Unspurervised Learning

Saquib Herman
saqher20@student.hh.se

November 3, 2021

1 Unspuervised Learning

In the unsupervised learning task, we have chosen to understand the deviation detection in human activities in the smart home environment. For this experiment data from all the 24 deployed sensors were used. We first use the sequence mining method to understand the precedence in activities and then use clustering to understand which activities happen together e.g. what are the rooms user goes to just before the exit. Also, applied the Isolation Forest algorithm to find the anomalies. The number of alorithms implemented is given in Table 1.

2 Enviornment Setup

This portion of the project is implemented entirely on Google Cloud Platform. Below are the configuration details of GCP.

2.1 Google Cloud Platform Setup

Below is the configuration deatils of GCP:

1. 1 Master Node
2. 2 Worker Nodes
3. Machine Type is n1-standard-4

- PySpark Version 3.1.2 is used.

Task Type	**Algorithm**	**Evaluation Criteria**
Clustering	KMeans	Silhoutte Score
Clustering	Bisect Mean	Silhoutte Score
Sequence Mining	PrefixSpan	Support
Anomaly Detection	Isolation Forest	None

Table 1: Implemented Algorithms

Region	europe-north1
Zone	europe-north1-a
Autoscaling	Off
Dataproc Metastore	None
Scheduled deletion	Off
Master node	Standard (1 master, N workers)
Machine type	n1-standard-4
Number of GPUs	0
Primary disk type	pd-standard
Primary disk size	500GB
Local SSDs	0
Worker nodes	2
Machine type	n1-standard-4
Number of GPUs	0
Primary disk type	pd-standard
Primary disk size	500GB
Local SSDs	0
Secondary worker nodes	0
Secure Boot	Disabled
VTPM	Disabled
Integrity Monitoring	Disabled
Cloud Storage staging bucket	dataproc-staging-europe-north1-13551691598-yezv75s5
Network	default
Network tags	None
Internal IP only	No
Image version	2.0.20-debian10
Created	Sep 19, 2021, 9:41:55 PM
Optional components	JUPYTER
Properties	Show properties
Advanced security	Disabled
Labels	goog-datap... : cluster-sm...
Encryption type	Google-managed key

Figure 1: GCP Configurations

- Following Google Cloud Products have been used:
  - Dataproc
  - Compute Engine
  - VPC Netwrok
  - Cloud Storage

3 Data Preparation

The given dataset is divided into three CSV files. To prepare the data for the data mining task, all the data is combined and stored in one parquet file. The reason for using parquet format is, parquet is more efficient in terms of storage and performance. The code snippet shown in Listing 1 is used for creating and storing data frames in parquet format.

```
#Combine the DataFrames into one
sensor_sample = sensor_int.union(sensor_float)
sensor_sample.count()

# Join the DataFrames
sensor_full = sensor_sample.join(sensor, on = "
    sensor_id", how = "inner")

# Save the sensor_full DataFrame in Parquet format
sensor_full.write.parquet('gs://smarthome-326501/
    sensor_full.parquet', mode='overwrite')
```

Listing 1: Data Preparation

4 Exploratory Data Analysis

Exploratory data analysis (EDA) is used to analyze and investigate data sets and summarize their main characteristics. Mainly below tasks have been done as a part of EDA on the whole dataset.

- Done initial analysis on each sensor data.
- Check min and max value of FLOAT sensors.
- Draw histograms

```
from pyspark_dist_explore import hist
import matplotlib.pyplot as plt
bedroom = sensor.filter(sensor.location == "bedroom")
fig, ax = plt.subplots()
hist(ax, bedroom.filter((bedroom.name == 'bedroom/bed/
    pressure') ).select('value'), color=['blue'])
plt.show()
```

Listing 2: Plotting Histogram

- GROUPBY hour and aggregated by value for bed pressure sensor.
- This analysis is done for all sensors.

```
bed_byHour_pandas.plot(x='hour',y='avg(value)',c='
    DarkBlue')
```

Listing 3: Plot Bedroom Bed Pressure By Hour

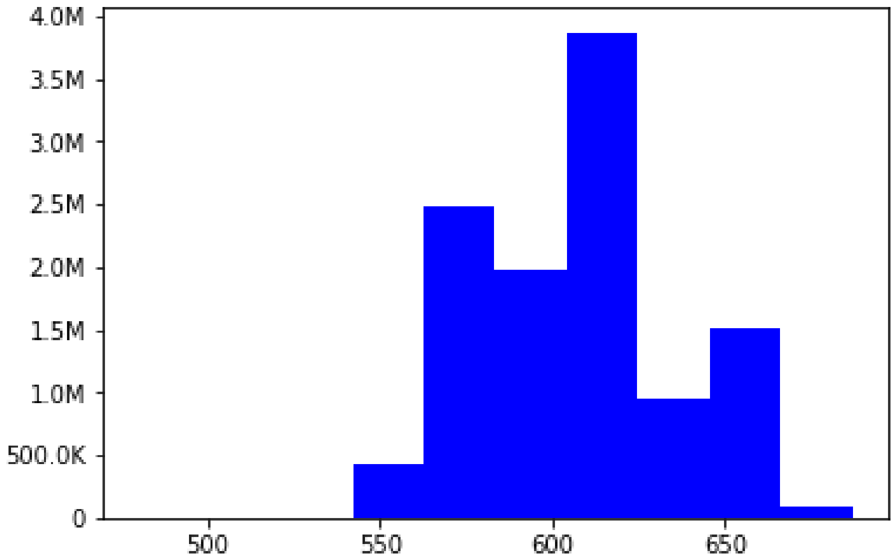

Figure 2: Bedroom Bed Pressure Sensor

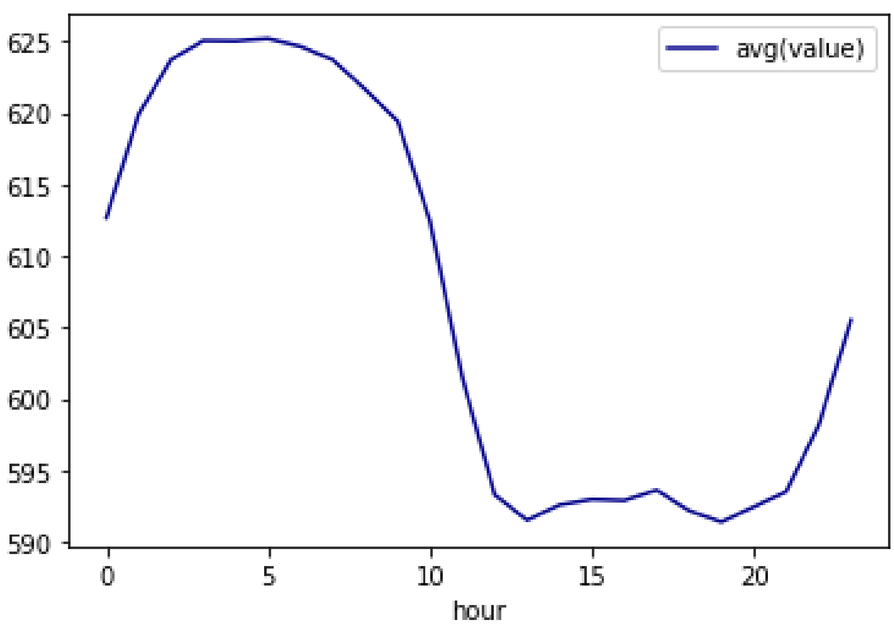

Figure 3: Bedroom Bed Pressure By Hour

Based on the EDA, threshold values of each sensor is obtained as hardcoded in code Listing 4.

```
active_threshold = {
  'balcon/door/contact' : 0, # max:1, min:0

  'bathroom/ambience/humidity' : 50, # max:98.56, min:15.2
  'bathroom/ambience/light' : 100, # max:1024, min:0
  'bathroom/ambience/motion' : 0, # max:1, min:0
  'bathroom/ambience/temperature' : 26, # max:29.23, min:21.13
  'bathroom/washingmachine/current' : 9, # max:141616.557, min:-141616.565

  'bedroom/ambience/motion' : 0, # max:1, min:0
  'bedroom/ambience_under_the_bed/motion' : 0, # max:1, min:0
  'bedroom/bed/pressure' : 600, # max:687, min:480
  'bedroom/weightscale/pressure' : 60, # max:293, min:0

  'corridor/ambience/motion' : 0, # max:1, min:0
  'corridor/ilifeRobot/current' : 9, # max:137744.237, min:0

  'entrance/door/contact' : 0, # max:1, min:0

  'kitchen/ambience/motion' : 0, # max:1, min:0
  'kitchen/coffeemaker/current' : 1, # max:123914.494, min:0
  'kitchen/dishwasher/current' : 10, # max:137744.237, min:0
  'kitchen/fridge/contact' : 0 , # max:1, min:0
  'kitchen/kettle/current' : 9, # max:97361.388, min:0
  'kitchen/microwave/current' : 9, # max:1343.936, min:0
  'kitchen/sandwichmaker/current' : 9, # max:137744.237, min:0
  'kitchen/stove/light' : 100, # max:1024, min:0,

  'livingroom/ambience/motion' : 0, # max:1, min:0
  'livingroom/couch/pressure' : 300, # max:449.0, min:0
  'livingroom/tv/light' : 100 # max:1024, min:3,
}
```

Listing 4: Min-Max and threshold values of each sensor

5 Visualization

5.1 Code Preparation for Visualization

- Given code in the Listing 5 is the data preparation for creating the heatmap visualization.
- Sensor activation time and value is captured for the whole duration.
- This calculation is done for all sensor.
- Later it is used for Sequence Mining also.

```
import random
import seaborn as sns

bed_pressure_start = []
bed_pressure_end = []
x = 0
y = 0
for row in bedroom_bed_pressure.rdd.collect():
y = row.value
if x == 0 and y == 1:
bed_pressure_start.append(row.timestamp)
elif x == 1 and y == 0:
bed_pressure_end.append(row.timestamp)

x = y

bed_pressure_hour = [bed_pressure_start[i].hour for i in range(len(bed_pressure_start))]
bed_pressure_minute = [bed_pressure_start[i].minute for i in range(len(bed_pressure_start))]

bed_pressure_sample_df = pd.DataFrame(list(zip(bed_pressure_hour, bed_pressure_minute)),
columns =['hour', 'minute'])
bed_pressure_sample_df = bed_pressure_sample_df.groupby(bed_pressure_sample_df.columns.tolist()).size().reset_index().\
rename(columns={0:'count'})

```

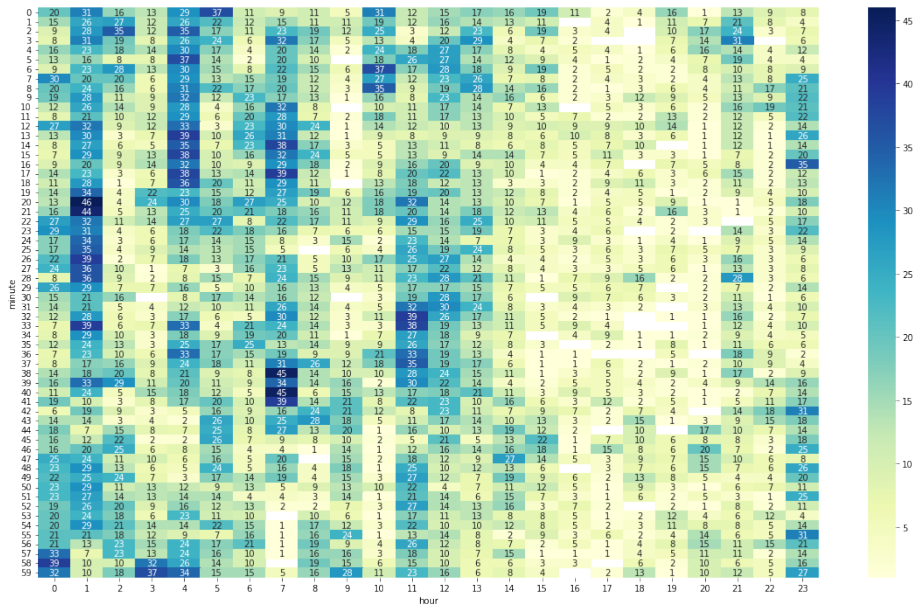

Figure 4: GCP Configurations

```
bed_pressure_sample_df = bed_pressure_sample_df.pivot(
    "minute", "hour", "count")

f, ax = plt.subplots(figsize=(20, 12))
ax = sns.heatmap(bed_pressure_sample_df,annot=True,
    cmap="YlGnBu")
```

Listing 5: Code Preparation for Heatmap Visualization

5.2 Heatmap Visualization

6 Code Restructuring

For better code readability and reusability Class and functions are created. Please refer Listing 6

```
class Smarthome:
  instances = []
  def __init__(self, name):
    self.name = name

  def creare_dataframe(self):
```

```
    self.dataframe = sensor.filter(sensor.name == self.name).select(['timestamp', 'value'])
    self.dataframe = self.dataframe.withColumn("timestamp", func.to_timestamp(self.dataframe.timestamp))

  def head(self, nrows=5):
    return self.dataframe.limit(nrows).toPandas()

......
......
  def start_end_timestamp(self):
    self.start = []
    self.end = []
    flag_one = 0
    flag_two = 0
........
........
........

sensor_name = 'balcon/door/contact'
balcon_door_contact = Smarthome(sensor_name)
process_sensor_data(balcon_door_contact, sensor_name)
balcon_door_contact.display_heatmap()
```

Listing 6: EDA of Bedroom sensors

7 Sequence Mining

For sequence mining, we have used the PrefixSpan method. The way this method works is that it discovers all frequent sequential patterns occurring in a sequence dataset. In order to use the most frequent sequences, we make use of the support parameter which is the number of sequence occurrences in the dataset to the total number of sequences in the dataset. By using a minimum support value we can filter out the less frequent sequences, which is set by us 0.70 for this study.

7.1 Data Preparation for Sequence Mining

- Given code in the Listing 7 is the data preparation for sequence mining.
- For each sensor one dataframe is created.
- Finally all dataframed joined.

```
df1 = pd.DataFrame(balcon_door_contact.start)
df1['sensor_name'] = 'balcon/door/contact'
df1.columns = ['timestamp', 'sensor_name']
df1['timestamp'] = pd.to_datetime(df1['timestamp'])
df2 = pd.DataFrame(bathroom_ambience_humidity.start)
df2['sensor_name'] = 'bathroom/ambience/humidity'
df2.columns = ['timestamp', 'sensor_name']
df2['timestamp'] = pd.to_datetime(df2['timestamp'])
.......
frames = [df1, df2,..]
df=pd.concat(frames)
df = df.sort_values(by='timestamp')
```

Listing 7: Data Preparation for Sequence Mining

7.2 Method

Code snippet Listing 8 shows that PySpark function PrefixSpan helps us to get the frequent sequence pattern. Here "support" is set as 0.70 for pattern length "3". It means association rule for 3 sensors can be found after running this code.

```
from pyspark.ml.fpm import PrefixSpan

spark_df = spark.createDataFrame(df)
df_array = spark_df.select('sequence')
prefixSpan = PrefixSpan(minSupport=0.70,
    maxPatternLength=3)

# Find frequent sequential patterns.
sequence = prefixSpan.findFrequentSequentialPatterns(
    df_array)
```

Listing 8: Data Preparation for Sequence Mining

7.3 Evaluation

After running the code in Listing 9, result it displayed is in Figure 5. Result shows that Bedroom motion sensor and Corridor motion sensors shows a strong association. For sequence mining, there is no evaluation method can be used. We can generate association rules based on support value and given pattern length. Example shows the output for minimum support for 0.70 for pattern length 3.

```
from pyspark.sql.functions import expr, round

```

	sequence	freq	size	percentage
0	[[kitchen/stove/light], [bedroom/ambience/moti...	129	[1, 2]	70.49
1	[[kitchen/stove/light], [bedroom/ambience/moti...	151	[1, 2]	82.51
2	[[kitchen/stove/light], [corridor/ambience/mot...	135	[1, 2]	73.77
3	[[kitchen/stove/light], [corridor/ambience/mot...	147	[1, 2]	80.33
4	[[kitchen/stove/light], [corridor/ambience/mot...	151	[1, 2]	82.51
...	...	...	...	...
166	[[bedroom/ambience/motion], [corridor/ambience...	169	[1, 2]	92.35
167	[[bedroom/ambience/motion], [corridor/ambience...	173	[1, 2]	94.54
168	[[bedroom/ambience/motion], [kitchen/ambience/...	137	[1, 2]	74.86
169	[[bedroom/ambience/motion], [kitchen/ambience/...	171	[1, 2]	93.44
170	[[bedroom/ambience/motion], [kitchen/ambience/...	142	[1, 2]	77.60

171 rows × 4 columns

Figure 5: Result of Sequence Mining

```
#get the size of each array within the arrays
filtered = sequence.withColumn('size', expr('transform
    (sequence, x -> size(x))'))

# Let's also add a column that tells us the percentage
     of the sequences
row_cnt = df_array.count()
filtered = filtered.withColumn('percentage',round((
    func.col("freq")/row_cnt)*100,2))
# Then filter out only the ones with more than 2
    elements
filtered = filtered.where(func.array_contains(filtered
    .size, 2))
filtered.toPandas()
```

Listing 9: Code to see the results of Sequence Mining

8 Clustering

In the clustering method, K-Means and bisecting K-Means are applied on the aggregated data. K-means works on building clusters on the data based on an initial set of randomly selected centroids. The centroids the re-centre closer to denser cluster of points as we go through many iterations of the algorithm. Bisect K-Means is just an alteration of the K-Means algorithm which is a form of hierarchical clustering.

8.1 Data Preparation for Clustering

Follows the below steps to preapre the data for clustering: Please refer Code snippet in Listing 10.

- Pivot the Dataframe .
- Facing difficulty in filling the null values in PySpark so filled it with minimum value for each sensor.
- Since some FLOAT sensor mostly current ones are fluctuating so set it manually after finding threshold as listed in Listing 4
- Created new dataframe for each sensor after Group-by by creating one day window with the aggregation function mean.
- Then joined all dataframes of each sensor.
- Total have 183 rows represents each day, output is hows in Figure 6.
- Each column have mean values for sesnors.

```
joined_df = df1.join(df2, ["window"], how='left')
joined_df = joined_df.join(df3, ["window"], how='left'
    )
joined_df = joined_df.join(df4, ["window"], how='left'
    )
......
......
......
```

Listing 10: Joinind Pivoted DataFrames

8.2 Method

Choosing KMeans and Bisecting KMeans over other methods because of the below reasons:

- Relatively simple to implement.
- Guarantees convergence.
- Generalizes to clusters of different shapes and sizes.

	window	balcon_door_contact	bathroom_ambience_motion	bedroom_ambience_motion	bedroom_ambience_bed_motion	corridor_ambience_motion	entr
0	(2020-03-15 00:00:00, 2020-03-16 00:00:00)	0.000347	0.001065	0.005035	0.000000	0.007824	
1	(2020-04-01 00:00:00, 2020-04-02 00:00:00)	0.000833	0.164770	0.094214	0.000556	0.023773	
2	(2020-03-22 00:00:00, 2020-03-23 00:00:00)	0.009803	0.094502	0.006308	0.000000	0.001551	
	(2020-04-14						

Figure 6: Data Prepared for Clustering

8.3 Evaluation

The evaluation metric used in case of clustering is the silhouette score. The silhouette score gives us information on how the fit of the clustering algorithm is on the dataset. Figure 7 compares both the algorithms based on silhouette score. Silhouette score is choosed over Elbow method because of the two main reasons:

- It is better in measuring **Cohesiveness** within the clusters.
- It is better in measuring **Separation** between the clusters.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig, ax = plt.subplots(1,1, figsize = (8,6))
ax.plot(range(2,kmax),bkmcost[2:kmax],color='blue')
ax.plot(range(2,kmax),kmcost[2:kmax],color='red')
ax.set_xlabel('k')
ax.set_ylabel('cost')
```

Listing 11: Comparing Clustering with Plot

8.4 Results

Clearly Bisecting Kmeans emerged as a winner here with Silhouette squared euclidean distance = 0.3417920068927815 for k=9, where k is the number of clusters.

Centroids obtained by clustering methods: we could use these groups to target similar activity segments. For example if we do some research about the groups and discover that one is mostly a certain pattern and activity frequency, and relate with that to address our business problem. We could also learn a bit more about our clustering by calling on various aggregate statistics for each one of the clusters across each of the variables in our dataframe.

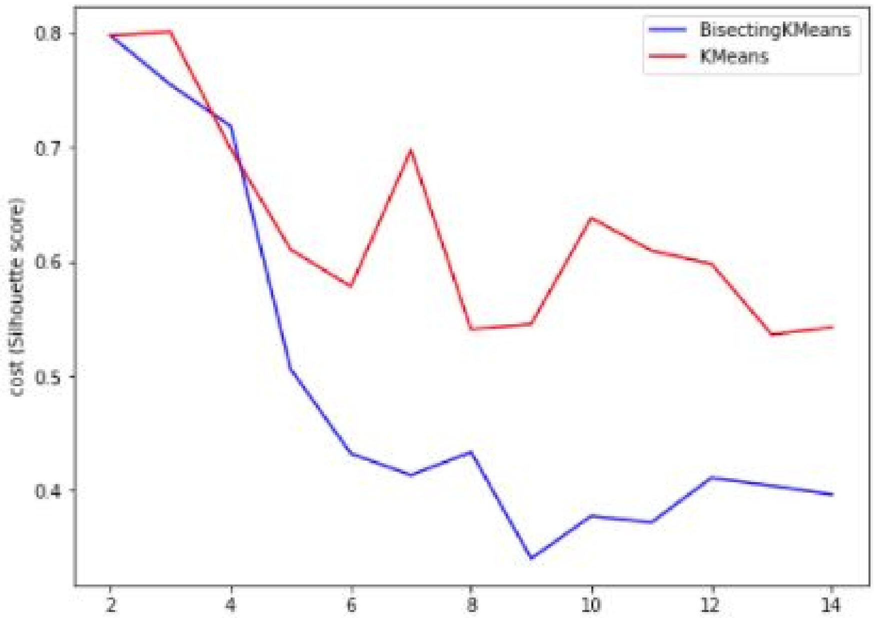

Figure 7: KMeans v/s Bisecting KMeans

9 Anomaly Detection

9.1 Method

The method used here for anomaly detection is the Isolation Forest. Isolation Forests are able to exploit subsampling to achieve a low linear time-complexity and a small memory-requirement, and to deal with the effects of swamping and masking effectively. This gives us better tools to improve our detection rates.

9.2 Evaluation

For Isolation Forest we used 100 number of estimators and contamination value 0.2. Isolation Forest works based on outlier score. Score is calculated for each data point. Outliers is judged based on score. All the anomalies has higher scores. Code snippet shows in Listing 12

9.3 Results

The result after running the Isolation Forest is shown in Figure 8. Total 37 anomalies found in the dataframe, each anomaly represent one day.

```
model=IsolationForest(n_estimators=100,max_samples='
    auto',contamination=float(0.2))
```

	balcon_door_contact	bathroom_ambience_motion	bedroom_ambience_motion	bedroom_ambience_bed_motion	corridor_ambience_motion	entrance_do
5	0.000579	0.000000	0.172884	0.000000	0.146529	
6	0.116910	0.000000	0.009838	0.012998	0.000000	
7	0.000127	0.223866	0.211007	0.000023	0.001829	
32	0.207593	0.000000	0.016169	0.012188	0.100463	
33	0.101481	0.000000	0.010382	0.009398	0.048542	
36	0.000000	0.000000	0.000000	0.000000	0.000000	
43	0.000278	0.000000	0.003206	0.002708	0.004051	
61	0.238172	0.000000	0.002582	0.008800	0.014033	
62	0.000000	0.000000	0.002095	0.004294	0.005914	
67	0.167643	0.084418	0.062156	0.013834	0.006194	
70	0.000000	0.049630	0.154178	0.003356	0.067384	
73	0.000000	0.000000	0.009505	0.004689	0.087424	
75	0.341415	0.000000	0.020951	0.010726	0.032494	
76	0.133125	0.000000	0.003021	0.002257	0.012627	
79	0.005003	0.000000	0.012067	0.005408	0.007817	
81	0.051096	0.000000	0.006312	0.006914	0.010411	
83	0.228206	0.000000	0.004826	0.004155	0.006597	
85	0.340125	0.000000	0.005973	0.016543	0.015964	
89	0.000000	0.000000	0.000000	0.000000	0.000000	
90	0.000000	0.000000	0.000000	0.000184	0.000000	
92	0.137262	0.000000	0.010458	0.008084	0.006856	
101	0.034110	0.000000	0.005475	0.003623	0.141971	
102	0.000000	0.000000	0.150596	0.000213	0.124222	
108	0.001551	0.000000	0.004236	0.003785	0.005544	
109	0.392535	0.000000	0.000748	0.011210	0.022689	
110	0.132901	0.000000	0.004746	0.007085	0.008648	
111	0.000000	0.000000	0.004167	0.013474	0.014377	
117	0.000834	0.000000	0.006159	0.000000	0.029185	
119	0.000081	0.194236	0.121100	0.001748	0.016435	
121	0.094977	0.223148	0.198333	0.005671	0.020093	
123	0.000081	0.000000	0.183073	0.004421	0.159844	
127	0.141528	0.000000	0.028808	0.016875	0.035139	
133	0.409179	0.000000	0.007356	0.014515	0.013229	
135	0.141550	0.000000	0.028391	0.025498	0.047430	
154	0.057752	0.000000	0.004596	0.006656	0.017688	
158	0.237766	0.000000	0.015880	0.011319	0.033229	
174	0.000000	0.005463	0.193472	0.000266	0.138171	

37 rows × 25 columns

Figure 8: Anomalies found with Isolation Forest

```
model.fit(df)

df['scores'] = model.decision_function(df)
df['anomaly_score'] = model.predict(df)

df[df['anomaly_score']==-1]
```

Listing 12: Isolation Forest

10 Supervised Learning based on Sequence Mining

As a Supervised Learning task, we're trying to predict whether resident about **to leave** the house or going to **stay back**. Frequent Pattern find in the Sequence Mining result could be used to create new labels.

10.1 Data Preparation for Supervised Learning

Below steps are execute for creating the dataframe for Supervised Learning

- Pivot the dataframe with aggregation function "sum" for every second.
- Fill the null values.
- Call User defined function for creating new labels called "occupied". Please refer the code in Listing 13.

```
old_value = 0
counter = 0
status = 1

def getStatus(new_value):
  global old_value
  global counter
  global status

  if counter != 0:
    counter = counter - 1
    old_value = new_value
    return status

  if old_value != new_value:
    status = 1 - status
    counter = 30

  old_value = new_value

  return status

# Define the method as a UDF
udfOccupancy = udf(getStatus)

# Create a new column using your UDF
pivotDF = pivotDF.withColumn('occupied',
 udfOccupancy(pivotDF["entrance/door/contact"]))
```

...wave/current	kitchen/sandwichmaker/current	kitchen/stove/light	livingroom/ambience/motion	livingroom/couch/pressure	livingroom/tv/light	ts_day	occupied
0	0	1024.0	0.0	268.0	1024.0	2020-03-01	1
0	0	1024.0	0.0	268.0	1024.0	2020-03-01	1
0	0	1024.0	0.0	0.0	1024.0	2020-03-01	1
0	0	1024.0	0.0	268.0	1024.0	2020-03-01	1
0	0	1024.0	0.0	268.0	1024.0	2020-03-01	1

Figure 9: Creating new label "occupied" or "home-or-away"

```

pivotDF = pivotDF.withColumn('home_or_away',(when(
 pivotDF["occupied"] == 1, "home").otherwise("away")
 ))
```

Listing 13: Creating New Label

```
# Read in functions we will need
from pyspark.ml.feature import VectorAssembler
from pyspark.sql.types import *
from pyspark.sql.functions import *
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import MinMaxScaler

input_columns = df.columns # Collect the column
 names as a list
input_columns = input_columns[1:-1] # keep only
 relevant columns: from column 1 to
dependent_var = 'home_or_away'
```

Listing 14: Supervised Learning on Labelled Dataframe

- **Not able to proceed from here**. Please refer Figure 10

```
# change label (class variable) to string type to prep for reindexing
# Pyspark is expecting a zero indexed integer for the label column.
# Just in case our data is not in that format... we will treat it by using the StringIndexer built in method
renamed = df.withColumn("label_str", df[dependent_var].cast(StringType())) #Rename and change to string type
indexer = StringIndexer(inputCol="label_str", outputCol="label") #Pyspark is expecting the this naming convention
indexed = indexer.fit(renamed).transform(renamed)

[Stage 43:>                                                          (0 + 8) / 9]
```

Figure 10: Needs PySpark Performance Tuning

- **StringIndexer function of PySpark halts the progress**.

It seems PySpark performance tuning is required to fix this issue.