子集，例如(i38, e10)将得到比(i38, e10,p15)更高的分数。

3.将最终得到的节点按照分数降序排列，按照从上到下的顺序遍历，判断当前节点和其他节点的重合度。

例如，按照最终的分数排序之后，得到的结果分数从高到低的顺序是：(e10, p15)、(e10, p15, l2)、(i38, c1)

则第二个节点和第一个节点重合度过高，去除第二个节点，保留第一个节点，第三个节点与其他节点均不重合，保留。

###这一步骤的出发点是，借鉴“简洁性”的考量指标，用最少的元素数量表示异常

## 结果与问题

此方法的结果和 Squeeze 相比更差，且具有较大差距。

存在以下两个问题：

① 基本流程 2 依旧未能达到理想的效果，无法有效区分子集和父集

② 算法涉及到组合问题，时间复杂度高，运行速度较慢

# 2、滑动窗口法

## 基本流程

例如：通过 score 阈值筛选之后得到的叶节点如下：

| | | | | |
|---|---|---|---|---|
| 7818 | a4 | b4 | c1 | d5 |
| 7819 | a4 | b4 | c6 | d5 |
| 7820 | a4 | b4 | c7 | d5 |
| 7821 | a4 | b4 | c8 | d5 |
| 7822 | a1 | b35 | c3 | d7 |
| 7823 | a1 | b2 | c1 | d2 |
| 7824 | a1 | b25 | c4 | d2 |
| 7825 | a6 | b3 | c3 | d5 |
| 7826 | a5 | b26 | c2 | d3 |
| 7827 | a2 | b22 | c3 | d5 |
| 7828 | a3 | b9 | c5 | d7 |
| 7829 | a5 | b10 | c3 | d7 |
| 7830 | a1 | b25 | c4 | d3 |
| 7831 | a5 | b6 | c5 | d8 |
| 7832 | a1 | b25 | c4 | d8 |
| 7833 | a1 | b25 | c4 | d6 |

可以观察到，最终包含根因的叶节点存在“聚集”现象，程序可以表示为：在覆盖 4 行的“窗口”中，包含根因的叶节点能够至少稳定出现 3 次，基于这样的出发点，算法流程为：

1.设置窗口大小(例如：4)，允许中断的次数(例如：1)，从上到下滑动窗口：窗口的初始位置在[7818, 7821]行，得到(7818, 7819, 7820)、(7818, 7819, 7821)、(7818, 7820, 7821)、(7819, 7820, 7821)这四种求交集的组合，可以得到四种组合下所求交集均为(a4、b4、d5)，将交集结果送入备选项中；若一个窗口所对应的四种组合交集不唯一，则取交集元素最多的结果或者如果存在多于一个的相同元素数目的交集组合，则再次计算元素数目相同的交集组合的交集，若结果非空，则送入备选项；即，每个窗口只能产出 0 个或 1 个根因备选项。

2.若备选项中已经存在和当前企图加入的元素相同的元素，则直接对根因备选项中的 count+1 即可。

3.将备选项列表中的元素按照出现次数降序排列，判断备选项中是否存在相同的元素或者存在父子集合关系的元素，对于相同的元素，直接合并即可，元素对应的 count+1；若是存在父子集合关系的元素，只保留出现次数较多的一方。若存在父子集合关系的元素数目相同，则保留子集。将最终筛选之后的元素视为根因。

## 结果与问题

| tp | fp | fn | f1 | precision | recall | length | |
|---|---|---|---|---|---|---|---|
| 513 | 870 | 972 | 0.3578 | 0.3709 | 0.3455 | 495 | a33* |
| 327 | 710 | 1158 | 0.2593 | 0.3153 | 0.2202 | 495 | a33 |
| | | | | | | | |
| 230 | 752 | 760 | 0.2332 | 0.2342 | 0.2323 | 495 | a23* |
| 313 | 519 | 677 | 0.3436 | 0.3762 | 0.3162 | 495 | a23 |
| | | | | | | | |
| 454 | 1007 | 1031 | 0.3082 | 0.3107 | 0.3057 | 495 | a32* |
| 622 | 421 | 857 | 0.4932 | 0.5964 | 0.4206 | 495 | a32 |

在 Squeeze-A33 数据集上的 F1 值比 Squeeze 方法提高了约 0.1，但在其他数据集测试中发现，效果差于 Squeeze 且相差较大。不过可以观察到，当 element 减小时，此方法往往会更明显地出现“多检”，导致 F1 值下降。

###备注，尝试过将滑动窗口的筛选结果替换 Squeeze 拐点法过滤筛选的结果，之后再进行 Squeeze 中基于密度的聚类，但是滑动窗口筛选所得的结果往往数量规模很少，已经不具备聚类的条件，实践中，聚类也无法形成较好的聚类结果

# 九、建树法最终版本

建树法在前期的方向上继续改进，对每一个节点的得分进行评估时，不仅评估他是否导致了故障(得分增加)，同时考虑如果他没有导致故障，增加了他不是根因的可能性，这时也要减去一部分得分，以下是算法的详细步骤：

1. 读取指定文件夹中的 injection_info.csv 文件，获取所有时间戳值。

2. 遍历所有文件，并查找以时间戳命名的 CSV 文件。

3. 对每个 CSV 文件，从文件中读取真实值和预测值。对于数据行中的每个元素，根据其属于哪个组件，将其分配到相应的变量中。

4. 根据真实值和预测值之间的差异，选择所有满足条件的行，并按照这些行的偏差分数排序。

5. 针对所有选中的行，计算出所有可能的子集合，并将这些子集合存储在一个集合中。

6. 针对所有选中的行，统计每个子集合得分，并将得分存储在一个字典中。如果子集合是根因，则它越可能与异常数据相关，并具有更高的得分。与之相反，如果子集合出现在正常数据中，则它的得分会降低。

7. 对字典中所有的子集合按得分排序，并筛选出最可能是根因的节点或子集合（称为

“root cause”）。这些节点被认为与异常数据最相关，具有较高的得分。

8. 对于给定的“root cause”，算法尝试去除重复项并且确定哪些节点是实际上造成问题的根因。这一步骤包括如下几个子步骤：

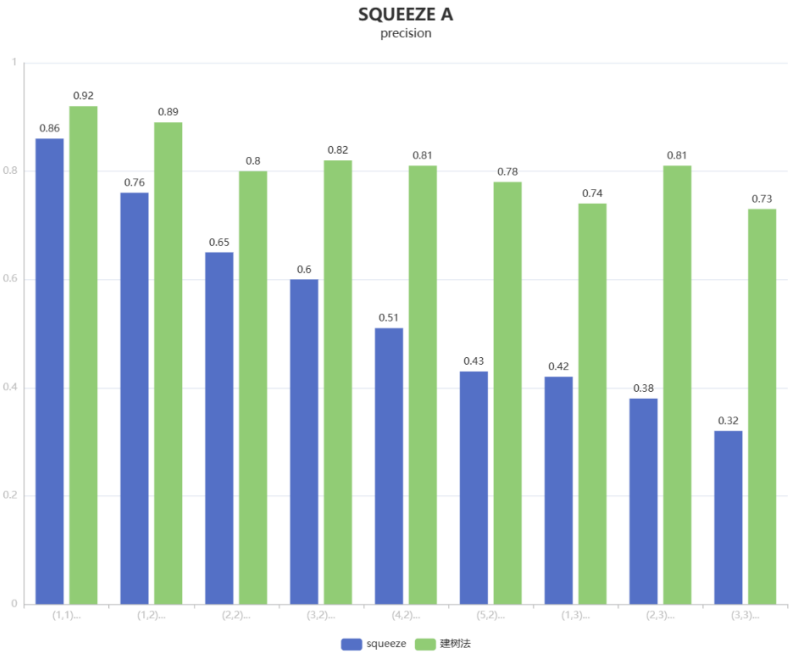
9. 将“root cause”按长度排序（从短到长），并将首个元素视为潜在的根因。

10. 对于每个可能的根因，检查是否与现有的根因重叠。如果是，则将其删除。这是避免重复根因的过程。

11. 针对剩余的根因，再次进行排序，并筛选最可能是实际根因的节点或子集合。

12. 最终输出被认为是实际根因的节点或子集合。

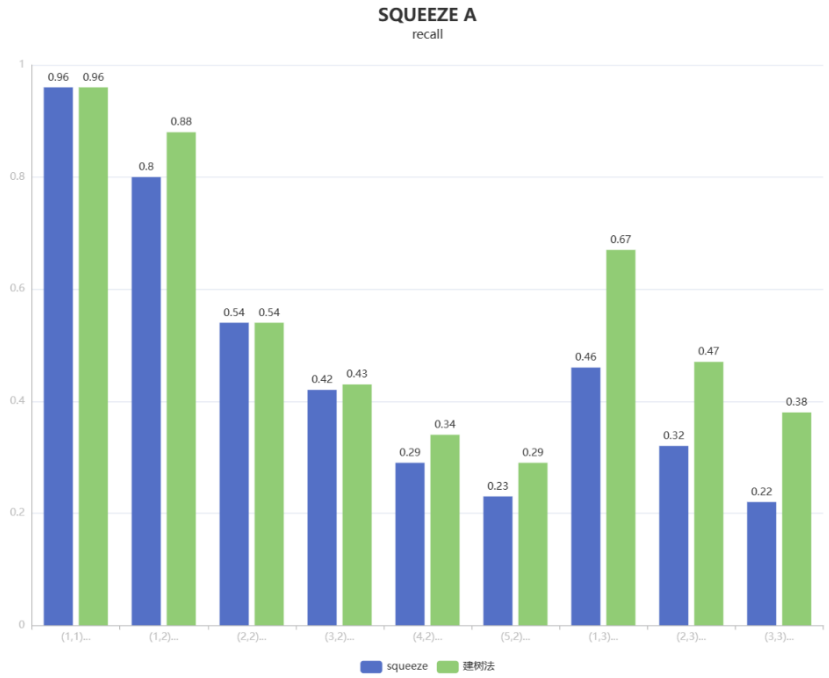
结果对比




查准率（Precision）表示正样本预测为正样本的比例。其计算公式为：

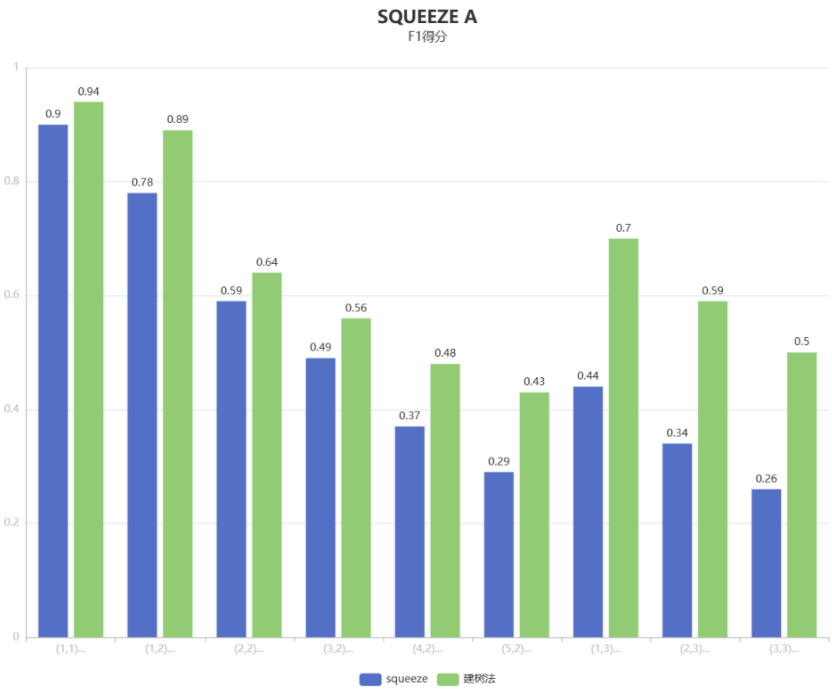
Precision=TP/TP+FP

通过上图可以看出我们的算法更准确地预测正例，避免把负例预测成正例的情况，从而提高准确性。




首先，recall 是一个评估分类模型性能的指标，反映了模型正确检测到正样本的能力。其计算公式为：recall=TP/TP+FN

通过表可以看出：我们的算法能够更好地捕捉正样本，即更多的真正例被检测出来。有更高的准确性，即正确识别出的异常数据更可能是真正的异常数据。




从 F1 得分可以看出，我们的算法在综合考虑查准率和查全率的情况下，具有更好的表现。在实际应用中具有更高的准确性和可靠性。