Documentation for the PerfMon Anomaly Detector and Analyzer (git: dev/gts.tsa.anomaly)
Created: 14/08/2017
Updated: 22/09/2017
Author: Xingchen Wan

The Anomaly Detector detects anomalous behaviours in the time series. (e.g.: sudden spikes and falls, mean shift and persistent trend)

The Anomaly Analyzer analyses the output from Anomaly Detector, and explores potential association, correlation or other interesting relationships between occurrences of different types of anomalies in different data series (e.g. two or more anomalies always occur at the same time). It is also able to find correlations between data series themselves, which include both anomalous and non-anomalous points.

This documentation is meant to be a comprehensive reference. For quick start, go to Page 4

Usage notices in the documentation are in three levels:

Warning signals that an inappropriate may significantly corrupt the performance of the tools.
Caution signals an important notice, and usually restrictions on configurations and result interpretation.
Note are explanatory texts or reminders on the functionalities or configurations.

Note:
a. Analyzer takes output of Detector as input, it will not work if run before Detector or logging of Detector is turned off.
b. Analyzer does not analyze the original time series data, but analyzes the anomalies identified by Detector.

Overview of File Structure:

Modules (/modules)**:**
1. modules/**DataFetcher.py**: Fetches data from MySQL database or csv file *(Detector only)*.
2. modules/**Pickler.py**: reads and saves the output form DataFetcher into local persistent data files ("pickles"), so that every time the detector is run, only new data is processed and old ones are not processed repeatedly. *(Detector Only)*
3. modules/**Detector.py**: the master module of AnomalyDetector that directly handles user request to detect anomalies in frames or date(s). *(Detector Only)*
4. modules/**Initializer.py**: Initialize/Update observables according to user configuration. *(Both Detector and Analyzer)*
5. modules/**Initializer_xxx.py**: The initializer for data other than those from perfmondb. Xxx is the corresponding data type (e.g. perfcoldb means the initializer for raw transactions data) this file is used **in lieu of** initializer.py for these cases.
5. modules/**Logger.py** and **Visualizer.py**: Creates textual/visual output from the detectors. *(Both Detector and Analyzer)*
6. modules/**Analyzer.py**: the master module of Associator and Correlator that directly handles user request to detect anomalies in frames or date(s). *(Analyzer Only)*

Config (/config.py)**:**
Master configuration file for the Anomaly Detector and Analyzer. Configures the parameters of algorithms, connection profile of databases and different observables.

Algorithms (/analyzer_algorithms and /detector_algorithms)**:**
1. detector_algorithms: Core algorithms of the Anomaly Detector (See Section III)
2. analyzer_algorithms: Core algorithms of the Anomaly Analyzer (See Section IV)

Intermediate Files (/pickles)**:**
Directory containing the files generated/accessed by the program. Handled primarily by modules/Pickler.py. Internally grouped in the hierarchy of pickles/*type of pickle, (e.g. time series, assemblers)*/*observable*.pkl

Input (/input):
The tool is designed for data of MySQL table PerfMon/perfmondb – the processed data. Data can either be directly fetched from the data table, or read from a csv file that has a data structure similar to the table.

However, certain different data structure (e.g. PerfMon/perfcoldb – the raw transactions) may be parsed into a compatible data structure (see Section VI: Parsers) via a parser module (Initializer_xxx.py).

Output (/output):
Default directory of Detector output and is internally grouped in the hierarchy of output/*observable name*/ *string key*. csv

Training Files for Assemblers (/assembler_train)
Training files for SVM assemblers. (See Section III: Assemblers)
Documentation Summary:

Section I and II provide a high-level overview of the system structure.
Section III, IV and V explain how to configure Anomaly Detector, Analyzer and data retrieval modules.
Section VI explains the configuration of wrappers, which are modules to parse input data other than those retrieved from perfmondb.
Section VII explains how the textual and visual output from the Detector/Analyzer may be interpreted.
Section VIII contains the documentation on the usage of the functions that are callable by the end-user.
Section IX contains the bibliography and acknowledgements to the prior work this system has benefited from.

Development Information:

Environment:
Anaconda Python 3.5.2
(Not Python 2.x compatible)

External Packages:
pandas*
numpy*
scikit/learn*
scipy*
statsmodels*
matplotlib*
networkX*
networkXViewer^{+}
** Included in Anaconda*
+ Optional

Contact Information

Xingchen Wan
Summer Intern,
Trading System Analytics

Email: xingchen.wan@deutsche-boerse.com
Alternative email: xingchen.wan@st-annes.ox.ac.uk

Table of Contents

Quick Start

For details on the arguments of the functions (and other functions not shown here), go to Section VIII.

To run your task as a python script, enter the script in main.py. To run interactively, use command_line.py

Example 1:

Go to the root directory of the Anomaly Detector and Analyzer, in terminal run:

... python command_line.py

If an error prompts, edit command_line.py and ensure the path after #! is the local directory for the correct python interpreter.

>>> **Initializer.update()**

This command pulls new data from database and merges the new data with the existing record. This creates new files when the program is run for the first time. See details in Section VIII.

>>> **Detector.detect_timestamp("2017-07-20")**

This command analyzes everything on the timestamp specified. To filter the data see details in Section VIII

The output csv is grouped by string key. To group the anomalies by timestamp, use:

>>> **Analyzer.get_anomalous_records(time_from='2017-07-20', time_to="2017-07-20")**

Example 2:

You are suspicious of certain business ID/latency and would like to run analysis for that across all available timestamps. Again, in command_line.py, enter

>>> **Detector.detect_frames(observable_filter='WHATEVER OBSERVABLE YOU ARE INTERESTED', frame_filter='BUID YOU ARE INTERESTED)**

To show the plot,

>>> **Analyzer.plot_from_log("OBSERVABLE", "BUID")**

If graph does not show up,

>>> **plt.show()**

Example 3:

Investigate relationships between occurrences of anomalies

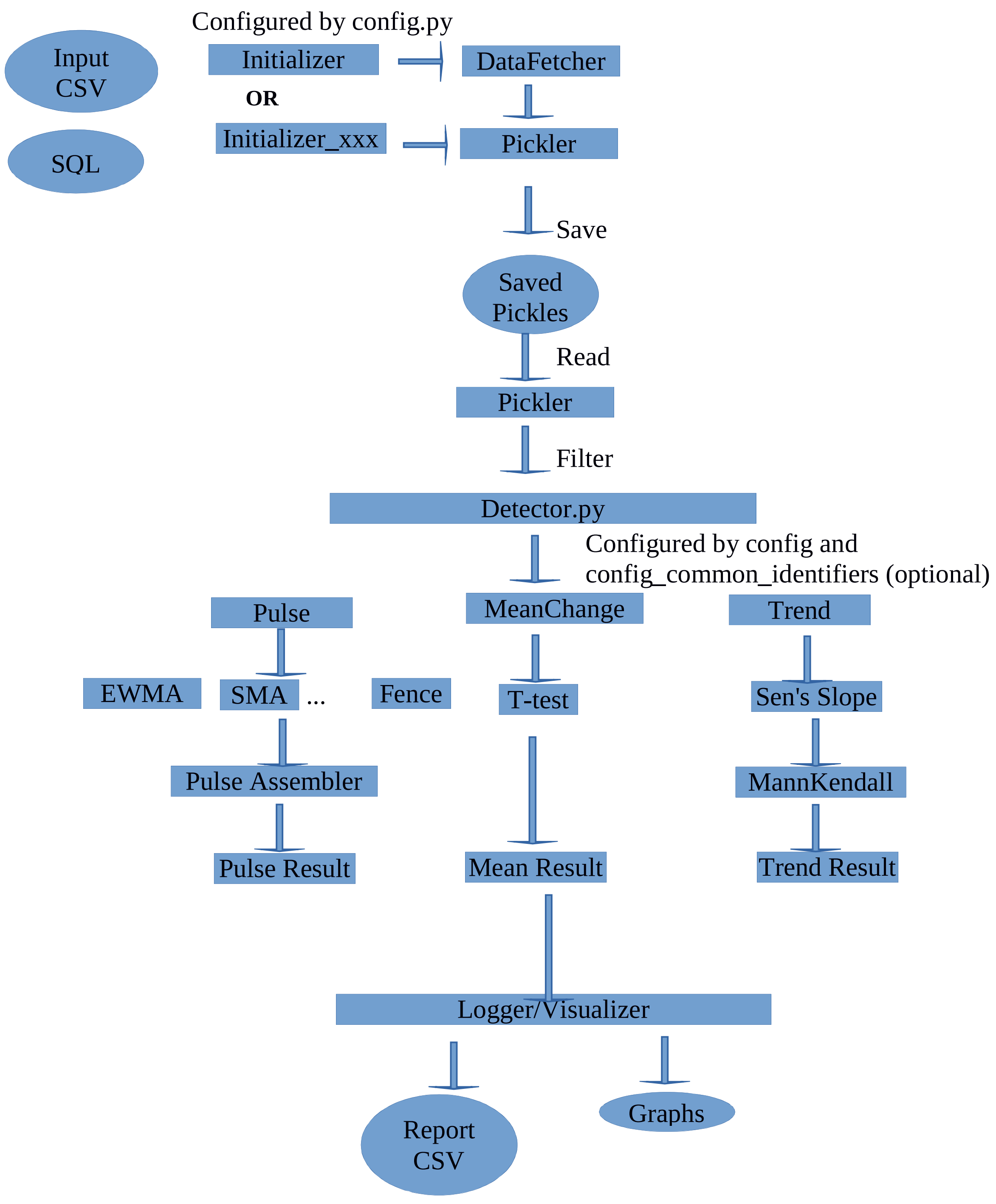
>>> **Analyzer.get_associations()**

If you enabled graph, again

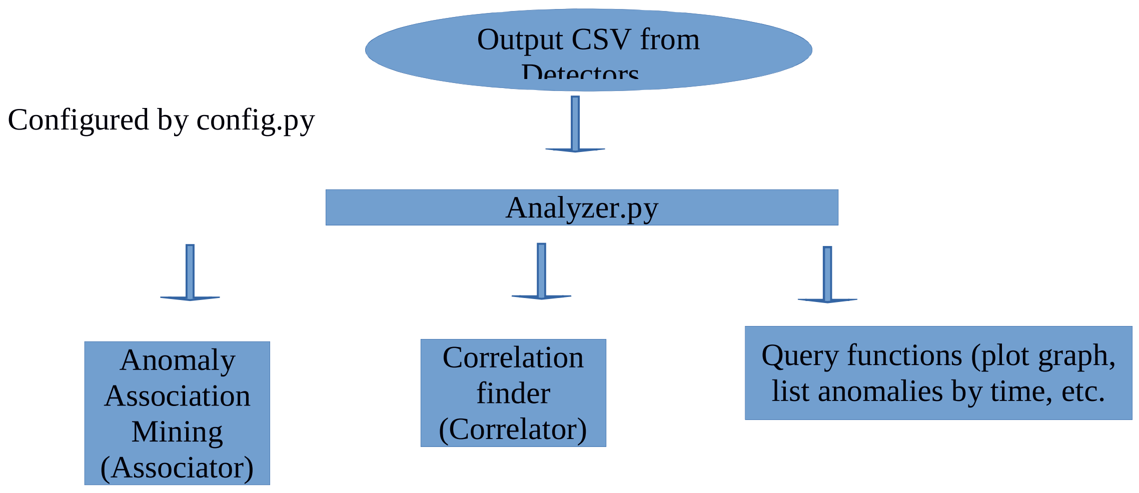
>>> **plt.show()**

You can also inspect the textual output. See section VII for details.

I. Operational Flowchart of Anomaly Detector



II. Operational Flowchart of Anomaly Analyzer


Output CSV from Detectors
Configured by config.py
Analyzer.py
Anomaly Association Mining (Associator)
Correlation finder (Correlator)
Query functions (plot graph, list anomalies by time, etc.

III. Anomaly Detection Algorithms and Configurations

Detecting algorithms are organized into five different files.
Configurations are parameters input of the algorithms. All Configurations (both detector and analyzer) are stored centrally at **config.py** in the root directory.

1. AnomalyDetector.py (Prototype**)**:

A file used to define the basic structure of detector object and some common methods/properties across all detectors; not meant to be invoked directly by user

*Configurations **in config.py***:
ENABLE_PULSE_DETECTORS (boolean): Self-explanatory
ENABLE_MEAN_CHANGEPT_DETECTORS (boolean): Self-explanatory
ENABLE_TREND_DETECTORS (boolean): Self-explanatory

*Consider turning off Mean Change and Trend Detectors for intra-day data to increase running time. Also, since the timespan for intraday data is much smaller, the temporal effect on the structure of time series (e.g. mean-shift and significant upwards or downwards trend) may be safely neglected.

2. MeanChange.py: detects step and persistent change in mean

- Welch's t-test

*Configurations **in config.py***:
MEANCHANGE_REF_SIZE (int >0): Size of reference window
MEANCHANGE_PROBE_SIZE (int, ∈ (0, REF_SIZE)): Size of probe window.
MEANCHANGE_ALERT_PCT_THRESHOLD (float, >0): Detection threshold e.g. 0.2 means only mean change more than 20% will be alerted.
MEANCHANGE_ALPHA (float, ∈ (0, 1]: Threshold p-value for rejecting/accepting the null hypothesis
MEANCHANGE_TEST_FREQ (int >0): Frequency of mean change test.

Welch's t-test, or unequal variances t-test, is a two-sample location test which is used to test the hypothesis that two populations have equal means. Welch's t-test is an adaptation of Student's t-test.

t-statistic is defined by:

$$t = \frac{\overline{X_1} - \overline{X_2}}{\sqrt{\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}}}$$

Where $\overline{X}$, s, N are mean, variance and number of samples respectively. 1 and 2 subscripts refer to reference and probe windows.

At point t, the probe window is taken as the set of points [t-PROBE_SIZE:t] and reference window is [t-PROBE_SIZE-WINDOW_SIZE:t-PROBE_SIZE-1]. Two-tailed t-test is conducted on these two set of samples and reports any change in mean found.

Caution: to ascertain a change in mean is sustained, alert will be delayed by PROBE_SIZE days after the actual change point occurred. Reducing PROBE_SIZE reduces delay, but also increases the chance of false positive.

3. Trend.py: detects the presence and strength of any trend presence in the data.

- **Mann-Kendall test.**

  Mann-Kendall test is a non-parametric test on whether there is a monotonic increasing or decreasing trend in the data set. Null hypothesis is that there is no trend in the data and alternative hypothesis is that there is an increasing or decreasing monotonous trend.

  Mann Kendall statistic is given by:

$$S = \sum_{k=1}^{n-1} \sum_{j=k+1}^{n} sgn(x_j - x_k)$$

$$sgn(x_j - x_k) = \begin{cases} 1 \; if \; x_j - x_k > 0 \\ 0 \; if \; x_j - x_k = 0 \\ -1 \; if \; x_j - x_k < 0 \end{cases}$$

  Where x_j, x_k are j-th and k-th observations.
  For large sample sizes (> 10), statistic S is approximately normal with mean and variances:

$$E(S) = 0$$

$$Var(S) = \frac{1}{18}[n(n-1)(2n+5) - \sum_{o=1}^{q} t_p(t_p - 1)(2t_p + 5)$$

  Where q is the number of tied groups.
  From which standard test statistic Z is computed as:

$$Z = \begin{cases} \frac{S-1}{\sqrt{Var(S)}} \; if \; S > 0 \\ 0 \; if \; S = 0 \\ \frac{S+1}{\sqrt{Var(S)}} \; if \; S < 0 \end{cases}$$

  Null hypothesis is rejected if $|Z| > Z_{1-\frac{\alpha}{2}}$ where $Z_{1-\frac{\alpha}{2}}$ is obtained from the standard normal distribution.
  For details of the Mann-Kendall test, please refer to
  http://vsp.pnnl.gov/help/Vsample/Design_Trend_Mann_Kendall.htm

- **Sen's slope**

  Configurations ***in config.py:***
  TREND_WINDOW_SIZE (int >0): Size of sliding window
  TREND_ALPHA (float, ∈ [0, 1]): Threshold p-value used in estimating Sen's slope
  TREND_ALERT_PCT_THRESHOLD (float, >0): Proportional change threshold for Sen's slope method. For example, 0.25 suggests that if dataset value changes more than 25% within TREND_WINDOW_SIZE days, a positive result will be reported.

  Sen's slope is an outlier-robust method in estimating the strength of data trend.

  To derive an estimate of the slope Q in the time series with N data points, slopes of all data pairs are first calculated:

$$Q_i = \frac{x_j - x_k}{j - k}, i = 1, 2, \ldots N, j > k$$

  Sen’s estimator of slope Q is given by:

$$Q = \begin{cases} Q_{\frac{N+1}{2}} \; if \; N \; is \; odd \\ \frac{1}{2}\left(Q_{\frac{N}{2}} + Q_{\frac{N+2}{2}}\right) if \; N \; is \; even \end{cases}$$

Note: The two algorithms are executed **sequentially**. Sen's slope is conducted on each relevant point in a window period defined by the user, if the slope exceeds some user defined threshold, Mann-Kendall test will be executed. If the test gives a positive result a trend flag will be marked.

4. Pulse.py: detects sudden and transient pulses in the data, which is the classical definition of anomalies. Note that the ensemble parameters are configured in a way to induce diversity across different base detectors, which improves the performance of the ensemble as a whole.

Different pulse algorithms have different support for online vs offline mode. Under online mode, the algorithm proceeds chronologically with a sliding window specified by window size, and decision on whether a point is anomalous depends only on data points before the present point. Under offline mode, there is no sliding window and algorithm takes all data points available and decision may depend on points after the present point. **Offline mode** is vectorised and is much faster, and can be used when the effect of time on the structure of the data is limited (such as for intra-day data, where data is largely time-stationary). **Online mode** is slower due to the sliding window inner-loop but is contextually adapted, and should be used when time has significant impact on data (e.g. daily data spanning over years).
Warning: offline mode is unsuitable for data with strong temporal influence, as it essentially ignores the effect of time and assumes the series to be stationary; force enabling them under such case can significantly deteriorate the performance of pulse detection!

Algorithm	Offline?	Online?
CUSUM-EWMA Method 1	No	**Yes**
CUSUM-EWMA Method 2	No	**Yes**
Simple Moving Average	**Yes** (Set WINDOW_SIZE to 'all')	**Yes** (Any numeric WINDOW_SIZE)
Grubbs Outlier Test	No	**Yes**
Least Square	No	**Yes**
Median Absolute Deviation	**Yes** (Set WINDOW_SIZE to 'all')	**Yes** (Any numeric WINDOW_SIZE)
Fence	**Yes** (Set WINDOW_SIZE to 'all')	**Yes** (Any numeric WINDOW_SIZE)
Local Outlier Factor	**Yes**	No
Isolation Forest	**Yes**	No

- **CUSUM-EWMA Method 1:**

  *Configurations **in config.py***:
  (Configurations are shared with CUSUM-EWMA Method 2)
  PULSE_EWMA_WINDOW_SIZE (int >0): Size of sliding window;
  PULSE_EWMA_SIGMA(int/float >0): Default value is 3, which means that if actual value deviates 3 standard deviation from the expected value, alert will be triggered;

  Cumulative Sum-Exponentially Weighted Moving Average (Method 1) combines classical EWMA and CUSUM control charts for anomaly detection.

  Method 1 uses the methodology suggested by Christodoulou, Vyron, and Yaxin Bi in paper "A combination of CUSUM-EWMA for Anomaly Detection in time series data." Data Science and Advanced Analytics (DSAA), 2015. 36678 2015. IEEE International Conference on. IEEE, 2015.

- **CUSUM-EWMA Method 2:**
  *Configurations **in config.py***:
  (Configurations are shared with CUSUM-EWMA Method 1)
  PULSE_EWMA_WINDOW_SIZE (int >0): Size of sliding window;
  PULSE_EWMA_SIGMA(int/float >0): Default value is 3, which means that if actual value deviates 3 standard

deviation from the expected value, alert will be triggered;

Method 2 uses the methodology suggested by N. Abbas, M. Riaz and R. J. Does in paper "Mixed exponentially weighted moving average cumulative sum charts for process monitoring" Quality and Reliability Engineering International, vol. 29, no. 3, pp. 345-356, 2013

- **Simple Moving Average (SMA):**

*Configurations **in config.py**:*
PULSE_SMA_WINDOW_SIZE (int >0 or 'all'): Same meaning as above. If 'all', the algorithm will not use sliding window but the entire series is tested at one iteration (offline mode)
PULSE_SMA_SIGMA (int/float >0): Same meaning as above.

SMA is to capture global anomalies. Hence a larger WINDOW_SIZE is recommended.

- **Grubb's Outlier Test,**

*Configurations **in config.py***:
PULSE_GRUBBS_WINDOW_SIZE (int >0): Same meaning as above
PULSE_GRUBBS_MAX_PREV_OUTLIER (int ∈ [0, WINDOW_SIZE]): Maximum previous number of outliers tolerated. Grubb's tests single outlier each time, so existing outliers in the sliding window impedes its performance. A non-zero entry will make the detector to test outliers first in the sliding window successively, and remove these anomalies up to MAX_PREV_OUTLIER before conducting Grubb's test on the actual test data.
Caution: If more outliers than MAX_PREV_OUTLIER is found in the sliding window, an error will be generated and the detector will return an inconclusive result.

Non-parametric Grubb's outlier test. At each step, it computes the Grubb's statistic of the series and the z-score of the test point and flags the point if the z-score exceeds the Grubb's statistic.

Grubbs statistic is given by:

$$G = \frac{\max_{i=1,\ldots,N} \left| Y_i - \overline{Y} \right|}{s}$$

Where $\overline{Y}$, s are sample mean and standard deviation respectively.

Null hypothesis is that the samples contain no outlier. Null hypothesis is rejected when:

$$G > \frac{N-1}{\sqrt{N}} \sqrt{\frac{t^2_{\frac{\alpha}{2N},N-2}}{N-2+t^2_{\frac{\alpha}{2N},N-2}}}$$

Where $t^2_{\frac{\alpha}{2N},N-2}$ is the upper critical value of t-distribution with N-2 d.o.f and a significance level of$\frac{\alpha}{2N}$. (Two-tailed)

- **Least Square Projection,**

*Configurations **in config.py***:
PULSE_LS_WINDOW_SIZE (int >0): Same meaning as above.
Caution: Recommended value > 10 for a reasonably reliable linear regression model to be established.
PULSE_LS_SIGMA (int/float >0): Same meaning as above

Least-square method first builds a linear regression model (OIS) based on the WINDOW_SIZE number of points prior to the point of interest, then it uses the linear model to predict the value of the point of interest. If the actual value deviates too much from the predicted value, an alert will be triggered.

Least-square optimizes by minimising the sum of squared residuals (SSR). Denote x as the observation and t the time. Suppose b as the parameter vector. SSR is given by:

$$S(b) = \sum_{i=1}^{n} (x_i - t_i^T b)^2$$

This function has a global minimum at b = β, given by:

$$\beta = \underset{b}{\operatorname{argmin}}\, S(b) = \left(\frac{1}{n}\sum_{i=1}^{n} t_i t_i^T\right)^{-1} * \frac{1}{n}\sum_{i=1}^{n} t_i x_i$$

- **Median Absolute Deviation (MAD).**

*Configurations **in config.py***:
PULSE_MAD_WINDOW_SIZE (int >0 or 'all'): Same meaning as above. If 'all', the algorithm will not use sliding window but the entire series is tested at one iteration (offline mode)
PULSE_MAD_SIGMA (int/float >0): Same meaning as above

MAD is a robust measurement of the spread of the data set. The algorithm first computes the MAD in the WINDOW_SIZE number of points prior to the point of interest, then computes the median deviation of the point of interest with respect to the previous calculation. If the actual value deviates too much from the predicted value, an alert will be triggered.

MAD is given by:

$$MAD = median(|X_i - median(X)|)$$

For univariate dataset X_1, X_2 … X_n
Caution: a non-zero MAD is required for effective detection. If MAD is zero, the detector will return an inconclusive result.

- **Fence**

*Configurations **in config.py***:
PULSE_FENCE_WINDOW_SIZE (int >0) or 'all': Same meaning as above. If ‘all’, the algorithm will not use sliding window but the entire series is tested at one iteration (offline mode)
PULSE_FENCE_ALPHA (float, ∈ (0, 1]): Alert trigger. e.g. 0.02 means values above 98th percentile under one tailed mode, or values below 1st and above 99th percentiles under two-tailed mode, will be flagged.
PULSE_FENCE_TWO_TAILED (boolean): Whether both upper and lower bounds are activated. If false, this detector only detects upper extremities.

Fence flags extreme values in last WINDOW_SIZE days.

- **Local Outlier Factor (LOF) – Offline only**

*Configurations **in config.py***:
PULSE_LOF_K (int >0): Value of k in LOF's k-NN score computation
PULSE_LOF_MIN_WINDOW_SIZE (int >0): Minimum series length for LOF algorithm to be enabled. NaN will be returned if series entered has length below this value.
PULSE_LOF_CONTAMINATION (float ∈ [0, 0.5]): Estimated proportion of outliers in the dataset.

The local outlier factor is based on a concept of a local density, where locality is given by nearest neighbours, whose distance is used to estimate the density. By comparing the local density of an object to the local densities of its neighbours, one can identify regions of similar density, and points that have a substantially lower density than their neighbours. These are considered to be anomalies.

- **Isolation Forest – Offline only**

*Configurations **in config.py***:
PULSE_IFOREST_CONTAMINATION (float ∈ [0, 0.5]): Same as above
PULSE_LOF_MIN_WINDOW_SIZE (int >0): Same as above

The Isolation Forest 'isolates' observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature.

Since recursive partitioning can be represented by a tree structure, the number of splitting required to isolate a sample is equivalent to the path length from the root node to the terminating node.

This path length, averaged over a forest of such random trees, is a measure of normality and our decision function.

Random partitioning produces noticeably shorter paths for anomalies. Hence, when a forest of random trees collectively produce shorter path lengths for particular samples, they are highly likely to be anomalies.

Note: The algorithms are executed **independently.** The output from each detector will then be passed to Pulse Assembler to give a final decision on whether the point is an anomaly.

5. PulseAssembler.py**:** Takes in the output from the pulse base detectors and assemble them into a final decision.

*Configurations **in config.py***:
PULSE_ASSEMBLER_TYPE (str ∈ ['vote', 'logic_or', 'SVM']): Type of assembler. See below for details on each type.

- **Vote** (default):

  *Configurations **in config.py***:
  PULSE_ASSEMBLER_VOTE_THRESHOLD (int, ∈ [1, number of pulse algorithms] or float, ∈ (0, 1]): Threshold for alert trigger. e.g. if float, 0.5 causes the assembler to flag points as anomalous if more than 50% of detectors agree so; if int, 3 causes the assembler to flag points anomalous if more than 3 detectors agree so. Each base detector is given an equal voting weight in this process. Unavailable base detectors (returning NaN will be excluded from proportions)
  PULSE_ASSEMBLER_VOTE_QUORUM (int, ∈ [0, number of pulse algorithms]): Minimum number of available base detectors (non-NaN returning) required for majority vote assembler to produce a positive result.

- **Logic OR:**

  Equivalent to "Vote" assembler with THRESHOLD set to 1.
  If any detector marks a point as anomalous, the point will be flagged as an anomaly. Primarily used to test performance of individual detector and how diverse different detectors span the sample space, and not meant for practical use as it will mark too many positive results.

- **Incremental Support Vector Machine (SVM):**

  The SVM is first initialized to imitate the behaviour of another assembler ("Vote" by default). Subsequently it is able to learn from user feedback and adjust the weight of the different detectors, discounting poorly-performing detectors and boosting detectors that best match user feedback.
  Caution: SVM needs to be initialized before it can be used! See the section containing documentation of the callable functions on how to initialize SVM.

IV. **Anomaly Analyzer Algorithms and Configurations**

1. Associator.py

Associator uses FPGrowth algorithm to generate the frequent patterns in the observations. It is able to detect associations between different types of anomalies in different observables, provided the pattern occurs frequent enough.

Caution: Associator takes the output of Anomaly Detector as input so it is important that Anomaly Detector output logs are not modified after they are generated!
Note While theoratically FPGrowth is able to generate any length of frequent patterns, the running speed will be extremely slow for large dataset with long frequent patterns, hence the pattern length is limited to 2. However, as a corollary from the Apriori theorem, this restriction does not cause any loss of useful information because set of items in frequent patterns with length more than 2 must be a subset of the set of patterns with length equal to 2. (E.g. if pattern (A, B, C) is a frequent pattern, (A, B), (B, C), (A, C) must be frequent patterns too. That implies if every item is regarded as a node and every correlation between a pair of items regarded as an edge in an undirected graph, patterns with length n will be n-cliques in the graph.)

Configurations ***in config.py:***
ANALYZER_MIN_SUPPORT (int > 0 or float (0, 1)): Minimum support for a pattern to be classified as a "frequent pattern". If int, this argument is interpreted literally. (E.g. if set to 10, patterns recurring more than or equal to 10 times are deemed 'frequent'. If float, the threshold support would be the product of this argument and number of observations (e.g. if set to 0.2, patterns occurring in more than 20% of the total observations will be deemed frequent)
Caution: Because anomalies are by definition rare (sometimes < 1%), using fractional MIN_SUPPORT can sometimes lead to no result at all.
Caution: FPGrowth has a time complexity of $O(n^2)$, where n is the number of items in header table (i.e. the items with occurrences above MIN_SUPPORT). Hence, if the MIN_SUPPORT is set to be too low, the algorithm can take a long time to run. On the other hand, since only anomalies are considered interesting events, MIN_SUPPORT should not be too high, too since anomalies are by definition rare. Therefore, some fine tuning here may be required.

ANALYZER_MIN_LIFT (float >= 0):
If ANALYZER_MIN_LIFT > 1, patterns with **lift > ANALYZER_MIN_LIFT** or **lift < 1 / ANALYZER_MIN_LIFT** are accepted.
If ANALYZER_MIN_LIFT < 1, patterns with **lift < ANALYZER_MIN_LIFT** or **lift > 1 / ANALYZER_MIN_LIFT** are accepted.
If ANALYZER_MIN_LIFT = 1 or ANALYZER_MIN_LIFT = 0: All patterns are accepted provided they meet other criteria.
If you do not wish to use MIN_LIFT filter, set ANALYZER_MIN_LIFT to 0 rather than 1.

Lift measures how dependent two or more items are. Mathematically, it is defined as:

$$Lift(A, B) = \frac{P(A \cap B)}{P(A)P(B)}$$

For pattern with length 1 lift is defined as 1. Lift > 1 suggests positive correlation and Lift < 1 suggests negative correlation. This is the reason why reciprocal of the value supplied is used in the boundary, too.

ANALYZER_MIN_CONF (float (0, 1)): Minimum confidence. Note that if **either** Confidence1 (P (A|B)) or Confidence 2 (P (B|A)) is above this threshold, that pattern will be deemed as satisfying this filter.

Confidence measures how confident the established is true out of the observations, mathematically, it is defined as:

$$Conf(A \rightarrow B) = P(A|\mathrm{B})$$

2. FPGrowth.py

This file contains the implementation of the FPGrowth algorithm used by Associator.

3. Correlator.py

Correlator is meant to be an in-depth relationship explorer of two series after Associator indicates some sort of interesting relations. It attempts to correlate **all** data points (not just anomalous points, as in Analyzer) two series and aims to find correlation in all or some parts of the data.
Correlator uses a correlation metric (Pearson's r, Spearman's ρ or Kendall's τ) to determine the strength of correlation between two series. It is also able to roughly determine a threshold in either x and y, if any, above which the correlation is stronger than the overall correlation.
Caution: This function uses an experimental algorithm, as no relevant research was found, so this function should be used with discretion.

The algorithm builds upon the assumption that generally truncation of data reduces correlation coefficient and if that is not true, it is likely that there is floor or ceiling effect in the data set (e.g. correlation only holds when a threshold is reached).
(See Section VII for examples with and without these effects and the output from this function for each case.)

Algorithm Pseudo-code:

```
Input: series1, series2, precision, min_point
r is a zero-matrix with numbers of columns and rows set to precision
reward us a zero-matrix with numbers of columns and rows set to precision

for each data point in series1 and series2:
    From a 2-tuple in form (x, y) = (series1Value, series2Value)
end for

for each series,
    Find boundary of segments of the series with constant interval
    // If series spans from 0 to 100 and precision=10, segment boundaries are (0, 10, 20, …, 100)
end for

for series1Boundary in series1SegmentBoundaries:
    for series2Boundary in series2SegmentBoundaries:
        Do linear regression on all points where x > series1Boundary and y > series2Boundary
        if number of points included < min_point:
            break the inner loop
        end if
        r[series1Boundary, series2Boundary] ← Selected correlation coefficient
        reward[series2Boundary, series2Boundary] ← Selected reward
    end for
end for

find the point with highest reward value
return the boundary values of that point
```

*Configurations **in config.py:***
None
Note: Correlator is not designed to find relationships amongst a large number of data series. That should be done by Associator instead. Therefore, the parameters of the correlator algorithm is instead defined as argument to the calling function. See Section VIII: Callable Functions → get_correlations for details.

V Data Retrieval, Processing and Storage Configuration.

In addition to the configuration of parameters of detector and analyzer algorithms as described above, **Config.py** also contains the configuration details on how data is retrieved, pre-processed and stored. The algorithm part of config.py is detailed in individual anomaly detector and Analyzer algorithms (Sections III and IV). This section contains the instruction to configure how data is retrieved, processed and stored.

Observables configurations are in form of Python dictionaries that configure how observables are initialized and updated. See below for all applicable keys in the dictionary. Note that only some of the keys are compulsory so not all of them will appear in a particular observable configuration.

Each key has an attribute Read or Write shown below. **Read** means that key is only used for look up. If the look up failed, either an error is raised or ignored or no data is changed. **Write** means that the value can be used to change data.

Some default config files are pre-loaded in config_files directory. To use one of these, copy the appropriate config file to the root directory and **change the file name to 'config.py'**. **If you do not change the file name, the detector will be unable to read the config file.**

Warning: Configuration here is for input data from perfmondb. For other types of data, some configurations may be different, or additional configuration on the wrapper may be required. See Section VI on how to configure those kinds of data!

- Master configuration (master_configs)

  DataType(str): **Read** 'Intraday' or 'Daily'. This impacts how time index is parsed. (Daily data does not have the time part in the datetime string, whereas intraday data either has both date and time, or time only)

  IdentifierColumn (str): **Read** Column name of identifier. Default is 'StrKey' if left blank. Must match the column name in database.

  TimeStampColumn (str): **Read** Column name of timestamp column. Default is 'Date' if left blank. Must match the column name in database.

  DefaultLogDirectory(str, path string): **Read/Write** the default directory where log files will be saved.

  DefaultPickleDirectory(str, path string): **Read/Write** the default directory where pickles (intermediate files, where the Anomaly Detector uses as input source) will be saved.

  SQLDefaultConnection: **Read** Default connection setting for the SQL database where data is to be fetched.
  Username (str), password (str), host (str, server address), port (str), database (str), table (str)

- Algorithm configurations

See Section III and Section IV

- Observable-specific configurations (observable_configs)

someObservable (Compulsory, str): **Read** Description of the observable. If data is retrived from database or csv, the name is arbitrary as long as it is consistent. If the data is parsed from raw data, the observable name must be in the list of observables in **Initializer_perfcoldb**.*Observables*.

  InputType (Compulsory, str): **Read** 'csv' or 'mysql' or 'synthesized'
  'Synthesized' means that the observable is not pulled as data from database directly, but computed from the raw transaction data.

  ID (Compulsory, str/int): **Read** Observable ID. This must match the ID in sql Database.

  ValueColumns (Compulsory, list of str): **Read** Column names of the variables to be monitored. Must match the name in sql database. e.g. ['Count', 'Sum', 'Average']. Do NOT enter column names for identifier column (default 'StrKey') and timestamp column (default 'Date') here.

csvFilePath (Compulsory for csv-type input, path str): **Read** Path of input csv file

csvDelimiter (Compulsory for csv-type input, single char): **Read** Delimiter. Example tab or semicolon

sqlStartDate (Optional, date str, ignored unless InputType is mysql): **Read** Start date of data retrieval from database if unspecified, all data will be retrieved for the first initialization of the observable, and subsequently sqlStartDate is set to be the last date in the previous record.

sqlEndDate (Optional, date str, ignored unless InputType is mysql): **Read** End date of data retrieval from database. If unspecified, the all data until latest available date will be retrieved.

AddRatio (Optional, list of tuples of two str): **Write** If non-blank, columns which is the ratio between columns specified by tuple elements will be added. E.g. [('A', 'B), ('C', 'D will add two columns that are ratio between A and B '_AOverB' and ratio between C and D '_CoverD'. Columns generated starts with _ to show it is a synthesised data. **Note:** synthesised data is by default excluded from pulse analysis. Only trend and changepoint analyses will be applied.

AddRatioToAggregate (Optional, list of str. requires non-blank 'AggregateStrKey'): **Write** If non-blank, program first locates the identifier of the aggregate item (e.g. 'All.All), then computes the ratio of each string in the list to the same column in the aggregate item. E.g. [‘A’, 'B'] generates two new columns '_AOverTotalA’ and '_BOverTotalB'

AggregateStrKey (Optional, str. Compulsory if AddRatioToAggregate is non-blank, otherwise this field is ignored): ***Read*** Identifier of the aggregate column. e.g. 'All', 'All.All.All'

VI. Wrappers

As mentioned in introduction, this tool is primarily for data with the data structure defined in PerfMon/perfmondb. Some wrappers are nevertheless provided for some other data structure to convert them into a compatible data structure.

To do so, you need to configure a file (Initailizer_xxxx.py) **in addition to** the usual config.py. See the guide below on how to configure these types of data.

1. Raw transactions (Data from PerfMon/perfcoldb)
Sometimes using raw transactions may be more desirable. For example, the Calculator module of PerfMon has finest time granularity of 1s. (I.e. at the finest scale, transactions happening within 1s are grouped and aggregated.) However, sometimes finer granularity may be desired and currently PerfMon does not provide that functionality. The wrapper module enables going to finer granularities such as 100ms, 50ms or even lower. In essence, the wrapper takes over the role of Calculator module in the PerfMon, but is able to handle finer time segmentation.

To parse raw data, go to modules/Initializer_perfcoldb.py. There are two dictionaries to configure how raw transactions data is retrieved and parsed.

Configurations:

input_source_config:

TimestampColumns (list of 2 str): data in perfcoldb has a crude and a fine timestamp. You need to provide names of both timestamp columns.

ValueColumns (list of n str): the column names of the data column whose data will be used to.

SQLDefaultConnection (dict): connection details where the **raw** data will be fetched

parsing_config:
Raw data needs to be parsed into data similar to the ones in PerfMon/perfmondb. This sets the parameters of the parsing process

SamplingFrequency (int), *SamplingFrequencyUnit* (str, 'ms' or’s’): sets the aggregation frequency. For e.g. if 100ms, transactions happening within 100ms will be grouped together, with statistics such as mean, median, 99th percentile, etc calculated

DestinationDirectory (str, path string): sets where the parsed data will be saved. Should ideally be a subdirectory in the pickles folder.

TimestampColumn (str): sets the time column name of the parsed data.

IdentifierColumn (str): sets the identifier column name of the parsed data.

Observables (list): sets the observables to be generated from raw data. The list must be a sub-set of the list in supported_observables in Initializer_perfcodb.py

Statistics (list): sets the statistics to be computed for each observable. The list must be a set-set of the list in suppored_statistics in Initializer_perfcoldb.py. Each statistics will be one column (Series) in the resulting data frame pickled.

Apart from the two dictionaries above, there are also two dictionaries where all supported_observables and supported_statistics are defined, where key is the name of observable or statistics, and value being how the observable or statistics are calculated. You may expand on that dictionary if you would like new observables and statistics.

After parsing, the parsed file will be deposited in the corresponding pickles folder in the same manner as the normal input data. These pickles are structurally identical to those fetched directly from perfmondb. You still need to set the parameters in config.py as you do for normal data, albeit there are some important differences. Here is the guide on how to set config.py for raw transaction data (where *as per normal* is indicated, there is no disagreement with the guides provided in Section V):

In config.py,
Master Configurations:

DataType: Since most of the time, the data parsed from perfcoldb is intraday, this should be 'Intraday'

IdentifierColumn, *TimestampColumn*: This must match exactly the value of the same key in **parsing_config**. NOT TimestampColumns in input_source_config.

DefaultPickleDirectory: This must match exactly the value of key DestinationDirectory in **parsing_config**

DefaultLogDirectory: As per normal.

SQLDefaultConnection: This is not used if the pickles are parsed from raw data, rather than pulled directly from database. So anything here is not used.

Algorithm configurations:

As per normal.

Observable_specific configuration:

SomeObservable:

As per normal. However, since observables for the raw data are generated by the the parser rather than pulled from the database, the values here must be properly defined in parsing_config.Observables. You do not need to enter all observables defined in **parsing_config**.*Observables*, but every observable appearing here must be defined in that file.

ValueColumns:

As per normal. Again, since these columns are generated by the parser, the column names here must match those defined in **parsing_configs**.*Statistics*.

CsvFilePath, csvDelimiter, sqlStartDate, sqlEndDate:

Since the data here are neither read from csv file nor the database directly. These fields are ignored. You can leave it blank.

Other keys:

As per normal.

2. Creating New Wrappers

Any data can be processed as long as it is converted into compatible data structures. This section provides a template of the target data structure that is compatible with the Anomaly Detector.

Data Type: Pandas DataFrame

Columns:

1. Timestamp: Can be any standard datetime format as long as it is recognized by and convertible to a pandas datetime index. This is the index column. The column name needs to match exactly 'TimestampColumn' in config.py -> master_config. Appropriate data type ('Daily', 'Intraday') needs to be entered into 'DataType' entry in master_config of config.py
2. Identifier: A text string identifying the data frame. Column name needs to match exactly 'IdentifierColumn' in master_config in config.py. If the identifier contains multiple layers, they need to be joined by '.'.
3. Value Columns: The actual statistics that will be analysed (e.g. Average, Count). New statistics that are functions of existing ones may be added.
   Note: by adding _ before the name of the column names here, the statistics will by default be excluded from Pulse Detectors.

   The dataframe should be saved as a pickle (.pkl) file. The path the dataframe is saved to needs to match exactly 'DefaultPickleDirectory' entry in master_config of config.py

Example:

Date	SampleIdentiferTxtStr	SampleStat1	SampleStat2	_SampleStat3
2017-05-02	ABCD.EFGH	12	12	1
2017-05-03	ABCD.EFGH	13	21	0.62
2017-05-02	HIJK.LMNO	100	200	0.50

VII. Interpreting Outputs

This section contains sample visual and textual output from the Detector and Analyzer, and outlines how these output should be interpreted.

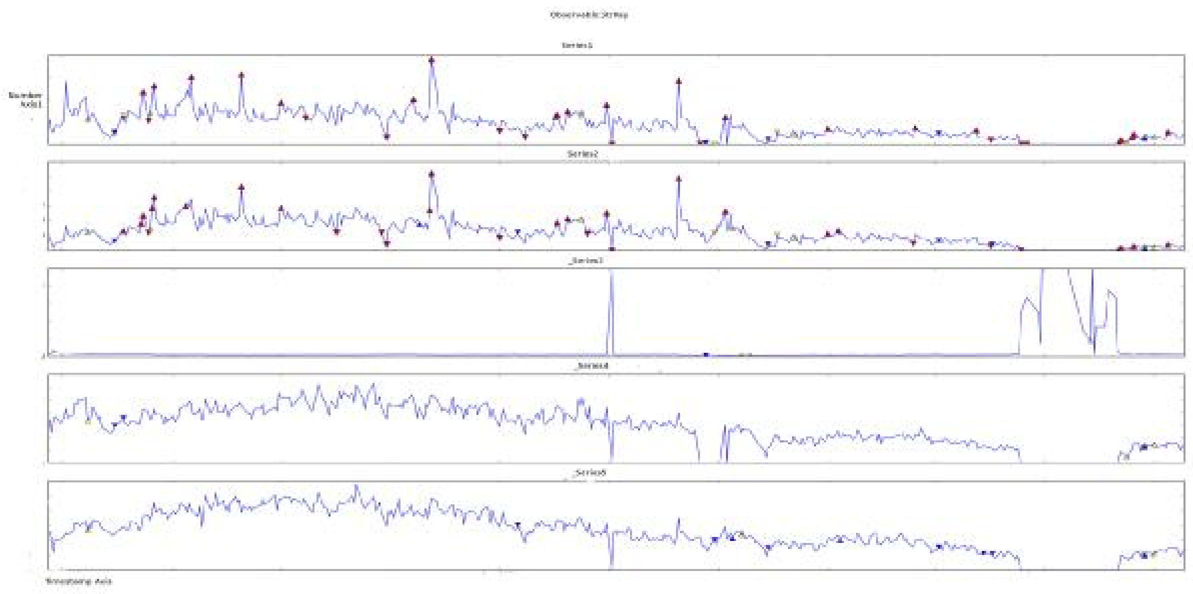
- **Anomaly Detector**

Textual

Date	StrKey	A	B	_AoverB	_AoverTotalA	_BoverTotalB	Result_A_PULSE	Result_A_PULSE	Result_B_MEAN_CHANGE	Result_A_MEAN_CHANGE	Result__BOverA_MEAN_CHANGE
2015-11-23	ABCD.EFGH	1131	56332	0.0200773983	0.0104190657	0.0782806386					
2015-11-24	ABCD.EFGH	1739	94379	0.0184257091	0.0107340378	0.0742155309					
2015-11-25	ABCD.EFGH	1355	68969	0.0196465079	0.0104919201	0.0715931763					
2015-11-26	ABCD.EFGH	1540	36972	0.0416531429	0.0197109908	0.0615965933					
2015-11-27	ABCD.EFGH	1771	59080	0.0299763033	0.0153045793	0.067278183					
2015-11-30	ABCD.EFGH	2372	80517	0.0294596172	0.01848936	0.0822924598					
2015-12-01	ABCD.EFGH	2187	83046	0.0263348024	0.0152461553	0.0719239656					
2015-12-02	ABCD.EFGH	2242	78943	0.0284002381	0.0156646288	0.0673417648					
2015-12-03	ABCD.EFGH	5255	183020	0.028712709	0.0160169954	0.0692577853					
2015-12-04	ABCD.EFGH	2303	86451	0.026639368	0.0095524889	0.0480445884					
2015-12-07	ABCD.EFGH	2057	82699	0.0248733358	0.0123580655	0.0633691331					
2015-12-08	ABCD.EFGH	2675	104613	0.0255704358	0.0155738755	0.0714227194					
2015-12-09	ABCD.EFGH	2370	95693	0.0247667018	0.0139322434	0.0635230313					
2015-12-10	ABCD.EFGH	2037	78840	0.0258371385	0.0156603164	0.0652909522					
2015-12-11	ABCD.EFGH	2973	116220	0.025580795	0.0160946297	0.0679662637					
2015-12-14	ABCD.EFGH	3805	138124	0.0275477108	0.0182520267	0.0763754876	^	^			
2015-12-15	ABCD.EFGH	2696	145070	0.0185841318	0.0160775731	0.0810974287					
2015-12-16	ABCD.EFGH	1646	92208	0.0178509457	0.0108218277	0.0624387179					
2015-12-17	ABCD.EFGH	2548	128540	0.0198226233	0.0144443626	0.0743930749					
2015-12-18	ABCD.EFGH	2389	122764	0.0194601023	0.0151355803	0.0760937343					
2015-12-21	ABCD.EFGH	1923	105507	0.0182262788	0.0154490094	0.0901753816					
2015-12-22	ABCD.EFGH	1846	108365	0.0170350205	0.01490248	0.0854609521					
2015-12-23	ABCD.EFGH	1662	99621	0.0166832294	0.016686747	0.099197427					
2015-12-28	ABCD.EFGH	1056	56742	0.018610553	0.0148143991	0.0911504894		v			
2015-12-29	ABCD.EFGH	1276	71332	0.0178881848	0.0144736842	0.0795567343					
2015-12-30	ABCD.EFGH	505	29685	0.0170119589	0.0058624813	0.0343772546		v	v		
2016-01-04	ABCD.EFGH	2799	149189	0.0187614368	0.0150735908	0.0820772353	^	^			

Default location: root dirctory → *DefaultLogDirectory* defined in config.py (e.g. 'output') → observable name (e.g. 'sample_observable_name') → string key.csv (e.g. 'abcd.csv')
If you prefer the Detector output organized by timestamp rather than by identifier, use Analyzer.get_anomaly_records (See documentation of that function in Section VII)

Visual:



When Detector.detect_frames (plot=True) or Analyzer.plot_from_log() is used, a graph similar to the one above is generated.

Legends:
Upwards pulse: ▲
Downwards pulse: ▼
Upwards Mean Change: ▲
Downwards Mean Change: ▼
Upwards trend: ▲
Downwards trend: ▼

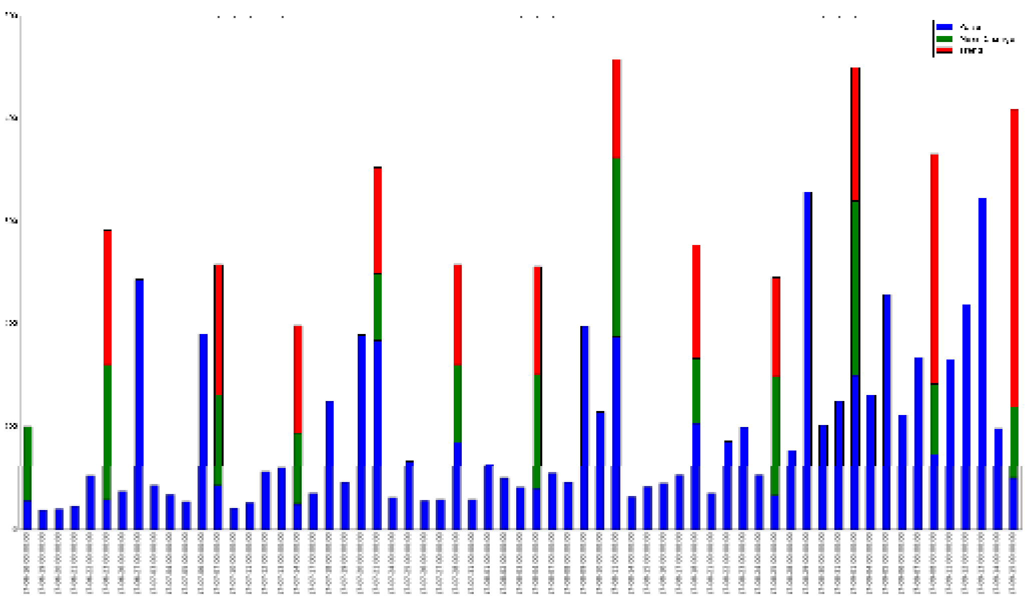
Note: It is possible that not all types of symbol defined above exist in the graph and that depends on the configuration and the data itself.

In Analyzer.get_anomaly_records, you may group the anomaly occurrences against time-axis. Sample textual and visual output are presented:

Textual:

```
17/06/2017        1  SampleObservable1_SampleStrKey1_Count_PULSE_UP
17/06/2017        2  SampleObservable1_SampleStrKey1_Count_MEAN_CHANGE_UP
17/06/2017        3  SampleObservable2_SampleStrKey2_Ratio_TREND_DOWN
18/06/2017        1  SampleObservable3_SampleStrKey1_Count_PULSE_UP
18/06/2017        2  SampleObservable1_SampleStrKey2_Count_MEAN_CHANGE_UP
18/06/2017        3  SampleObservable2_SampleStrKey2_Sum_TREND_DOWN
```

Visual:



- **Associator Output**

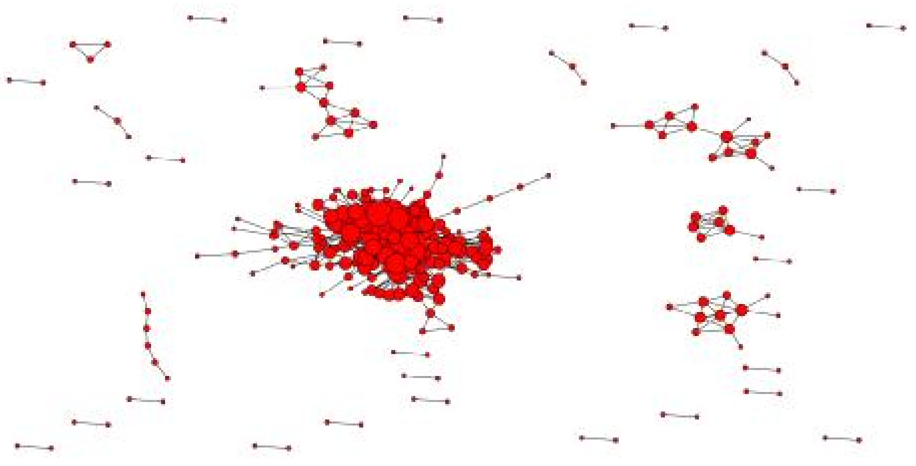
Textual:
Legends:
Set, 2-tuple: the 2-frequent pattern found. E.g. (A, B)
Support (int > 0): The number of times the patterns occur in all observations.
Confidence1, Confidence2 [0, 1]: The conditional probability P (A|B) and P (B|A) respectively.
MaxConfidence (Confidence1, Confidence2): The larger confidence out of the two above.

Lift □ [0, inf): Value of P (A^B) / (P (A) * P(B)). Lift = 1 suggests independence, Lift < 1 suggests negative correlation and Lift > 1 suggests positive correlation.
Caution: The reliability of Lift as a measure of interestingness is not ideal when the support is low. For example, if both X and Y are rare, a single occurrence of (X, Y) by coincidence can lead to a misleadingly large lift number. Hence, a large lift number does not necessarily suggest strong correlation. Lift and Support should be assessed holistically.

Set	Support	Lift	Confidence1	Confidence2	MaxConfidence
['SampleObservable_Member1_Result_SampleSeries1_PULSE_UP', 'SampleObservable_Member1_Result_SampleSeries2_PULSE_UP']	31	9.3276785714	0.8857142857	0.96875	0.96875
['SampleObservable_Member2_Result_SampleSeries1_PULSE_UP', 'SampleObservable_Member2_Result_SampleSeries2_PULSE_UP']	27	8.1096256684	0.7941176471	0.8181818182	0.8181818182
['SampleObservable_Member3_Result_SampleSeries1_PULSE_UP', 'SampleObservable_Member3_Result_SampleSeries2_PULSE_UP']	27	6.8413533835	0.7105263158	0.7714285714	0.7714285714
['SampleObservable_Member4_Result_SampleSeries1_PULSE_UP', 'SampleObservable_Member4_Result_SampleSeries2_PULSE_UP']	25	6.8831699346	0.6944444444	0.7352941176	0.7352941176
['SampleObservable_Member5_Result_SampleSeries1_PULSE_UP', 'SampleObservable_Member5_Result_SampleSeries2_PULSE_UP']	24	8.1862348178	0.6315789474	0.9230769231	0.9230769231
['SampleObservable_Member6_Result_SampleSeries1_PULSE_UP', 'SampleObservable_Member6_Result_SampleSeries2_PULSE_UP']	24	8.425	0.75	0.8	0.8
['SampleObservable_Member6_Result_SampleSeries1_PULSE_UP', 'SampleObservable_Member6_Result_SampleSeries2_PULSE_UP']	24	6.8080808081	0.6666666667	0.7272727273	0.7272727273
['SampleObservable_Member7_Result_SampleSeries2_PULSE_UP', 'SampleObservable_Member7_Result_SampleSeries1_PULSE_UP']	22	11.4061538462	0.8461538462	0.88	0.88
['SampleObservable_Member3_Result_SampleSeries1_PULSE_UP', 'SampleObservable_Member4_Result_SampleSeries1_PULSE_UP']	22	5.4195906433	0.5789473684	0.6111111111	0.6111111111
['SampleObservable_Member8_Result_SampleSeries2_PULSE_UP', 'SampleObservable_Member8_Result_SampleSeries1_PULSE_UP']	22	7.8042105263	0.5789473684	0.88	0.88
['SampleObservable_Member9_Result_SampleSeries1_PULSE_UP', 'SampleObservable_Member9_Result_SampleSeries2_PULSE_UP']	22	8.8578255675	0.7096774194	0.8148148148	0.8148148148
['SampleObservable_Member10_Result_SampleSeries1_PULSE_UP', 'SampleObservable_Member10_Result_SampleSeries2_PULSE_UP']	22	8.5810185185	0.6875	0.8148148148	0.8148148148
['SampleObservable_Member6_Result_SampleSeries1_PULSE_UP', 'SampleObservable_Member9_Result_SampleSeries1_PULSE_UP']	21	6.3413978495	0.5833333333	0.6774193548	0.6774193548
['SampleObservable_Member11_Result_SampleSeries1_PULSE_UP', 'SampleObservable_Member11_Result_SampleSeries2_PULSE_UP']	21	11.3413461538	0.8076923077	0.875	0.875
['SampleObservable_Member3_Result_Count_PULSE_UP', 'SampleObservable_Member4_Result_Count_PULSE_UP']	21	5.9470588235	0.6	0.6176470588	0.6176470588
['SampleObservable_Member12_Result_SampleSeries1_PULSE_UP', 'SampleObservable_Member12_Result_SampleSeries2_PULSE_UP']	20	9.9851851852	0.7407407407	0.8	0.8
['SampleObservable_Member4_Result_SampleSeries1_PULSE_UP', 'SampleObservable_Member3_Result_SampleSeries2_PULSE_UP']	20	5.3492063492	0.5555555556	0.5714285714	0.5714285714

Each 'Set' is a list of two anomaly identifiers, in form of
"observable_StringKey_SeriesName_TypeofAnomaly_UpOrDown"
See Section IV for the meaning of the columns.

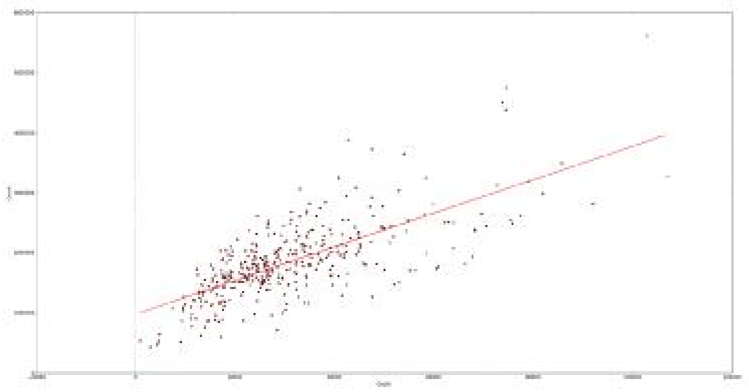
Visual

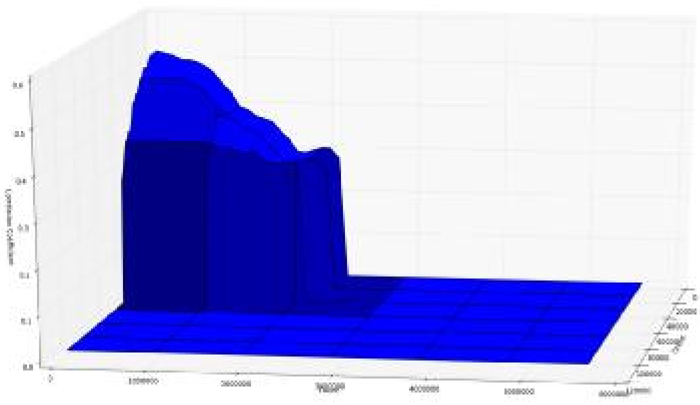


Note: Anomaly identifiers, which appear as labels of the individual nodes, are redacted for this illustration.
Each node represents an anomaly identifier in the form of
"observable_StringKey_SeriesName_TypeofAnomaly_UpOrDown"
Larger nodes suggest these anomalies happen more often. Edges between nodes suggest association. Stronger line (hardly visible for this illustration) suggests stronger relationship as measured by confidence and lift.

· **Correlator**:

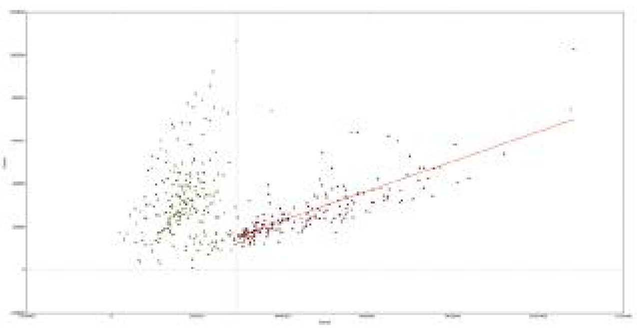
Example 1: Output from the data without floor or ceiling effect

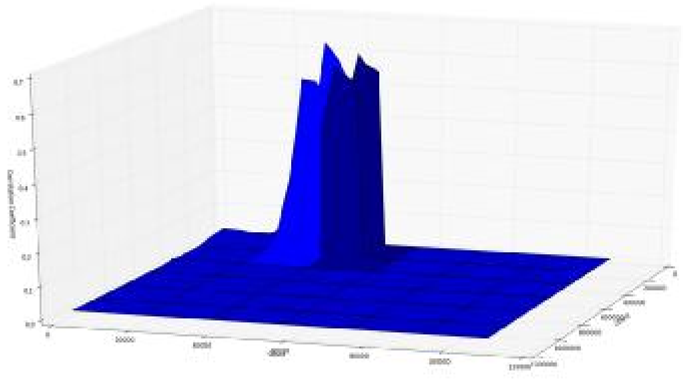




With all (or almost all) data follow the correlation, we can see
generally correlation coefficient reduces with truncation and best correlation coefficient occurs when there is no truncation. (At higher range the correlation coefficient is reduced to 0 because the number of points above that threshold is too few – algorithm does not consider these cases as they may give meaningless high R-value (e.g. R-value of two data points is always 1))

Example 2: Output from data with floor effect – strong correlation observed only above certain threshold





Visual inspection can tell that despite the low overall R-value (r = 0.15), above certain threshold of x-value the correlation is stronger. The algorithm gives a rough estimate of that threshold and for this case, the red points (points considered in the red regression line) has r = 0.79. Also, from the surface plot, we can see that the highest r occurs when there is some truncation.

VIII. Callable Functions

This section contains the documentation of the functions that should be called by users directly.

As a rule of thumb, the parameters of these functions are generally filters only. Observable and Algorithm specific configurations should be done at config.py

- Initialization – Initialize or update data from database**:**

  Initializer**.update**(observable_filter=None, flush=False)
  Fetches new data for observables configured in config.py.

Parameters	**observable_filter**: List/str/None. If list, only observables within the list will be updated. If str, only observables matching the provided regex filter will be updated. If None, all observables configured will be updated **flush**: boolean If true, old data is deleted and replaced with new data. Otherwise new data will be concatenated to the old data. Config.py The behaviour of the function is also controlled by master_config and observable specfic configs in config.py See sections III and V on how to configure config.py.
Returns	None Relevant data will be pickled into local disk according to the setting in config.py

  Initializer**.get_monitored_observables_list**()
  Fetches the list of configured observables in config.py

Parameters	None
Returns	List of configured observables.

  Initializer_perfcoldb.**update**(flush=False,)
  When the input data is from perfcoldb rather than the usual perfmondb, use this initializer **in-lieu** of the normal one to parse the data into compatible data structure.

Parameters	**flush**: boolean If true, old data is deleted and replaced with new data. Otherwise new data will be concatenated to the old data. input_source_config, parsing_config in modules/Initializer_perfcoldb.py * These are two dictionaries under initializer_perfcoldb.py. See Section VI on how to configure these two dictionaries.
Returns	None Relevant data will be pickled into local disk according to the settings in config.py and Initializer_perfcoldb.py.

- Detection - detect anomalies, either by string key or by date**:**

  Detector. **detect_frames**(observable_filter=None, frame_filter=None, series_filter=None, only_common_identifiers=False, only_n_largest=None, only_longer_than=None, only_more_than=None, time_from=None, time_to=None, trading_hours_only=True, plot=False, output_log=True)
  Detects anomalies in single or multiple data frame across all available timestamps.
  Warning: Although time range filtering is provided, if you are interested in anomalies only on one or few time

stamps (e.g. on one day only), you should not use this function by limiting the time_from, time_to arguments to a very small range because that will deprive the detector accessing historical data. This will deteriorate detector performance and if the time range is too small (smaller than sliding window size of individual detectors), stops detectors from working altogether. Instead, you should use Detector.**detect_timestamp** function which is specifically designed for this purpose.

Note:

1. If plot is True, this function generates graph but does not automatically plots it immediately. Insert plt.show() command to show graph(s).
2. Turning plot to True may slow down the computing speed. The recommended approach is to process the frames first, and then use Analyzer.**plot_from_log** to view the visualization.

Parameters	**observable_filter**: list/str/None If list, only observables within the list will be updated. If str, only observables matching the provided regex filter will be updated. If None, all observables will be included. **frame_filter**: list/str/None If list, only frames within the list will be updated. If str, only frames matching the provided regex filter will be updated. If None, all frames will be included.. e.g. “^(ABCD.EFGH)$” e.g. ['ABCD.EFGH', 'IJK.LMN'] **series_filter:** list/str/None If list, only series within the list will be updated. If str, only series matching the provided regex filter will be updated. If None, all series will be included. e.g. ['A', 'B'] **only_common_identifiers:** bool If true, only the frames with string keys defined in config_common_identifiers will be updated. **only_n_largest**: 3-list [int, str (series name), str (timestamp)] Frame filter. e.g. [5, 'A', '2017-01-03'] analyses only 5 largest frames based on 'A' on 2017-01-03 e.g. [5, 'A', None] analyses 5 largest frames based on average 'A' across all available dates **only_longer_than**: int Frame filter. Only consider frames with data points more than this value **only_more_than:** 3-list [int/float, str (series name), str (timestamp)/None] Frame filter. e.g. [0.1, 'C', '2017-01-03']: analyses frames whose 'C' on '2017-01-03' is more than 0.1 e.g. [0.1, 'C', None]: analyses frames whose 'C' are more than or equal to 0.1 on any date. **time_from, time_to**: str, datetime string. If daily data, datetime string accepted is 'YYYY-mm-dd' If intra-day data, datetime string accepted is 'YYYY-mm-dd hh:mm:ss' or 'hh:mm:ss' Frame filter. Filters based on time range. **trading_hours_only**: boolean If true, only datapoints within trading hours (0900 – 1700) will be considered. This argument is ignored if the data is not intra-day. This argument can be overridden if time_from and/or time_to are set. **plot**: boolean If true, a graph containing original data and detected anomalies will be shown

	output_log: boolean If true, the output log containing same information will be produced. The log file is the name of the frame string key.
Returns	None

Detector. **detect_timestamp**(timestamp, observable_filter=None frame_filter=None, only_common_identifiers=None, only_n_largest=None, only_longer_than=None, only_more_than=None, series_filter=None, output_log=True)
Detects anomalies in all or selected data frames across a single timestamp.
Note: If you prefer to view the anomalous results grouped by timestamp rather than by string key or other identifier, you may use Analyzer. **get_anomaly_records** with time_from and time_to arguments set to switch to that view.

Parameters	**timestamp**: datetime string The timestamp interested. If daily data, datetime string accepted is 'YYYY-mm-dd' If intra-day data, datetime string accepted is 'YYYY-mm-dd hh:mm:ss' or 'hh:mm:ss' **observable_filter, frame_filter**, **series_filter, only_common_identifiers, only_n_largest**, **only_longer_than**, **only_more_than**: Same as above **output_log**: boolean If true, the output log containing same information will be produced. If the record for the current string key is already present, the new data will be concatenated to the existing csv file. Otherwise, a new csv file with name same as the string key and a single row will be created under the usual output directory.
Returns	None

- Assembler

Each of the pulse algorithms return a decision whether a point is deemed as anomalous, and assembler takes output of all the base detectors to generate a unified decision. 'vote' simply returns a point as anomalous if more than half of the base detectors deem so, SVM on the other hand is able to learn from user feedback and adjust the weight of base detectors.

Detector.**init_assembler**(observable, basis_assembler='vote', max_train_cnt=0,)
Initialize a SVM instance for an observable. The SVM is initialized to imitate another assembler ('basis_assmbler') using training data.

Parameters	**observable**: str Name of observable. **basis_assembler**: str The type of basis assembler chosen. Note that this assembler must be one of the assemblers defined in config. **max_train_cnt**: int The number of frames to be used as training data. Exact frames to be used for SVM training will be selected at random of the pool of frames of the observable.
Returns	None A SVM instance will be saved in the pickle directory, with file name *observable*+SVM.pkl

Detector. **train_assembler**(observable, train_data_path=None)
Train an initialized SVM instance for the observable from a user-feedback csv file.

Parameters	**observable**: str Name of observable. Train_data_path: str (path string) File path of the training csv file
Returns	None A trained SVM instance will be saved in the pickle directory

· Analyzer **–** detect associations between anomalies across frames, series and observables**:**
Analyzer. **get_anomaly_records**(print_on_screen=False, **kwargs):
Fetches and displays anomaly records with user customized filters (time range, observable, frame and series filter). The result is grouped by timestamp. If you prefer result grouped by string key, go directly to the output directory and view the output csv files. You may also use Analyzer.plot_from_log to generate visualization.

Parameters	**print_on_screen**: boolean If true, the output will be print on screen. ****kwargs**: Optional keyword arguments: **AnomalyDetector_output_dir**: str (String path) The path of the output directory of anomaly detector (analyzer takes output from detector as input) **time_from, time_to**: str (timestamp string) Filter based on range of time. If daily data, datetime string accepted is 'YYYY-mm-dd' If intra-day data, datetime string accepted is 'YYYY-mm-dd hh:mm:ss' or 'hh:mm:ss' **observable_filter, frame_filter, series_filter** (list or str) Same as the meaning in Detector. Plot (bool): If True, number/composition of anomaly occurences will be plotted against appropriate time axis. *Passing other kwargs defined in the Associator class will be accepted (See get_associations function). However, this function merely fetches the anomaly records, so these additional arguments are ignored.
Returns	Pandas DataFrame containing anomaly records

Analyzer.**get_associations**(print_on_screen=False, **kwargs):
Find associations between anomalies across frames, series and observables.
Note:
1. This function finds associations of **anomalous points only.** Through this, the number of points analysed is reduced drastically thereby allowing a breadth-first search across a large amount of data. get_correlations function, on the other hand, considers all points.
2. If plot is True, this function generates graph but does not automatically plots it immediately. Insert plt.show() command to show graph(s).

Parameters	**print_on_screen**: boolean If true, the output will be print on screen. ****kwargs**: Optional keyword arguments:

	AnomalyDetector_output_dir: str (String path) The path of the output directory of anomaly detector (analyzer takes output from detector as input) **time_from, time_to**: str (timestamp string) Filter based on range of time. If daily data, datetime string accepted is 'YYYY-mm-dd' If intra-day data, datetime string accepted is 'YYYY-mm-dd hh:mm:ss' or 'hh:mm:ss'. **observable_filter, frame_filter, series_filter** (list or str) Same as the meaning in Detector. **min_support, min_lift, min_conf** (int or float) Minimum support, lift and confidence for frequent pattern generation (see Section IV) **exclude** (list of strs or tuples of 2 strs) e.g. exclude=[('A', 'B'), 'C'] *Exclude* is a parameter that **excludes** anomaly identifiers* that matches the regex expression in the string or the tuple of strings. The difference between single string and tuple of 2 strings (string A and string B) is as below: Single string: Each frequent pattern consists of two anomaly identifiers (identifier A and identifier B). If **any** of the two identifiers match the single string, that pattern is excluded. (For the example above, any frequent pattern containing 'C' will be excluded) Tuple of 2 strings: Only if identifier A matches string A **and** identifier B matches string B, **or** if identifier A matches string B **and** identifier B matches string A, that frequent pattern is excluded (For the example above, if both identifiers of the frequent pattern contain 'Count', that frequent pattern is excluded) If the length of list passed to *exclude* has a length more than one, frequent patterns satisfying **any** condition specified in the list will be excluded. **include** (list of strs or tuples of 2 strs) *Include* works the same way as *exclude*, with the **only** difference that only frequent patterns matching the regex expression will be **included**. Any other patterns are discarded. Again, if the length of list passed to include has a length more than one, the frequent patterns only need to match one of the conditions to be included. ***anomaly identifiers** are strings representing a type of anomaly, in the format of Observable_StrKey_Series_AnomalyType_UporDOWN. A frequent set is a pair of two such strings. **plot**: bool If True, a network graph showing the correlation relationship between the series or frames (depending on correlate_at_frame_label setting) will be shown.
Returns	Pandas DataFrame with Analyzer output.

Analyzer.**get_correlations**(observable_filter1, frame_filter1, series_filter1, observable_filter2, frame_filter2, series_filter2, **kwargs)
Compute correlation between two series
Note: This function correlates the all points and is more computationally expensive so it is not suitable for breadth-first search of correlations (hence observable, frame and series are required exactly rather than a regex filter). It is suitable as a more in-depth exploration after the get_associations returns a potentially interesting association between two series.

Parameters	**observable_filter1, frame_filter1, series_filter1**:

	The names of observable, frame and series. The series specified should be unique. **observable_filter2, frame_filter2, series_filter2**: Same as above ****kwargs**: optional keyword arguments: **precision**: int The data range of the two series will be bootstrapped into a meshgrid of ranges and linear regression is run on each of the mesh to find the optimal r-value for linear regression. This parameter defines the granualarity of the meshgrid **min_point_count**: int or float [0, 1]: This defines the minimum number of points needed to be fit for linear regression. If int, this parameter is interpreted literally. If float, this parameter is interpreted as the proportion of the total number of data points. **plot**: bool If True, a scatter plot with optimal truncation boundary (if any) and a linear regression line will be generated. **plot_surface**: bool If True, a surface plot of how r-squared value of linear regression varies with series1 and series2 will be generated. **correlation_metric**: str, ('pearson', 'spearman', 'kendall') Select the correlation metric to determine the strength of correlation. Defult is Spearman
Returns	None, graphs will be plotted if appropriate plot flags are True, Summary of the result is printed on-screen

Analyzer. **plot_from_log**(observable, str_key, file_path=None, date_from=None, date_to=None)
Generate visualization of data and anomalies detected from log directly. It generates the same graph during anomaly detection if the plot flag is on, but this function does not re-compute anomalies from detector algorithms
Note: this function generates graph but does not automatically plots it immediately. Insert plt.show() command to show graph(s).

Parameters	**observable**: str Name of observable. This is not a regex expression and needs to match an actual observable name exactly. e.g. 'SampleObservable' **str_key**: str String key of the frame. This is not a regex expression and needs to match an actual string key exactly. e.g. 'ABCD.EFGH' **file_path**: str (path string) File path of the log file. If None, the file_path will be the default one defined in config.py. **scale**: str ('linear', 'log') Axis type of the graph (linear-linear or semilog y). Note that x-axis is time axis so this setting does not affect it. **time_from, time_to**: str, datetime string The range of time where graph will be plotted.

	If daily data, datetime string accepted is 'YYYY-mm-dd' If intra-day data, datetime string accepted is 'YYYY-mm-dd hh:mm:ss' or 'hh:mm:ss'
Returns	None

IX. Reference

Papers:

1. Carter, Kevin M., and William W. Streilein. "Probabilistic reasoning for streaming anomaly detection." Statistical Signal Processing Workshop (SSP), 2012 IEEE. IEEE, 2012.

2. Micenková, Barbora, Brian McWilliams, and Ira Assent. "Learning Outlier Ensembles: The Best of Both Worlds—Supervised and Unsupervised." Proceedings of the ACM SIGKDD 2014 Workshop on Outlier Detection and Description under Data Diversity (ODD2). New York, NY, USA. 2014.

3. Vallis, Owen, Jordan Hochenbaum, and Arun Kejariwal. "A Novel Technique for Long-Term Anomaly Detection in the Cloud." HotCloud. 2014.

4. Wei, Li, et al. "Assumption-Free Anomaly Detection in Time Series." SSDBM. Vol. 5. 2005.

5. Aggarwal, Charu C., and Saket Sathe. "Theoretical foundations and algorithms for outlier ensembles." ACM SIGKDD Explorations Newsletter 17.1 (2015): 24-47.

6. Burnaev, Evgeny, and Vladislav Ishimtsev. "Conformalized density-and distance-based anomaly detection in time-series data." arXiv preprint arXiv:1608.04585 (2016).

7. Christodoulou, Vyron, and Yaxin Bi, "A combination of CUSUM-EWMA for Anomaly Detection in time series data." Data Science and Advanced Analytics (DSAA), 2015. 36678 2015. IEEE International Conference on. IEEE, 2015.

8. N. Abbas, M. Riaz and R. J. Does, "Mixed exponentially weighted moving average cumulative sum charts for process monitoring" Quality and Reliability Engineering International, vol. 29, no. 3, pp. 345-356, 2013

9. Drápela, Karel, and Ida Drápelová. "Application of Mann-Kendall test and the Sen's slope estimates for trend detection in deposition data from Bílý Kříž (Beskydy Mts., the Czech Republic) 1997-2010." Beskydy 4.2 (2011): 133-146.

Open-source repositories:

LinkedIn Luminol
Numenta Anomaly Benchmark
Etsy Skyline
Twitter BreakoutDetection and AnomalyDetection
Enaeseth Python-fp-growth
Lytics anomalyzer