wtphm

Jun 03, 2020

User Guide

The **W**ind **T**urbine **P**rognostics and **H**ealth **M**anagement library processes wind turbine events (also called alarms or status) data, as well as operational SCADA data (the usually 10-minute data coming off of wind turbines) for easier fault detection, prognostics or reliability research.

Turbine alarms often appear in high numbers during fault events, and significant effort can be involved in processing these alarms in order to find what actually happened, what the root cause was, and when the turbine came back online. This module solves this by automatically identifying stoppages and fault periods in the data and assigning a high-level "stoppage category" to each. It also provides functionality to use this info to label SCADA data for training predictive maintenance algorithms.

Although there are commercial packages that can perform this task, this library aims to be an open-source alternative for use by the research community.

Please reference this repo if used in any research. Any bugs, questions or feature requests can be raised on GitHub. Can also reach me on twitter @leahykev.

CHAPTER 1

Installation

Install using pip!

```
pip install wtphm
```

CHAPTER 2

Is my Data Compatible?

The data manipulated in this library are turbine events/status/alarms data and 10-minute operational SCADA data. They must be in the formats described below.

2.1 Event Data

The `event_data` is related to any fault or information messages generated by the turbine. This is instantaneous, and records information like faults that have occurred, or status messages like low- or no- wind, or turbine shutting down due to storm winds.

The data must have the following column headers and information available:

- `turbine_num`: The turbine the data applies to
- `code`: There are a set list of events which can occur on the turbine. Each one of these has an event code
- `description`: Each event code also has an associated description
- `time_on`: The start time of the event
- `stop_cat`: This is a category for events which cause the turbine to come to a stop. It could be the functional location of where in the turbine the event originated (e.g. pitch system), a category for grid-related events, that the turbine is down for testing or maintenance, in curtailment due to shadow flicker, etc.
- In addition, there must be a specific event `code` which signifies return to normal operation after any downtime or abnormal operating period.

2.2 SCADA/Operational data

The `scada_data` is typically recorded in 10-minute intervals and has attributes like average power output, maximum, minimum and average windspeeds, etc. over the previous 10-minute period.

For the purposes of this library, it must have the following column headers and data:

- `turbine_num`: The turbine the data applies to
- `time`: The 10-minute period the data belongs to
- availability counters: Some of the functions for giving the batches a stop category rely on availability counters. These are sometimes stored as part of scada data, and sometimes in separate availability data. They count the portion of time the turbine was in some mode of operation in each 10-minute period, for availability calculations. For example, maintenance time, fault time, etc. In order to be used in this library, the availability counters are assumed to range between 0 and n in each period, where n is some arbitrary maximum (typically 600, for the 600 seconds in the 10-minute period).

2.2.1 Overview

The three main parts of the package are the *wtphm.batch*, and *wtphm.pred_processing* modules, as well as the *wtphm.clustering* subpackage.

wtphm.batch contains the functions for creating the batches of turbine alarms and assigning a high-level reason for the stoppage, gleaned from the events and scada data. More information on this can be found in[1].

wtphm.pred_processing contains functions for labelling SCADA data based on the batches, for purposes of fault detection or prognosis.

wtphm.clustering deals with clustering together different similar alarm sequences, explored in[2]. This part of the library isn't updated as much as the others - development may be needed in some parts.

Information and critiques on some of the issues surrounding the data coming from wind turbines more generally can be found in[3].

2.2.2 Input Data Needed for Batch Creation

Batches are groups of turbine events generated by the alarm system which appear during a stoppage. The data to be used for creating the batches and assigning a high level cause for the stop must have certain features, described here.

Events Data

The `event_data` is related to any fault or information messages generated by the turbine. This is instantaneous, and records information like faults that have occurred, or status messages like low- or no- wind, or turbine shutting down due to storm winds.

The data must have the following column headers and information available:

- `turbine_num`: The turbine the data applies to
- `code`: There are a set list of events which can occur on the turbine. Each one of these has an event code
- `description`: Each event code also has an associated description
- `time_on`: The start time of the event
- `stop_cat`: This is a category for events which cause the turbine to come to a stop. It could be the functional location of where in the turbine the event originated (e.g. pitch system), a category for grid-related events, that the turbine is down for testing or maintenance, in curtailment due to shadow flicker, etc.

[1] Leahy, K., Gallagher, C., O'Donovan, P., Bruton, K. & O'Sullivan, D. T. (2018), 'A Robust Prescriptive Framework and Performance Metric for Diagnosing and Predicting Wind Turbine Faults based on SCADA and Alarms Data with Case Study', Energies 11(7), pp. 1–21.

[2] Leahy, K., Gallagher, C., O'Donovan, P., & O'Sullivan, D. T. J. (2019), 'Issues with Data Quality for Wind Turbine Condition Monitoring and Reliability Analyses', Energies, 12(2):201; https://doi.org/10.3390/en12020201

[3] Leahy, K., Gallagher, C., O'Donovan, P. & O'Sullivan, D. T. (2018), 'Cluster analysis of wind turbine alarms for characterising and classifying stoppages', IET Renewable Power Generation 12(10), 1146–1154.

- In addition, there must be a specific event `code` which signifies return to normal operation after any downtime or abnormal operating period.

SCADA Data

The `scada_data` is typically recorded in 10-minute intervals and has attributes like average power output, maximum, minimum and average windspeeds, etc. over the previous 10-minute period.

For the purposes of this library, it must have the following column headers and data:

- `turbine_num`: The turbine the data applies to
- `time`: The 10-minute period the data belongs to
- availability counters: Some of the functions for giving the batches a stop category rely on availability counters. These are sometimes stored as part of scada data, and sometimes in separate availability data. They count the portion of time the turbine was in some mode of operation in each 10-minute period, for availability calculations. For example, maintenance time, fault time, etc. In order to be used in this library, the availability counters are assumed to range between 0 and *n* in each period, where *n* is some arbitrary maximum (typically 600, for the 600 seconds in the 10-minute period).

Sample Data

There is sample data provided in the examples folder of the github repository. This is 2 months' of real data for 2 turbines, but has been fully anonymised. For the `event_data`, all codes have been mapped to a random set of numbers, and descriptions have been removed. For the `scada_data`, all values have been normalised between 0 and 1, with the exception of the availability counters ("lot", "rt", etc.) and features counting the number and duration of alarms in the past 48 hours, "num_48h" and "dur_48h").

The data can be imported like so:

```
>>> import wtphm
>>> import pandas as pd
>>> event_data = pd.read_csv('examples/event_data.csv',
...                          parse_dates=['time_on', 'time_off'])
>>> event_data.duration = pd.to_timedelta(event_data.duration)
>>> scada_data  = pd.read_csv('examples/scada_data.csv',
...                           parse_dates=['time'])
```

```
>>> event_data.head()
   turbine_num  code             time_on            time_off        duration  stop_
↪cat            description
0           22     9 2015-11-01 00:03:56 2015-11-01 00:23:56 0 days 00:20:00      ␣
↪ok  description anonymised
1           21    93 2015-11-01 00:09:54 2015-11-01 00:10:56 0 days 00:01:02      ␣
↪ok  description anonymised
2           21    97 2015-11-01 00:10:56 2015-11-01 00:37:39 0 days 00:26:43      ␣
↪ok  description anonymised
3           22   165 2015-11-01 00:16:39 2015-11-06 05:03:35 5 days 04:46:56      ␣
↪ok  description anonymised
4           22    93 2015-11-01 00:23:56 2015-11-01 00:24:58 0 days 00:01:02      ␣
↪ok  description anonymised
```

```
>>> scada_data.head()
                 time  turbine_num  wind_speed        kw  wind_speed_sd  wind_speed_
↪max  ...  lot  wot  est   mt   rt  eect
```

(continues on next page)

(continued from previous page)

```
0 2015-11-01 00:00:00          22    0.148473  0.009655       0.064693        0.
↪110283  ...  0.0  0.0  0.0  0.0  0.0   0.0
1 2015-11-01 00:10:00          22    0.125081  0.004962       0.066886        0.
↪084016  ...  0.0  0.0  0.0  0.0  0.0   0.0
2 2015-11-01 00:20:00          22    0.121183  0.004913       0.060307        0.
↪086624  ...  0.0  0.0  0.0  0.0  0.0   0.0
3 2015-11-01 00:30:00          22    0.137752  0.004454       0.067982        0.
↪104322  ...  0.0  0.0  0.0  0.0  0.0   0.0
4 2015-11-01 00:40:00          22    0.171540  0.040889       0.066886        0.
↪113077  ...  0.0  0.0  0.0  0.0  0.0   0.0
[5 rows x 22 columns]
```

2.2.3 Group Faults of the Same Type

Sometimes it is useful to treat similar types of fault together as the same type of fault. An example would be faults across different pitch motors on different turbine blades being grouped as the same type of fault. This is useful, as there are typically *very* few fault samples on wind turbines, so treating these as three separate types of faults would give even fewer samples for each class.

The `wtphm.batch.get_grouped_event_data()` function does this. For the pitch fault example above, the grouping would give the events "pitch thyristor 1 fault" with code 501, "pitch thyristor 2 fault" with code 502 and "pitch thyristor 3 fault" with code 503 all the same event description and code, i.e. they all become "pitch thyristor 1/2/3 fault (original codes 501/502/503)" with code 501. Note that this is an entirely optional step before creating the batches of events.

```
>>> # codes that cause the turbine to come to a stop
... stop_codes = event_data[
...     (event_data.stop_cat.isin(['maintenance', 'test', 'sensor', 'grid'])) |
...     (event_data.stop_cat.str.contains('fault'))].code.unique()
>>> # each of these lists represents a set of pitch-related events, where
... # each memeber of the set represents the same event but along a
... # different blade axis
... pitch_code_groups = [[300, 301, 302], [400, 401], [500, 501, 502],
...                      [600, 601], [700, 701, 702]]
>>> event_data[event_data.code.isin(
...     [i for s in pitch_code_groups for i in s])].head()
     turbine_num  code             time_on            time_off duration  stop_cat    ␣
↪                    description
112           22   502 2015-11-01 21:04:26 2015-11-01 21:04:36 00:00:10  fault_pt ␣
↪description anonymised pitch axis 3
114           22   601 2015-11-01 21:04:28 2015-11-01 21:04:36 00:00:08  fault_pt ␣
↪description anonymised pitch axis 2
119           22   601 2015-11-01 21:04:36 2015-11-01 21:04:36 00:00:00  fault_pt ␣
↪description anonymised pitch axis 2
131           22   600 2015-11-01 21:04:36 2015-11-01 21:04:36 00:00:00  fault_pt ␣
↪description anonymised pitch axis 1
132           22   600 2015-11-01 21:04:36 2015-11-01 21:04:36 00:00:00  fault_pt ␣
↪description anonymised pitch axis 1
```

As can be seen, the events data has a number of different codes for data along different pitch axes.

Below, we group these together as the same code (note the descriptions have been anonymised):

```
>>> event_data, stop_codes = wtphm.batch.get_grouped_event_data(
...     event_data=event_data, code_groups=pitch_code_groups,
```

(continues on next page)

(continued from previous page)

```
...     fault_codes=stop_codes)
>>> # viewing the now-grouped events from above:
... event_data.loc[[112, 114, 119, 131, 132]]
     turbine_num  code               time_on            time_off duration  stop_cat  ␣
↪                                                          description
112           22   500 2015-11-01 21:04:26 2015-11-01 21:04:36 00:00:10  fault_pt ␣
↪description anonymised pitch axis 1/2/3 (original codes 500/501/502)
114           22   600 2015-11-01 21:04:28 2015-11-01 21:04:36 00:00:08  fault_pt  ␣
↪   description anonymised pitch axis 1/2 (original codes 600/601)
119           22   600 2015-11-01 21:04:36 2015-11-01 21:04:36 00:00:00  fault_pt  ␣
↪   description anonymised pitch axis 1/2 (original codes 600/601)
131           22   600 2015-11-01 21:04:36 2015-11-01 21:04:36 00:00:00  fault_pt  ␣
↪   description anonymised pitch axis 1/2 (original codes 600/601)
132           22   600 2015-11-01 21:04:36 2015-11-01 21:04:36 00:00:00  fault_pt  ␣
↪   description anonymised pitch axis 1/2 (original codes 600/601)
```

2.2.4 Creating Batches

As mentioned in[4], turbine alarms often occur in "showers" which can overwhelm operators and make it difficult to pinpoint the root cause of a stoppage. *wtphm.batch.get_batch_data()* creates the batches. The algorithm is as follows, as described in detail in[1]:

- A list of event codes which causes the turbine to stop, `fault_codes`, are passed to the function, as well as the code which signifies the turbine returning to normal operation after downtime, `ok_code`.
- The earliest event in the `event_data` which matches a code in `fault_codes` is gotten. Every event between then and the next earliest `ok_code` event are stored as a batch.
- The next earliest event in `event_data` which matches a code in `fault_codes` is gotten. Every event between then and the next earliest `ok_code` event are stored as a batch, etc.

```
>>> # create the batches
... batch_data = wtphm.batch.get_batch_data(
...     event_data=event_data, fault_codes=stop_codes, ok_code=207,
...     t_sep_lim='1 hours')
>>> batch_data.loc[15:20]
    turbine_num  fault_root_codes     all_root_codes         start_time  ... fault_
↪dur down_dur     fault_event_ids      all_event_ids
15           22     (68, 113, 500)     (68, 113, 500) 2015-12-03 12:1...  ... ␣
↪00:03:20 00:07:25  Int64Index([29...  Int64Index([29...
16           22         (144, 500)         (144, 500) 2015-12-08 16:3...  ... ␣
↪00:11:50 00:15:55  Int64Index([32...  Int64Index([32...
17           22              (73,)          (73, 141) 2015-12-10 18:1...  ... ␣
↪00:00:00 00:00:17  Int64Index([33...  Int64Index([33...
18           22      (77, 85, 164)      (77, 85, 164) 2015-12-11 10:0...  ... ␣
↪03:03:14 03:07:25  Int64Index([34...  Int64Index([34...
19           22      (77, 85, 164)      (77, 85, 164) 2015-12-14 12:3...  ... ␣
↪09:52:49 09:52:51  Int64Index([36...  Int64Index([36...
20           22   (68, 113, 144,...   (68, 113, 144,... 2015-12-16 10:0...  ... ␣
↪01:09:01 01:09:02  Int64Index([38...  Int64Index([38...
[6 rows x 10 columns]
```

Note that if two stoppages occur in quick succession, i.e. one batch ends and another quickly begins, the `t_sep_lim` argument in *wtphm.batch.get_batch_data()* allows us to treat both as the same continuous batch. For more

[4] Qiu, Y., Feng, Y., Tavner, P., Richardson, P., Erdos, G. & Chen, B. (2012), 'Wind turbine SCADA alarm analysis for improving reliability', Wind Energy 15(8), 951–966.

information about the various columns and parameters, see the `wtphm.batch.get_batch_data()` documentation.

Below, we view one of the batches and the event behind it in a bit more detail:

```
>>> batch_data.loc[20]
turbine_num                              22
fault_root_codes         (68, 113, 144, 500)
all_root_codes           (68, 113, 144, 500)
start_time               2015-12-16 10:00:05
fault_end_time           2015-12-16 11:09:06
down_end_time            2015-12-16 11:09:07
fault_dur                    0 days 01:09:01
down_dur                     0 days 01:09:02
fault_event_ids         Int64Index([3868, 386...
all_event_ids           Int64Index([3868, 386...
Name: 20, dtype: object
>>> event_data.loc[batch_data.loc[20, 'all_event_ids']].head()
      turbine_num  code                time_on             time_off duration  stop_cat    ␣
↪                                                            description
3868           22   144 2015-12-16 10:00:05 2015-12-16 10:00:13 00:00:08  fault_pt    ␣
↪                                                  description anonymised
3867           22    68 2015-12-16 10:00:05 2015-12-16 10:00:13 00:00:08  fault_pt    ␣
↪                                                  description anonymised
3866           22   500 2015-12-16 10:00:05 2015-12-16 10:00:13 00:00:08  fault_pt ␣
↪description anonymised pitch axis 1/2/3 (original codes 500/501/502)
3865           22   113 2015-12-16 10:00:05 2015-12-16 10:00:13 00:00:08  fault_pt    ␣
↪                                                  description anonymised
3869           22   300 2015-12-16 10:00:10 2015-12-16 10:00:13 00:00:03  fault_pt ␣
↪description anonymised pitch axis 1/2/3 (original codes 300/301/302)
```

We can also see the corresponding SCADA data. Note that the down-time counter, 'dt', which count the number of seconds in each 10-minute period the turbine was down, is active after the start time of the batch, and goes back to zero after it reactivates.

```
>>> start = batch_data.loc[20, 'start_time'] - pd.Timedelta('20 minutes')
>>> end = batch_data.loc[20, 'down_end_time'] + pd.Timedelta('20 minutes')
>>> t = batch_data.loc[20, 'turbine_num']
>>> scada_data.loc[
...     (scada_data.time >= start) & (scada_data.time <= end) &
...     (scada_data.turbine_num == t),
...     ['time', 'turbine_num', 'wind_speed', 'kw', 'ot', 'sot', 'dt']]
                    time  turbine_num  wind_speed        kw     ot     sot     dt
6425 2015-12-16 09:50:00           22    0.245289  0.298111  600.0  600.0    0.0
6426 2015-12-16 10:00:00           22    0.281027  0.454494  600.0  600.0    0.0
6427 2015-12-16 10:10:00           22    0.263158  0.016645   11.0    6.0  594.0
6428 2015-12-16 10:20:00           22    0.226446  0.005421    0.0    0.0  600.0
6429 2015-12-16 10:30:00           22    0.217674  0.004993    0.0    0.0  600.0
6430 2015-12-16 10:40:00           22    0.195257  0.004906    0.0    0.0  600.0
6431 2015-12-16 10:50:00           22    0.179337  0.005240    0.0    0.0  600.0
6432 2015-12-16 11:00:00           22    0.234243  0.004948    0.0    0.0  600.0
6433 2015-12-16 11:10:00           22    0.246589  0.005344    0.0   53.0  547.0
6434 2015-12-16 11:20:00           22    0.258285  0.285964  355.0  600.0    0.0
```

2.2.5 Assigning High-Level Root Causes to Stoppages

Once the batches have been obtained, the `event_data` and `scada_data` can be used to assign a "stop category" to the batch. Here the "stop category" refers to a functional location on the turbine using some pre-determined taxonomy, or that the turbine was down due to grid issues, testing, maintenance, etc.

This library provides a family of functions that use two main sources of information to get the stop categories: the "root" events of a batch, and the SCADA data availability counters.

Using the root events

The root events refer to the event(s) that occur at the start of the batch, and are stored as `fault_root_codes` in the `event_data`. Since these are the events that initially cause the turbine to stop, the `stop_cat` of these events are used to assign a `stop_cat` to the batch, i.e. the entire stoppage, as a whole.

To get the `root_cats`, use the *`wtphm.batch.get_root_cats()`* function:

```
>>> root_cats = wtphm.batch.get_root_cats(batch_data, event_data)
>>> root_cats.loc[15:20]
15             (fault_pt, fault_pt, fault_pt)
16                       (fault_pt, fault_pt)
17                                  (sensor,)
18                     (grid, grid, fault_pt)
19                     (grid, grid, fault_pt)
20   (fault_pt, fault_pt, fault_pt, fault_pt)
```

The names of the categories in `root_cats` come from the `stop_cat` of the events from which they are made. Here, "fault_pt" refers to a pitch fault.

From here, we can assign a category to a batch if every member of the `root_cats` is the same, for example "fault_pt":

```
>>> all_pt_ids = wtphm.batch.get_cat_all_ids(root_cats, 'fault_pt')
>>> batch_data.loc[all_pt_ids, 'batch_cat'] = 'fault_pt'
>>> # note the entries compared to above
... batch_data.loc[15:20, 'batch_cat']
15    fault_pt
16    fault_pt
17         NaN
18         NaN
19         NaN
20    fault_pt
Name: batch_cat, dtype: object
```

Or, assign a category if just a single `stop_cat` appears in the `root_cats`. This is useful for if, e.g., we know that an appearance of a grid fault anywhere in the `root_cats` is indicative of a grid fault having taken place:

```
>>> grid_ids = wtphm.batch.get_cat_present_ids(root_cats, 'grid')
>>> batch_data.loc[grid_ids, 'batch_cat'] = 'grid'
>>> batch_data.loc[15:20, 'batch_cat']
15    fault_pt
16    fault_pt
17         NaN
18        grid
19        grid
20    fault_pt
Name: batch_cat, dtype: object
```

The most common `root_cat` in a batch can also be used to label:

```
>>> root_cats.loc[[5, 57, 62]]
5     (grid, grid, fault_pt)
57          (grid, fault_pt)
62      (fault_pt, fault_pt)
Name: fault_root_codes, dtype: object
>>> most_common_cats = wtphm.batch.get_most_common_cats(root_cats)
>>> most_common_cats.loc[[5, 57, 62]]
5                grid
57    grid, fault_pt
62          fault_pt
Name: fault_root_codes, dtype: object
```

Note that entries with a tied "most common" category will be labelled as both.

Using the Availability Counters

In 10-minute SCADA data there are often counters for when the turbine was in various different states, for calculating contractual availability. In a lot of cases, these count the number of seconds in each 10-minute period the turbine was in a certain availability state.

Below, we mark batches as "maintenance" any time the maintenance counter in the corresponding 10-minute SCADA data was active for more than 60 seconds over the duration of the batch. The counter here is represented by the 'mt' column of the SCADA data.

```
>>> maint_ids = wtphm.batch.get_counter_active_ids(
...     batch_data=batch_data, scada_data=scada_data, counter_col='mt',
...     counter_val=60)
>>> batch_data.loc[
...     maint_ids,
...     ['turbine_num', 'fault_root_codes', 'start_time', 'down_end_time']]
    turbine_num fault_root_codes          start_time       down_end_time
55           21            (16,) 2015-12-10 21:59:33 2015-12-11 13:44:37
```

Combining the Labelling Methods

In[1], a combination of the above is described to label the stoppages. This combination is available in *`wtphm.batch.get_batch_stop_cats()`*. From the documentation for that function:

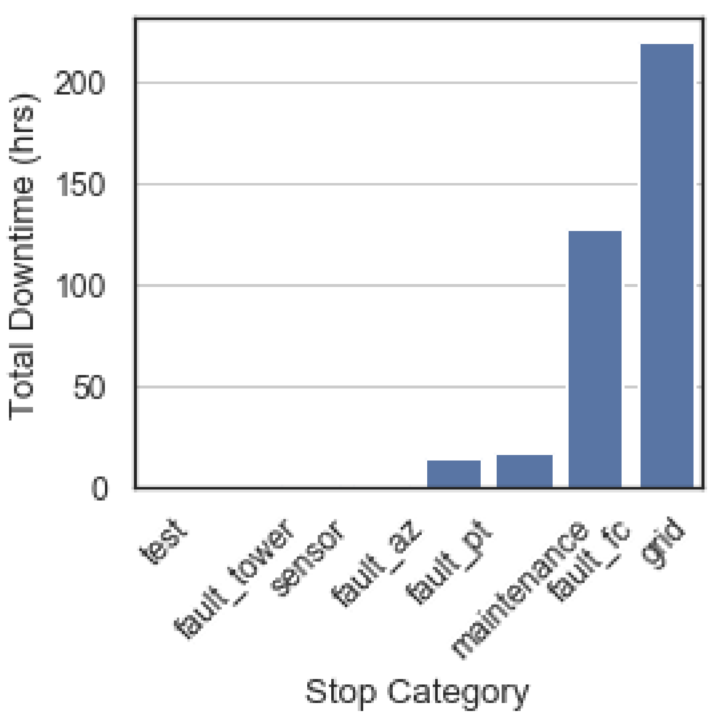
> Labels the batches with an assumed stop category, based on the stop categories of the root event(s) which triggered them, i.e. the one or more events occurring simultaneously which caused the turbine to stop (items lower down supersede those higher up):
>
> - If **all** root events in the batch are "normal" events, then the batch is labelled normal
> - Otherwise, label as the most common stop cat in the initial events
> - If a single sensor category event is present, label sensor
> - If a single grid category event is present, label grid. Also label grid if the grid counter was active in the scada data. This is a timer indicating how long the turbine was down due to grid issues, used for calculating contract availability
> - If the maintenance counter was active in the scada data, label maint
> - There is an additional column labelled "repair". If the repair counter was active, the turbine was brought down for repairs, and this is given the value "TRUE" for these times

```
>>> batch_data = wtphm.batches.get_batch_stop_cats(
...     batch_data, event_data, scada_data, grid_col='lot', maint_col='mt',
...     rep_col='rt')
>>> batch_data.batch_cat
0         fault_pt
1         fault_pt
2             test
3         fault_pt
4         fault_pt
Name: batch_cat, Length: 71, dtype: object
```

2.2.6 Analysing Stoppages

Getting the batch data allows for more complex analysis. Below, the total duration of every stop category in the batches is plotted:

```
>>> durations = batch_data.groupby(
...     'batch_cat').down_dur.sum().reset_index().sort_values(by='down_dur')
>>> durations.down_dur = durations.down_dur.apply(
...     lambda x: x / np.timedelta64(1, 'h'))
>>> sns.set(font_scale=1.2)
>>> sns.set_style('white')
>>> fig, ax = plt.subplots(figsize=(4, 3))
>>> g = sns.barplot(data=durations, x='batch_cat', y='down_dur', ax=ax,
...                 color=sns.color_palette()[0])
>>> g.set_xticklabels(g.get_xticklabels(), rotation=40)
>>> ax.set(xlabel='Stop Category', ylabel='Total Downtime (hrs)')
>>> ax.yaxis.grid()
```



2.2.7 Labelling the SCADA data

Once the stoppages have been identified, the data can be labelled for prognosis or other analysis. This is achieved in the *wtphm.pred_processing* module.

The `wtpum.pred_processing.label_stoppages()` function provides a number of ways of labelling the `scada_data`. For example, suppose we want to label the some specific stoppages in the SCADA data:

```
>>> fault_batches = batch_data.loc[[20, 21]]
>>> fault_batches[
...     ['turbine_num', 'fault_root_codes', 'start_time', 'down_end_time',
...      'down_dur', 'repair']]
    turbine_num      fault_root_codes           start_time       down_end_time down_dur␣
↪ repair
20           22  (68, 113, 144, 500) 2015-12-16 10:00:05 2015-12-16 11:09:07 01:09:02␣
↪  False
21           22            (144, 500) 2015-12-16 15:03:28 2015-12-16 15:25:42 00:22:14␣
↪  False
>>>
>>> scada_l = wtphm.pred_processing.label_stoppages(
...     scada_data, fault_batches, drop_fault_batches=False,
...     label_pre_stop=False)
>>> start = fault_batches.start_time.min() - pd.Timedelta('30T')
>>> end = fault_batches.down_end_time.max() + pd.Timedelta('30T')
>>> s_cols = ['time', 'turbine_num', 'stoppage', 'pre_stop', 'batch_id']
>>> scada_l.loc[(scada_l.time >= start) & (scada_l.time <= end) &
...             (scada_l.turbine_num == 22), s_cols]
                    time  turbine_num  stoppage  batch_id
6424 2015-12-16 09:40:00           22         0        -1
6425 2015-12-16 09:50:00           22         0        -1
6426 2015-12-16 10:00:00           22         0        -1
6427 2015-12-16 10:10:00           22         1        20
6428 2015-12-16 10:20:00           22         1        20
6429 2015-12-16 10:30:00           22         1        20
6430 2015-12-16 10:40:00           22         1        20
6431 2015-12-16 10:50:00           22         1        20
6432 2015-12-16 11:00:00           22         1        20
6433 2015-12-16 11:10:00           22         1        20
6434 2015-12-16 11:20:00           22         0        -1
6435 2015-12-16 11:30:00           22         0        -1
6436 2015-12-16 11:40:00           22         0        -1
6437 2015-12-16 11:50:00           22         0        -1
...                  ...          ...       ...       ...
6455 2015-12-16 14:50:00           22         0        -1
6456 2015-12-16 15:00:00           22         0        -1
6457 2015-12-16 15:10:00           22         1        21
6458 2015-12-16 15:20:00           22         1        21
6459 2015-12-16 15:30:00           22         1        21
6460 2015-12-16 15:40:00           22         0        -1
6461 2015-12-16 15:50:00           22         0        -1
```

In addition, the times leading up to the stoppages can be labelled in the scada data, and the times during the stoppages themselves removed. This is useful for identifying "pre-stop" periods. Here, the times between 30 minutes before and 10 minutes before a fault are labelled as "pre-stop" periods.

```
>>> scada_l = wtphm.pred_processing.label_stoppages(
...     scada_data, fault_batches, drop_fault_batches=True,
...     label_pre_stop=True, pre_stop_lims=['30 minutes', '10 minutes'])
```

(continues on next page)

(continued from previous page)

```
>>> start = fault_batches.start_time.min() - pd.Timedelta('60T')
>>> s_cols = ['time', 'turbine_num', 'stoppage', 'pre_stop', 'batch_id']
>>> scada_l.loc[(scada_l.time >= start) & (scada_l.time <= end) &
...             (scada_l.turbine_num == 22), s_cols]
                     time  turbine_num  stoppage  pre_stop  batch_id
6421 2015-12-16 09:10:00           22         0         0        -1
6422 2015-12-16 09:20:00           22         0         0        -1
6423 2015-12-16 09:30:00           22         0         0        -1
6424 2015-12-16 09:40:00           22         0         1        20
6425 2015-12-16 09:50:00           22         0         1        20
6426 2015-12-16 10:00:00           22         0         0        20
6434 2015-12-16 11:20:00           22         0         0        -1
...                   ...          ...       ...       ...       ...
6452 2015-12-16 14:20:00           22         0         0        -1
6453 2015-12-16 14:30:00           22         0         0        -1
6454 2015-12-16 14:40:00           22         0         1        21
6455 2015-12-16 14:50:00           22         0         1        21
6456 2015-12-16 15:00:00           22         0         0        21
6460 2015-12-16 15:40:00           22         0         0        -1
6461 2015-12-16 15:50:00           22         0         0        -1
```

Note that the times of the actual faults have been dropped from the data. This function can also drop additional batches from the SCADA data, so that, e.g. only times leading up to a specific type of fault are included, whereas all other stoppages are removed from the data. This is useful for building or simulating normal behaviour models.

The *wtphm.pred_processing* also has a function *wtphm.pred_processing.get_lagged_features()*. This is useful for classification, and allows features from time $t - T$ to be incorporated at time t.

2.2.8 References

2.2.9 wtphm.batch

This module contains functions for creating the `batch_data`.

See more in the *Overview*.

`wtphm.batch.`**`get_grouped_event_data`**(*event_data*, *code_groups*, *fault_codes*)

Groups together similar event codes as the same code.

This returns the events dataframe but with some fault events which have different but similar codes and descriptions grouped together and relabelled to have the same code and description.

More info in the *Group Faults of the Same Type* section of the user guide.

Parameters

- **event_data** (*pandas.DataFrame*) – The original events/fault data.
- **fault_codes** (*numpy.ndarray*) – All event codes that will be treated as fault events for the batches
- **code_groups** (*list-like, optional, default=None*) – The groups of similar events with similar codes/descriptions. Must be a list or list-of-lists, e.g. `[[10, 11, 12], [24, 25], [56, 57, 58]]` or `[10, 11, 12]`.

Returns

- **grouped_event_data** (*pandas.DataFrame*) – The `event_data`, but with codes and descriptions from `code_groups` changed so that similar ones are identical
- **grouped_fault_codes** (*pandas.DataFrame*) – The `fault_codes`, but with the similar codes in each group treated as identical

`wtphm.batch.`**`get_batch_data`**(*event_data*, *fault_codes*, *ok_code*, *t_sep_lim='12 hour'*)

Get the distinct batches of events as they appear in the `event_data`.

Each batch is a group of fault events that occurred during a fault-related shutdown. A batch always begins with a fault event from one of the codes in `fault_codes`, and ends with the code `ok_code`, which signifies the turbine returning to normal operation.

More info in can be found in *Creating Batches*.

Parameters

- **event_data** (*pandas.DataFrame*) – The original events/fault data.
- **fault_codes** (*numpy.ndarray*) – All event codes that will be treated as fault events for the batches
- **ok_code** (*int*) – A code which signifies the turbine returning to normal operation after being shut down or curtailed due to a fault or otherwise
- **t_sep_lim** (str, default='1 hour', must be compatible with `pd.Timedelta`) – If a batch ends, and a second batch begins less than `t_sep_lim` afterwards, then the two batches are treated as one. It treats the the turbine coming back online and immediately faulting again as one continuous batch. This effect is stacked so that if a third fault event happens less than `t_sep_lim` after the second, all three are treated as the same continuous batch.

Returns

batch_data (*pd.DataFrame*) – DataFrame with the following headings:

- `turbine_num`: turbine number of the batch
- `fault_root_codes`: the fault codes present at the first timestamp in the batch
- `all_root_codes`: all event start codes present at the first timestamp in the batch
- `start_time`: start of first event in the batch
- `fault_end_time`: `time_on` of the last fault event in the batch
- `down_end_time`: the `time_on` of the last event in the batch, i.e. the last `ok_code` event in the batch
- `fault_dur`: duration from start of first fault event to start of final fault event in the batch
- `down_dur`: duration of total downtime in the batch, i.e. from start of first fault event to start of last `ok_code` event
- `fault_event_ids`: indices in the events data of faults that occurred
- `all_event_ids`: indices in the events data of all events (fault or otherwise) that occurred during the batch

`wtphm.batch.`**`get_batch_stop_cats`**(*batch_data*, *event_data*, *scada_data*, *grid_col*, *maint_col*, *rep_col*, *grid_cval=0*, *maint_cval=0*, *rep_cval=0*)

Labels the batches with an assumed stop category, based on the stop categories of the root event(s) which triggered them, i.e. the one or more events occurring simultaneously which caused the turbine to stop (items lower down supersede those higher up):

- If **all** root events in the batch are "normal" events, then the batch is labelled normal

- Otherwise, label as the most common stop cat in the initial events
- If a single sensor category event is present, label sensor
- If a single grid category event is present, label grid. Also label grid if the grid counter was active in the scada data. This is a timer indicating how long the turbine was down due to grid issues, used for calculating contract availability
- If the maintenance counter was active in the scada data, label maint
- There is an additional column labelled "repair". If the repair counter was active, the turbine was brought down for repairs, and this is given the value "TRUE" for these times.

Parameters

- **batch_data** (*pd.Dataframe*) – The batch data
- **event_data** (*pd.Dataframe*) – The events data
- **scada_data** (*pd.Dataframe*) – The scada data.
- **grid_col, maint_col, rep_col** (*string*) – The columns of `scada_data` which contain availabililty counters for grid issues, turbine maintenance and repairs, resepctively
- **grid_cval, maint_cval** (*int*) – The minimum total sum of the grid, maintenance and repair counters throughout the duration of a batch for it to be marked as grid, repair or maintenance

Returns

batch_data_sc (*pd.DataFrame*) – The original `batch_data` DataFrame, but with the following headings added:

- batch_cat: The stop categories of each batch
- repairs: The repair status of each batch

`wtphm.batch.`**`get_root_cats`**(*batch_data*, *event_data*)

Gets the categories for the root alarms in the `batch_data`

Parameters

- **batch_data** (*pd.Dataframe*) – The batch data
- **event_data** (*pd.Dataframe*) – The events data

Returns root_cats (*pd.Series*) – Series of tuples, where each tuple contains strings of the `stop_cats` for each of the root alarms in a batch

`wtphm.batch.`**`get_most_common_cats`**(*root_cats*)

Gets the most common root fault category from a dictionary of root alarms

Parameters root_cats (*pd.Series*) – Series of tuples, where each tuple contains strings of the `stop_cats` for each of the root alarms in a batch

Returns most_common_cats (*pd.Series*) – Each entry in the series is a string containing the most commonly occurring root fault in `cat_counts`. In the case of a draw, then both are added, e.g. 'test, grid'

`wtphm.batch.`**`get_cat_all_ids`**(*root_cats*, *cat*)

Get an index of batches where there is only a single certain category present in the categories of the root alarms.

Parameters

- **root_cats** (*pd.Series*) – Series of strings, where each string is the categories of each of the root alarms in a batch, separated by commas.

- **cat** (*string*) – The category to check the presence of

Returns **cat_present_idx** (*pd.Index*) – The index of batch entries where `cat` was the only category present in the `root_cats`

`wtphm.batch.`**`get_cat_present_ids`**(*root_cats*, *cat*)

Get an index of batches where a certain category is present in the categories of the root alarms.

Parameters

- **root_cats** (*pd.Series*) – Series of strings, where each string is the categories of each of the root alarms in a batch, separated by commas.
- **cat** (*string*) – The category to check the presence of

Returns **cat_present_idx** (*pd.Index*) – The index of batch entries where `cat` was present in the `root_cats`

`wtphm.batch.`**`get_counter_active_ids`**(*batch_data*, *scada_data*, *counter_col*, *counter_value=0*)

Get an index of batches during which a certain scada counter was active

In 10-minute SCADA data there are often counters for when the turbine was in various different states, for calculating contractual availability. This function finds the named `counter_col` in `scada_data`, and identifies any sample periods where this value was above `counter_value`.

If any of these sample periods fall within a certain batch, then this function returns those batch ids.

Parameters

- **batch_data** (*pd.DataFrame*) – The batches of events
- **scada_data** (*pd.DataFrame*) – The 10-minute SCADA data
- **counter_col** (*string*) – The column in the SCADA data with a counter
- **counter_value** (*int*) – Any SCADA entries with a counter above this value will have their index returned

Returns **counter_active_index** (*pd.Index*) – The id's of `counter_col` columns in `scada_data` which have a val above `counter_value`.

2.2.10 wtphm.pred_processing

This module contains functions for processing scada data ahead of using it for fault detection or prognostics. Read more in the *Labelling the SCADA data* section of the User Guide.

`wtphm.pred_processing.`**`label_stoppages`**(*scada_data, fault_batches, drop_fault_batches=True, label_pre_stop=True, pre_stop_lims=['90 minutes', 0], oth_batches_to_drop=None, drop_type=None*)

Label times in the scada data which occurred during a stoppage and leading up to a stoppage as such.

This adds a column to the passed `scada_data`, "stoppage", and an optional column "pre_stop". "stoppage" is given a 1 if the scada point in question occurs during a stoppage, and "pre_stop" is given a 1 in the samples leading up to the stoppage. Both are 0 otherwise. These vary under different circumstances (see below). It also adds a "batch_id" column. For entries with a "pre_stop" or "stoppage" column of 1, "batch_id" corresponds to the batch giving it that label.

Parameters

- **scada_data** (*pandas.DataFrame*) – Full set of SCADA data for the turbine.
- **fault_batches** (*pandas.DataFrame*) – The dataframe of batches of fault events, a subset of the output of :func:wtphm.batch.get_batch_data‘

- **drop_fault_batches** (*bool, default=True*) – Whether to drop the scada entries which correspond to the stoppage periods covered by `fault_batches`. i.e. not the pre-fault data, but the fault data itself. This is highly recommended, as otherwise the stoppages themselves will be kept in the returned data, though the "stoppage" column for these entries will be labelled as "1", while the fault-free data will be labelled "0".
- **label_pre_stop** (*bool; default=True*) – If True, add a column to the returned `scada_data_l` for "pre_stop". Samples in the time leading up to a stoppage are given label 1, and 0 otherwise.
- **pre_stop_lims** (2*1 list of `pd.Timedelta`-compatible strings, default=['90 mins', 0]) – The amount of time before a stoppage to label scada as "pre_stop". E.g., by default, "pre_stop" is labelled as 1 in the time between 90 mins and 0 mins before the stoppage occurs. If ['120 mins', '20 mins'] is passed, scada samples from 120 minutes before until 20 minutes before the stoppage are given the "pre_stop" label 1.
- **oth_batches_to_drop** (*pd.DataFrame, optional; default=None*) – Additional batches, independent of dropping the `fault_batches` if `drop_fault_batches` is passed, which should be dropped from the scada data. If this is passed, `drop_type` must be given a string as well.
- **drop_type** (*str, optional; default=None*) – Only used when `oth_batches_to_drop` has been passed. If 'both', the stoppage and pre-stop entries (according to pre_stop_lims) corresponding to batches in `oth_batches_to_drop` are dropped from the scada data. If 'stop', only the stoppage entries are dropped If 'pre', opnly the pre-stop entries are dropped

Returns **scada_data_l** (*pd.DataFrame*) – The original scada_data dataframe with the "pre_stop", "stoppage" and "batch_id" columns added.

`wtphm.pred_processing.`**`get_lagged_features`**(*X*, *y*, *features_to_lag_inds*, *steps*)

Returns an array with certain columns as lagged features for classification

Parameters

- **X** (*m*n np.ndarray*) – The input features, with m samples and n features
- **y** (*m*1 np.ndarray*) – The m target values
- **features_to_lag_inds** (*np.array*) – The indices of the columns in `X` which will be lagged
- **steps** (*int*) – The number of lagging steps. This means for feature 'B' at time T, features will be added to X at T for B@(T-1), B@(T-2)...B@(T-steps).

Returns

- **X_lagged** (*np.ndarray*) – An array with the original features and lagged features appended. The number of samples will necessarily be decreased because there will be some samples at the start with NA values for features.
- **y_lagged** (*np.ndarray*) – An updated array of target vaues corresponding to the new number of samples in `X_lagged`

2.2.11 wtphm.clustering

wtphm.clustering.batch_clustering

This module is for dealing with clustering certain similar batches of turbine events together.

It contains functions for extracting clustering-related features from the batches, as well as functions for silhouette plots for evauating them.

This code was used in the following paper:

Leahy, Kevin, et al. "Cluster analysis of wind turbine alarms for characterising and classifying stoppages." IET Renewable Power Generation 12.10 (2018): 1146-1154.

`wtphm.clustering.batch_clustering.`**`get_batch_features`**(*event_data*, *fault_codes*, *batch_data*, *method*, *lo=1*, *hi=10*, *num=1*, *event_type='fault_events'*)

Extract features from batches of events which appear during stoppages, to be used for clustering.

Only features from batches that comply with certain constraints are included. These constraints are chosen depending on which feature extraction method is used. Details of the feature extraction methods can be found in [1].

Note: For each "batch" of alarms, there are up to `num_codes` unique alarm codes. Each alarm has an associated start time, `time_on`.

Parameters

- **event_data** (*pandas.DataFrame*) – The original events/fault data. May be grouped (see :func:wtphm.batch_clustering.get_grouped_events_data').
- **fault_codes** (*numpy.ndarray*) – All event codes that will be treated as fault events for the batches
- **batch_data** (*pandas.DataFrame*) – The dataframe holding the indices in `event_data` and start and end times for each batch
- **method** (*string*) – One of 'basic', 't_on', 'time'.

  basic:

  – Only considers batches with between `lo` and `hi` individual alarms.

  – Array of zeros is filled with `num` corresponding to order of alarms' appearance.

  – Does not take into account whether alarms occurred simultaneously.

  – Resultant vector of length `num_codes * hi`

  t_on:

  – Only consider batches with between `lo` and `hi` individual `time_ons`.

  – For each `time_on` in each batch, an array of zeros is filled with ones in places corresponding to an alarm that has fired at that time.

  – Results in a pattern array of length `num_codes * hi` which shows the sequential order of the alarms which have been fired.

  time:

  – Same as above, but extra features are added showing the amount of time between each `time_on`

- **lo** (*integer, default=1*) – For `method='basic'`, only batches with a minimum of `lo` alarms will be included in the returned feature set. for `method='t_on'` or `method='time'`, it's the minimum number of `time_ons`.
- **hi** (*integer, default=10*) – For `method='basic'`, only batches with a maximum of `hi` alarms will be included in the returned feature set. for `method='t_on'` or `method='time'`, it's the maximum number of `time_ons`.
- **num** (*integer, float, default=1*) – The number to be placed in the feature vector to indicate the presence of a particular alarm

- **event_type** (*string, default='fault_events'*) – The members of batch_data to include for building the feature set. Should normally be 'fault_events' or 'all_events'

Returns

- **feature_array** (*numpy.ndarray*) – An array of feature arrays corresponding to each batch that has has met the `hi` and `lo` criteria
- **assoc_batch** (*unmpy.ndarray*) – An array of 2-length index arrays. It is the same length as `feature_array`, and each entry points to the corresponding `feature_array`'s index in `batch_data`, which in turn contains the index of the `feature_array`'s associated events in the original `events_data` or `fault_data`.

References

[1] Leahy, Kevin, et al. "Cluster analysis of wind turbine alarms for characterising and classifying stoppages." IET Renewable Power Generation 12.10 (2018): 1146-1154.

`wtphm.clustering.batch_clustering.`**`sil_1_cluster`**(*X, cluster_labels, axis_label=True, save=False, save_name=None, x_label='Silhouette coefficient values', avg_pos=0.02, w=2.3, h=2.4*)

Show the silhouette scores for `clusterer`, print the plot, and optionally save it

Parameters

- **X** (*np.array or list-like*) – Features (possibly `feature_array` - need to check!)
- **cluster_labels** (*list of strings*) – the labels of each cluster
- **axis_label** (*Boolean, default=True*) – Whether or not to label the cluster plot with each cluster's number
- **save** (*Boolean, default=False*) – Whether or not to save the resulting silhouette plot
- **save_name** (*String*) – The saved filename
- **x_label** (*String*) – The x axis label for the plot
- **avg_pos** (*float*) – Where to position the text for the average silghouette score relative to the position of the "average" line
- **w** (*float or int*) – width of plot
- **h** (*float or int*) – height of plot

Returns **fig** (*matplotlib figure object*) – The silhouette analysis

`wtphm.clustering.batch_clustering.`**`sil_n_clusters`**(*X, range_n_clusters, clust*)

Compare silhouette scores across different numbers of clusters for AgglomerativeClustering, KMeans or similar

Parameters

- **X** (*np.array or list-like*) – Features (possibly `feature_array` - need to check!)
- **range_n_clusters** (*list-like*) – The range of clusters you want, e.g. [2,3,4,5,10,20]
- **clust** (*sklearn clusterer*) – the sklearn clusterer to use, e.g. KMeans

Returns

- **cluster_labels** (*numpy.ndarray*) – The labels for the clusters, with each one corresponding to a feature vector in `X`.
- *Also prints the silhouette analysis*

wtphm.clustering.batch_clustering.**cluster_times**(*batch_data*, *cluster_labels*, *assoc_batch*, *event_dur_type='down_dur'*)

Returns a DataFrame with a summary of the size and durations of batch members

Parameters

- **batch_data** (*pandas.DataFrame*) – The dataframe holding the indices in `event_data` and start and end times for each batch
- **cluster_labels** (*numpy.ndarray*) – The labels for the clusters, with each one corresponding to a feature vector in `assoc_batch`
- **assoc_batch** (*nunmpy.ndarray*) – Indices of batches associated with each `feature_array`. Obtained from :func:`.get_batch_features`
- **event_dur_type** (*string*) – The event group duration in batch_data to return, i.e. either 'fault_dur' or 'down_dur'. 'down_dur' means the entire time the turbine was offline, 'fault_dur' just means while the turbine was faulting. See :func:`wtphm.batch.Batches.get_batch_data` for details

Returns summary (*Pandas.DataFrame*) – The DataFrame has the total duration, mean duration, standard deviation of the duration and number of stoppages in each cluster.

wtphm.clustering.event_probs

This module is for working with events data from wind turbines. It looks at all eventes generated and sees if there are some events which trigger others. Event A triggers Event B if: t_s_A <= t_s_B and t_e_A >= t_s_B

So we can find the probability that any given A event (known as a parent event) has triggered any B events, and the probability that any given B event (known as a child event) has been triggered by any A events.

wtphm.clustering.event_probs.**get_trig_summary**(*events*, *codes*, *tsa_op1='ge'*, *tsa_op2='le'*, *t_hi=0.9*, *t_lo=0.1*)

Gets probabilities that pairs of events will trigger one another, and the derived relationship between these pairs

This function takes a list of event codes. It finds all combinations of pairs of codes from this and splits them into "A" and "B" codes. It then counts the number of events with code A which have triggered one or more events with code B and vice-versa. It then computes a probability that if an A event occurs, it will trigger a B event, and vice-versa. From there, it deduces the relationship between pairs of events, as derived from [1].

Event A is triggered by Event B if:

T_s_A >= T_s_B & T_s_A <= T_e_B

where T_s_A, T_s_B and T_e_B are the start time of events A and B, and the end time of event B, respectively.

Parameters

- **events** (*pandas.DataFrame*) – The events data from a wind turbine. Must be free of NA values.
- **codes** (*list-like*) – The event codes to look at
- **tsa_op1** (*String, default 'ge'*) – Operator to use for T_s_A >= T_s_B or T_s_A > T_s_B. Can be one of:

  'ge': <= 'gt': <
- **tsa_op2** (*String (default 'le')*) – Operator to use for T_s_A <= T_e_B or T_s_A < T_e_B. Can be one of: 'le': >= 'lt': >

- **t_hi** (*float (default 0.9)*) – Threshold of % of A events which trigger B events at or above which relationship 3 is *True* (or % B triggering A for relationship 4, or % of both for relationship 1). See 'relationship' in the returned *trig_summary* dataframe below.
- **t_low** (*float (default 0.1)*) – Threshold of % of A events which trigger B events (or vice-versa) at or below which relationship 2 is *True*. See 'relationship' in the returned *trig_summary* dataframe below.

Returns

trig_summary (*Pandas.DataFrame*) – A matrix consisting of the following:

- *A_code*: the event code of the "A" events
- *A_desc*: description of the "A" events
- *B_code*: the event code of the "B" events
- *B_desc*: description of the "B" events
- *A_count*: number of "A" events in the data
- *A_trig_B_count*: number of "A" events which trigger one or more "B" events
- *A_trig_B_prob*: ratio of "A" events which have triggered one or more "B" events, to the total number of "A" events
- *B_count*: Number of "B" events in the data
- *B_trig_A_count*: number of "B" events which trigger one or more "A" events
- *B_trig_A_prob*: ratio of "B" events which have triggered one or more "A" events, to the total number of "B" events
- *relationship*: Number 1-5 indicating the relationship events A have to events B:

1. High proportion of As trigger Bs & high proportion of Bs trigger As. Alarm A & B usually appear together; A ~= B
2. Low proportion of As trigger Bs & low proportion of Bs trigger As. A & B never or rarely appear together; A n B ~= 0
3. High proportion of As trigger Bs & less than high proportion of Bs trigger As. B will usually be triggered whenever alarm A appears - B is a more general alarm; A e B
4. High proportion of Bs trigger As & less than high proportion of As trigger Bs. A will usually be triggered whenever alarm B appears - A is a more general alarm; B e A
5. None of the above. The two alarms are randomly or somewhat related; A n B != 0

References

[1] Qiu et al. (2012). Wind turbine SCADA alarm analysis for improving reliability. Wind Energy, 15(8), 951–966. http://doi.org/10.1002/we.513

`wtphm.clustering.event_probs.`**`short_summary`**(*trig_summary*, *codes*, *t=0.7*)

Returns an even more summarised version of trig_summary, showing important relationships

Parameters

- **trig_summary** (*Pandas.DataFrame*) – Must be the *trig_summary* obtained from `get_trig_summary()`
- **codes** (*int, list*) – A single, or list of, event code(s) of interest, i.e. the events that trigger other events

- **t** (*float*) – The threshold for a 'significant' relationship. E.g., if t=0.7, only events that trigger other events with a probability >= 0.7 will be displayed.

Returns

- **df** (*Pandas.DataFrame*)
- *A dataframe consisting of the following –*
  - *parent_code*: the triggering events code
  - *child_code*: the triggered events code
  - *trig_prob*: the probability that *parent_code* events will trigger *child_code* events
  - *trig_count*: the count of *parent_code* events which have triggered *child_code* events

`wtphm.clustering.event_probs.`**`get_trig_summary_verbose`**(*events*, *codes*, *tsa_op1='ge'*, *tsa_op2='le'*)

Gets probabilities that certain events will trigger others, and that certain events will be triggered *by* others. Can be calculated via a duration-based method, or straightforward count.

This takes a list of event codes. It creates two separate sets of "parent" and "child" events, with all the parent events having the same event code and all the child events having another event code (though it does not necessarily have to be different). It then iterates through every parent event instance to see if it has triggered one or more child events. It counts the number of parent events which have triggered one or more child events for each event code. It also gives a probability that any new parent event will trigger a child event by finding the ratio of parent events which have triggered a child event to those which haven't.

Event A is triggered by Event B if:

T_s_A >= T_s_B & T_s_A <= T_e_B

where T_s_A, T_s_B and T_e_B are the start time of events A and B, and the end time of event B, respectively.

Parameters

- **events** (*Pandas.DataFrame*) – The events data from a wind turbine. Must be free of NA values.
- **codes** (*list-like*) – The event codes to look at
- **tsa_op1** (*String (default 'ge')*) – Operator to use for T_s_A >= T_s_B or T_s_A > T_s_B. Can be one of: 'ge': <= 'gt': <
- **tsa_op2** (*String (default 'le')*) – Operator to use for T_s_A <= T_e_B or T_s_A < T_e_B. Can be one of: 'le': >= 'lt': >

Returns

trig_summary (*Pandas.DataFrame*) – A matrix consisting of the following:

- *parent_event*: the event code of the parent event
- *parent_desc*: description of the parent event
- *p_count*: total number of parent events matching the event code
- *p_dur*: total duration of parent events matching the event code
- *p_trig_count*: number of parent events which have triggered child events
- *p_trig_dur*: duration of parent events which have triggered child events
- *child_event*: the event code of the child event
- *child_desc*: description of the child event

- *c_count*: total number of child events matching the event code
- *c_dur*: total duration of child events matching the event code
- *c_trig_count*: number of child events which have been triggered by parent events
- *c_trig_dur*: duration of child events which have been triggered by parent events

CHAPTER 3

Github Page

Can be found at https://github.com/lkev/wtphm.

CHAPTER 4

Indices and tables

- genindex
- modindex

Python Module Index

w

Index