

复现第三周进度(4.8-4.14)

Anomaly Detection--异常检测

Anomaly Generation

WSU CASAS hh125 rules日志数据集的信息：

- 时间：2023-03-01---2023-04-30。HAWatcher是三周数据训练，一周数据测试，可以模仿。这里我选择04-01 ---- 04-14两周内的数据作为测试，其余数据用于训练。
- 设备：8 Motion Sensors (M), 2 Door Sensor (D), 6 Temperature Sensors (T),4 lights(L),3 fans(Fan)
- 日志遵循格式“日期 时间\t设备编号\t状态”

进度记录

4.8

编写anomaly_detection.py

4.9-4.10

编写anomaly_detection.py

小插曲

- 在semantic_analysis.py中添加从文件中读取相关性列表的功能（不然每次main.py生成相关性要花好长时间），因此需要想一种方法**能够将一个列表保存在一个特定格式的文件中，python事先在保存时就知晓了这个列表所有的格式和数据结构，因此在读取时就很方便**

之前将相关性保存在txt文件中，结果读取时非常不方便，查阅资料得知可以采用下面这种方法：

使用JSON

JSON（JavaScript Object Notation）是一种轻量级的数据交换格式，易于人阅读和编写，同时也易于机器解析和生成。Python内置的 `json` 模块可以直接处理Python列表和字典等，将它们转换为JSON格式字符串，并保存到文件中。读取时，可以将JSON格式的字符串恢复为Python的数据结构。

如图

```
MAIN.py × Allcorrelations_after_refining.txt × Allcorrelations_after_refining.json × seman
1  [
2  {
3      "event": {
4          "subject": "M002",
5          "attribute": "motion",
6          "constraint": "ON",
7          "extraConstraint": null
8      },
9      "conditionState": null,
10     "followedEventOrState": {
11         "subject": "L001",
12         "attribute": "lightingMode",
13         "constraint": "ON",
14         "extraConstraint": null
15     },
16     "type": "e2e"
17 },
18 {
19     "event": {
20         "subject": "M002",
21         "attribute": "motion",
22         "constraint": "OFF",
23         "extraConstraint": null
24     },
25     "conditionState": null,
26     "followedEventOrState": {
27         "subject": "L001",
28         "attribute": "lightingMode",
29         "constraint": "OFF",
30         "extraConstraint": null
31     },
32     "type": "e2e"
33 },
34 {
35     "event": {
```

通过将生成的所有相关性保存在json文件的方式可以方便的实现文件读取。

```
# 保存到JSON文件
semantic_analysis.save_correlations_to_json(all_correlations,
'Allcorrelations_after_refining.json')

# 从JSON文件中读取
loaded_correlations =
semantic_analysis.load_correlations_from_json('Allcorrelations_after_refining.json')

# 打印读取到的correlations
for correlation in loaded_correlations:
    print(correlation)
print(len(loaded_correlations))
```

下面是读取结果，读取成功：

```

'e2s' correlation:
pre_event=Event(subject='Fan3', attribute='fanMode', constraint='ON',extraConstraint='None')
condition=None
following_state=State(subject='T104', attribute='temperature', constraint='>=25')
'e2s' correlation:
pre_event=Event(subject='Fan3', attribute='fanMode', constraint='OFF',extraConstraint='None')
condition=None
following_state=State(subject='T104', attribute='temperature', constraint='<26')
'e2s' correlation:
pre_event=Event(subject='T106', attribute='temperature', constraint='>=25',extraConstraint='None')
condition=None
following_state=State(subject='Fan1', attribute='fanMode', constraint='ON')
'e2s' correlation:
pre_event=Event(subject='T106', attribute='temperature', constraint='>=26',extraConstraint='None')
condition=None
following_state=State(subject='Fan2', attribute='fanMode', constraint='ON')
'e2s' correlation:
pre_event=Event(subject='T106', attribute='temperature', constraint='>=26',extraConstraint='None')
condition=None
following_state=State(subject='Fan3', attribute='fanMode', constraint='ON')
64

```

- 接下来在处理逻辑时，因为把一个continue不小心写成了break卡了一晚上没找出原因QAQ
- 接下来又有个问题：

```

Traceback (most recent call last):
  File "G:\学习\IOT\HAWatcher复现\MAIN.py", line 98, in <module>
    anomaly_detection.detection(loaded_all_correlations)
  File "G:\学习\IOT\HAWatcher复现\anomaly_detection.py", line 170, in detection
    devices[correlation.followedEventOrState.subject][
TypeError: unsupported operand type(s) for +: 'NoneType' and 'str'

```

因为温度一开始的值没有初始化（None），这块建议在device_information处对每个设备设置一个值表示被赋值过还是未初始化，另外，如果没被赋值过就不应该参与异常检测，否则会造成误报

对此有两种解决方法：

- 1.如果该日志对应的设备还未初始化，则依照该日志更新设备状态，然后跳过该条日志的异常检测
- 2.根据正常数据集手动/自动推断出最开始的设备状态并初始化

第一种方法由于跳过日志的操作，可能会跳过异常日志，造成误报，**因此不可取。**

第二种方法是最符合常理的，具体来讲，可以手动查看4月1号之前的设备的最后的状态，他们就是测试集设备的最初状态

注意：这在正常的家居异常检测中是不会出现未初始化的问题的，我们出现这个问题主要是因为分割数据集的原因

处理了这些问题后，发现误报率仍然很高，于是接着找问题

- 1.

```
anomaly detected in contextual checking:
2013-04-01 11:59:39.512483 M004 OFF
violated
'e2s' correlation:
pre_event=Event(subject='M004', attribute='motion', constraint='OFF',extraConstraint='None')
condition=None
following_state=State(subject='L003', attribute='lightingMode', constraint='OFF')
```

发现这个相关性报错了，分析可得日志没问题，是相关性还是不够严谨，所以是假设检验的问题：

```
'e2s' correlation:
pre_event=Event(subject='M004', attribute='motion', constraint='OFF',extraConstraint='None')
condition=None
following_state=State(subject='L003', attribute='lightingMode', constraint='OFF')
success_count=1390,total_count=1436
success_rate=0.967966573816156
p_value=0.0005852609228780649
```

发现成功率很高了，但是还是不够，所以成功率得再高一点

把p0改成了0.98，相关性从64个降到48个，误报显而易见的从二百多降到了一百多。说明之前确实筛的还是不够严。

2.

```
anomaly detected in contextual checking:
2013-04-01 11:59:42.982791 M003 OFF
violated
'e2s' correlation:
pre_event=Event(subject='M003', attribute='motion', constraint='OFF',extraConstraint='None')
condition=None
following_state=State(subject='L003', attribute='lightingMode', constraint='ON')
```

```
'e2s' correlation:
pre_event=Event(subject='M003', attribute='motion', constraint='OFF',extraConstraint='None')
condition=None
following_state=State(subject='L003', attribute='lightingMode', constraint='ON')
success_count=4558,total_count=4559
success_rate=0.9997806536521167
p_value=9.397135139798585e-39
```

6	M001 OFF and M003 OFF	L003 OFF
---	-----------------------	----------

这个几乎无解，根据规则看，这个相关性确实不严谨，但无奈成功率确实是太高了，分析可得，这个相关性是physical channel生成的，按理说会被更严谨的smartapp channel覆盖，那为什么上面这条没有在refine部分被覆盖呢？如果smartapp channel中有下面这个相关性，就可以覆盖掉上面不严谨的相关性

```
anomaly detected in contextual checking:
2013-04-01 11:59:42.982791      M003  OFF
violated
'e2s' correlation:
pre_event=Event(subject='M003', attribute='motion',
constraint='OFF',extraConstraint='None')
condition=State(subject='M001', attribute='motion', constraint='ON')
following_state=State(subject='L003', attribute='lightingMode', constraint='ON')
```

但smartapp channel中没有这个相关性，所以涉及到or和and的规则需要额外补充相关性.但按照原文的算法，smartapp channel生成e2s correlation只可能是不带condition的，所以这块的问题可能只能在未知任何自动化规则时生成。

所以我立马补全了refine部分，这样这种问题就肯定不会出现。

同时还有一个点有问题，这个是由于我之前对python面向对象机制的一个小细节没有注意造成的：

如果没有为类显式定义 `__eq__` 方法，直接比较两个对象使用 `obj1 == obj2`，Python 将默认使用对象的身份进行比较。也就是说，它会检查 `obj1` 和 `obj2` 是否引用内存中的同一个对象。这个过程类似于使用 `is` 操作符，它基于对象的内存地址来判断两个对象是否相同。

示例：

```
class MyClass:
    def __init__(self, value):
        self.value = value

obj1 = MyClass(5)
obj2 = MyClass(5)
obj3 = obj1  # obj3 引用的是 obj1 的对象

# 直接比较两个独立创建的对象
print(obj1 == obj2)  # 输出: False, 因为尽管 obj1 和 obj2 的内容相同，但它们在内存中是不同的对象。

# 比较两个引用相同对象的变量
print(obj1 == obj3)  # 输出: True, 因为 obj3 和 obj1 引用的是同一个对象。
```

在这个例子中，尽管 `obj1` 和 `obj2` 持有相同的数据（都有一个属性 `value`，且值相同），但它们是通过两次独立的构造函数调用创建的，因此它们位于内存中的不同位置。没有为 `MyClass` 定义 `__eq__` 方法，所以 `obj1 == obj2` 的比较结果是基于它们的内存地址，导致比较结果为 `False`。而 `obj1 == obj3` 为 `True`，因为 `obj3` 是 `obj1` 的一个别名，它们引用的是内存中相同的对象。

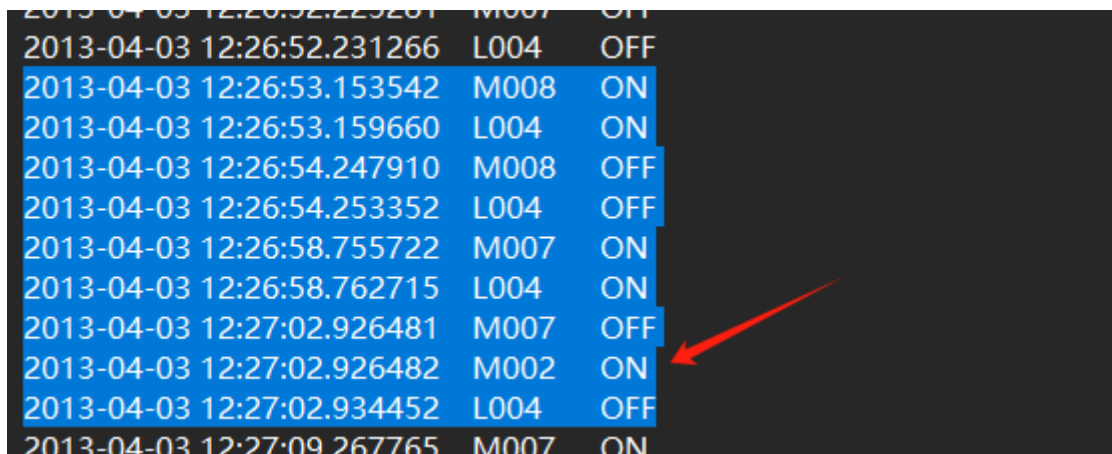
因此需要提前定义 `__eq__`

```
class MyClass:
    def __init__(self, value):
        self.value = value

    def __eq__(self, other):
        return self.value == other.value

obj1 = MyClass(10)
obj2 = MyClass(10)
obj1 == obj2 # 返回True
```

就是后事件，但若前事件的下一条是恶意插入的日志，那么就会造成误报，所以要修改一下算法，一秒之内出现后事件就ok

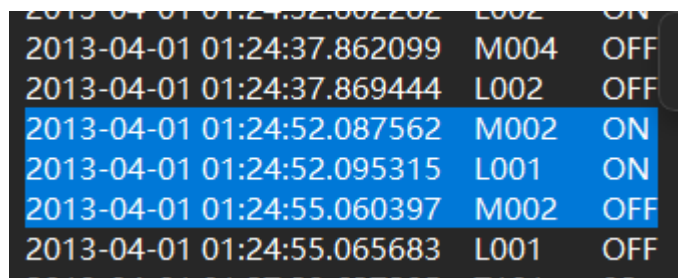


The image shows a log file with entries in the format: YYYY-MM-DD HH:MM:SS.SSSSSS M00X ON/OFF. A red arrow points to the entry: 2013-04-03 12:27:02.926482 M002 ON.

2013-04-03 12:26:52.231266	L004	OFF
2013-04-03 12:26:53.153542	M008	ON
2013-04-03 12:26:53.159660	L004	ON
2013-04-03 12:26:54.247910	M008	OFF
2013-04-03 12:26:54.253352	L004	OFF
2013-04-03 12:26:58.755722	M007	ON
2013-04-03 12:26:58.762715	L004	ON
2013-04-03 12:27:02.926481	M007	OFF
2013-04-03 12:27:02.926482	M002	ON
2013-04-03 12:27:02.934452	L004	OFF
2013-04-03 12:27:09.267765	M007	ON

后面还有很多因为这个造成的误报。

下面这个问题：



The image shows a log file with entries in the format: YYYY-MM-DD HH:MM:SS.SSSSSS M00X ON/OFF. The entries are: 2013-04-01 01:24:37.862099 M004 OFF, 2013-04-01 01:24:37.869444 L002 OFF, 2013-04-01 01:24:52.087562 M002 ON, 2013-04-01 01:24:52.095315 L001 ON, 2013-04-01 01:24:55.060397 M002 OFF, 2013-04-01 01:24:55.065683 L001 OFF.

2013-04-01 01:24:37.862099	M004	OFF
2013-04-01 01:24:37.869444	L002	OFF
2013-04-01 01:24:52.087562	M002	ON
2013-04-01 01:24:52.095315	L001	ON
2013-04-01 01:24:55.060397	M002	OFF
2013-04-01 01:24:55.065683	L001	OFF

违背了相关性

```
'e2e' correlation:
pre_event=Event(subject='L001', attribute='lightingMode',
constraint='ON',extraConstraint='None')
condition=None
following_event=Event(subject='M002', attribute='motion',
constraint='OFF',extraConstraint='None')
```

理由是一秒以内没有发生M002关闭，而是三秒以后。由此可见，只有符合smartapp channel发生的后事件才会在一秒以内发生，而physical channel correlations发生的后事件可能不能不在一秒以内发生。

但如果修改成三秒，也会造成相应的误报，所以干脆直接拿出当前日志的后两条日志检查是否与相关性匹配

至此这一部分的纠错环节就告一段落

接下里分析异常检测结果可得，别的误报原因都已解决，除了一些相关性成功率过高之外，针对该问题只需调高P0即可。

将p0设为0.9999，检测结果为：

```
anomaly_numbers:46
```

此时physical correlation没有一项通过了假设检验

将p0设为0.999，检测结果为：

```
anomaly_numbers:83
```

此时仍有四十多项physical correlation保留

计算准确率，召回率

$$\begin{aligned} Precision &= \frac{True\ Positive}{True\ Positive + False\ Positive} \\ Recall &= \frac{True\ Positive}{True\ Positive + False\ Negative} \\ False\ Alarm\ Rate &= \frac{False\ Positive}{All\ Events} \end{aligned}$$

真阳性：被检测到的插入的异常数目

假阳性：检测到异常但实际并不是异常的数目

假阴性：没有检测到的插入的异常数目

显然 假阴性=插入异常总数 (50) -真阳性

见证奇迹的时刻到了：

![image-20240411163210668](C:\Users\Administrator\AppData\Roaming\Typora\typora-user-

images\image-20240411163210668.png

```
anomaly_numbers:47
true_postive:47
false_postive:0
false_negative:3
precision:1.0
recall:0.94
```

可以看到结果还不错。

之前提到的只有smartapp correlations的情况，上周学长也提到这个数据集并没有考虑physical channel，所以从理论上分析准确率应该更高，召回率的话应该还是有physical correlation的更高一点。

```
anomaly_numbers:46
true_postive:46
false_postive:0
false_negative:4
precision:1.0
recall:0.92
```

```
2013-04-13 10:00:10.201779 L001 OFF
2013-04-13 10:00:10.201780 M002 ON
2013-04-13 10:00:11.482178 M002 ON
2013-04-13 10:00:11.487709 L001 ON
2013-04-13 10:00:12.650669 M002 OFF
```

手动查看可知假阴性的情况全部是因为日志存在两条相同的M002 ON（可能是因为正好插入，也可能是因为日志本身有问题），由于处理逻辑上有一些问题导致没有检测到

4.11

第二种情况：事先未知自动化规则

生成方式

生成的方式很简单，首先根据所有设备和其对应的可能的值，生成所有event、state

```
for device_name in devices:
    #不是温度的情况
    if devices[device_name]['attribute']!='temperature':
        all_events.append(semantic_analysis.Event(device_name
,devices[device_name]['attribute'],devices[device_name]['range'][0]))
        all_events.append(semantic_analysis.Event(device_name,
devices[device_name]['attribute'], devices[device_name]['range'][1]))
        all_states.append(semantic_analysis.Event(device_name,
devices[device_name]['attribute'], devices[device_name]['range'][0]))
```



```

        all_states.append(semantic_analysis.Event(device_name,
devices[device_name]['attribute'], devices[device_name]['range'][1]))
        #是温度的情况
    else:
        for degree in range(20, 36):
            all_events.append(semantic_analysis.Event(device_name,
devices[device_name]['attribute'], '<' + str(degree)))
            all_events.append(semantic_analysis.Event(device_name,
devices[device_name]['attribute'], '>=' + str(degree)))
            all_states.append(semantic_analysis.State(device_name,
devices[device_name]['attribute'], '<' + str(degree)))
            all_states.append(semantic_analysis.State(device_name,
devices[device_name]['attribute'], '>=' + str(degree)))

```

接下来根据生成的event和state，生成所有可能的e2e correlations 和e2s correlations

一条相关性可分为三个属性：trigger、condition、action

- trigger只可能是event
- condition可能有也可能无，若有则是state类型
- action是event或state类型

根据上面的规则可以生成所有可能的相关性

```

#生成所有e2e correlation且带condition
for pre_event in all_events:
    for condition in all_states:
        for followedEvent in all_events:
            new_correlations.append(semantic_analysis.Correlation(pre_event,
condition, followedEvent, 'e2e'))

# 生成所有e2e correlation且不带condition
for pre_event in all_events:
    for followedEvent in all_events:
        new_correlations.append(semantic_analysis.Correlation(pre_event, None,
followedEvent, 'e2e'))

#生成所有e2s correlation且带condition
for pre_event in all_events:
    for condition in all_states:
        for followedState in all_states:
            new_correlations.append(semantic_analysis.Correlation(pre_event,
condition, followedState, 'e2s'))

#生成所有e2s correlation且不带condition
for pre_event in all_events:
    for followedState in all_states:
        new_correlations.append(semantic_analysis.Correlation(pre_event, None,
followedState, 'e2s'))

```

相关函数在 `generate_correlations_without_rules.py`

开始生成相关性，结果生成了几千多万条，大概估算了一下程序要跑三四天才能跑完。。

于是果断放弃生成温度相关性，同时缩小日志数量，加快程序运行速度，达到演示的目的即可。

```
anomaly_numbers:46
true_postive:45
false_postive:1
false_negative:7
precision:0.9782608695652174
recall:0.8653846153846154
```

可以看到同样可以起到效果。

将已知规则和未知规则对比可得，二者召回率和准确率相差无几。通过调参二者都可以达到不错的效果。

从理论分析来看，二者也不应该有很大的差异，甚至在设备很多的情况下，未知规则的效果可能会更好（因为已知规则的在physical channel可能会有遗漏），但代价就是最初生成相关性时非常耗时。

因此其实HAWatcher如果在平台完全闭源的情况下检测也是完全ok的，只是比较耗时。

检验生成的相关性中是否包含smartapp correlations

事先未知自动化规则也就是测试的时候也假设不知道规则，就按假设检验的方法，先假设所有的可能性，再利用事件日志验证。

我们有必要最后和我们设置规则（作为ground truth）做一个比对。

在已知规则的情况下，`smartapp correlations` 共18条，

`physical_and_userActivity_correlations` 共5888条（未经过检验时的数量），我们可以查看他们是否都存在于未知规则时生成的所有相关性中：

```
smartapp correlations:
'e2e' correlation:
pre_event=Event(subject='M004', attribute='motion', constraint='OFF',extraConstraint='None')
condition=None
following_event=Event(subject='L002', attribute='lightingMode', constraint='OFF',extraConstraint='None')
has been found in new_correlations

smartapp correlations:
'e2e' correlation:
pre_event=Event(subject='M001', attribute='motion', constraint='ON',extraConstraint='None')
condition=State(subject='L003', attribute='lightingMode', constraint='OFF')
following_event=Event(subject='L003', attribute='lightingMode', constraint='ON',extraConstraint='None')
has been found in new_correlations

smartapp correlations:
'e2e' correlation:
pre_event=Event(subject='M003', attribute='motion', constraint='ON',extraConstraint='None')
condition=State(subject='L003', attribute='lightingMode', constraint='OFF')
following_event=Event(subject='L003', attribute='lightingMode', constraint='ON',extraConstraint='None')
has been found in new_correlations
```

结果显示所有第一种情况的相关性在未知自动化规则时都生成了（这也是符合常理的）：

```

18 smartapp_correlations.
5888 physical_and_userActivity_correlations.
23188505 new_correlations.
18 smartapp_correlations have been found in new_correlations.
5888 physical_and_userActivity__correlations have been found in new_correlations.

```

#	IF	Then	Generated correlations without rules
1	M002 ON	L001 ON	√
2	M002 OFF	L001 OFF	√
3	M004 ON	L002 ON	√
4	M004 OFF	L002 OFF	√
5	M001 ON or M003 ON	L003 ON	√
6	M001 OFF and M003 OFF	L003 OFF	√
7	M007 ON or M008 ON	L004 ON	√
8	M007 OFF and M008 OFF	L004 OFF	√
9	T101 >= 32 °C	Fan1 ON	√
10	T101 < 31 °C	Fan1 OFF	√
11	T103 >= 25 °C	Fan2 ON	√
12	T103 < 24 °C	Fan2 OFF	√
13	T104 >= 25 °C	Fan3 ON	√
14	T104 < 24 °C	Fan3 OFF	√
15	All physical_and_userActivity_correlations		√

总结

- 这三周下来，收获还是挺多的，第一次完整的复现了一篇论文，让一个系统从头到尾顺下来。最开始对结果并没有报太多的希望，不过没想到结果跑出来还是很不错的。
- 整个复现过程可以大概分为两块：

1.前一块不需要涉及日志审计，主要是correlation的生成等步骤，这时候还是比较有意思的

2.后一块涉及日志审计，主要是假设检验和异常检测部分，这块就会因为很多之前没有注意的问题，导致结果跑出来很难看，而你一时半会还不知道到底问题出在哪。。。这时候就很难受，需要不断地debug和从跑出来的结果中找答案

整个复现过程还是比较耗精力的，尤其是后期在结果有问题时总需要反过来用肉眼从日志数据中寻找问题，后期经历了大量的debug。

- 这篇论文和学长给的数据的格式有些不同，所以有些部分需要自己想办法做相应的调整，同时整个复现过程让我对日志审计方面有了更深入的了解和思考。
- 这篇论文我认为的亮点：
提取语义信息，将规则转化成e2e/e2s correlation以及相关性的生成
- 这篇论文我认为的局限性：、

1.事实上一个e2s correlation可能带condition，但按作者在Semantic Analysis介绍的流程上来看这种correlation是绝对不可能被生成出来的

2.作者运用word2vec模型计算属性之间相似度的流程在我看来并不是百分百的严谨，并且可能会漏掉一些可能的相关性，与其用花哨的方法不如简单粗暴一点，直接默认所有属性相似，最后筛就行了，这样虽然比较耗时，但准确率一定会更高的（设备很多的前提下）

- 收获：

- 1.对日志审计方面有了更深入的了解和思考。

- 2.锻炼了代码能力

- 3.增加了项目经历，体会了从头到尾完成一个项目的流程，可能遇到的问题。

- 4.有些明白为什么有些作者不开源了，如果是我的话，自己写的屎山代码实在是太丑陋了，并且只要不开源，一些可能会被察觉到的问题就不会被察觉。。