Luminaire: A hands-off Anomaly Detection Library

version

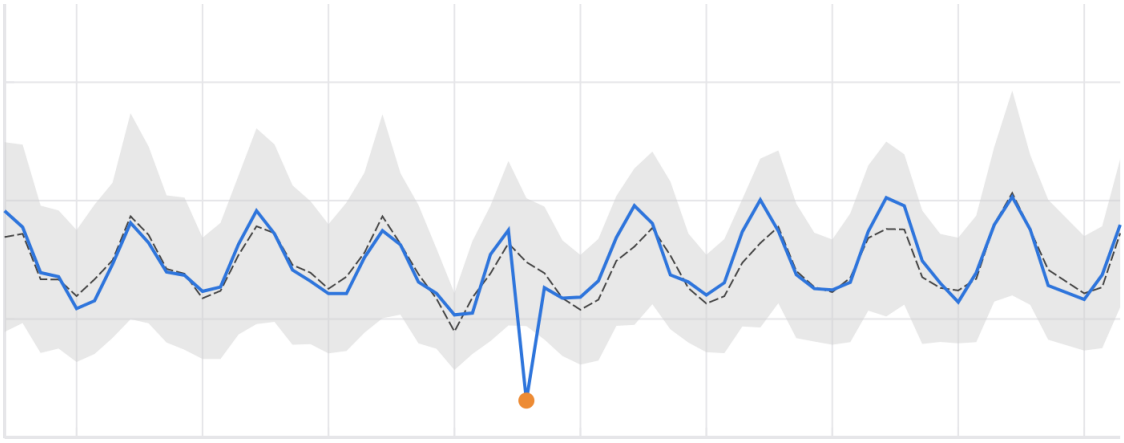
Zillow Group Data Governance team

July 29, 2020

Contents

Luminaire: A hands-off Anomaly Detection Library

Introduction



Luminaire is a python package that provides ML driven solutions for monitoring time series data. Luminaire provides several anomaly detection and forecasting capabilities that incorporate correlational and seasonal patterns in the data over time as well as uncontrollable variations. Specifically, Luminaire is equipped with the following key features:

- **Generic Anomaly Detection:** Luminaire is a generic anomaly detection tool containing several classes of time series models focused toward catching any irregular fluctuations over different kinds of time series data.
- **Fully Automatic:** Luminaire performs optimizations over different sets of hyperparameters and several model classes to pick the optimal model for the time series under consideration. No model configuration is required from the user.
- **Supports Diverse Anomaly Detection Types:** Luminaire supports different detection types:
    - Outlier Detection
    - Data Shift Detection
    - Trend Change Detection
    - Null Data Detection
    - Density comparison for streaming data

Data Exploration and Profiling

Luminaire performs exploratory profiling on the data before progressing to optimization and training. This step provides batch insights about the raw training data on a given time window and also enables automated decisions regarding data pre-processing during the optimization process. These tests and pre-processing steps include:

- Checking for recent data shifts
- Detecting recent trend changes
- Stationarity adjustments
- Imputation of missing data

Outlier Detection

Luminaire generates a model for a given time series based on its recent patterns. Luminaire implements several modeling techniques to learn different variational patterns of the data that ranges from ARIMA, Filtering Models, and Fourier Transform. Luminaire incorporates the global characteristics while learning the local patterns in order to make the learning process robust to any local fluctuations and for faster execution.

Configuration Optimization for Outlier Detection Models

Luminaire combines many techniques under hood to find the optimal model for every time series. Hyperopt is used at its core to optimize over the global hyperparameters for a given time series. In addition, Luminaire identifies whether a time series shows exponential characteristics in terms of its variational patterns, whether holidays have any effects on the time series, and whether the time series shows a long term correlational or Markovian pattern (depending on the last value only).

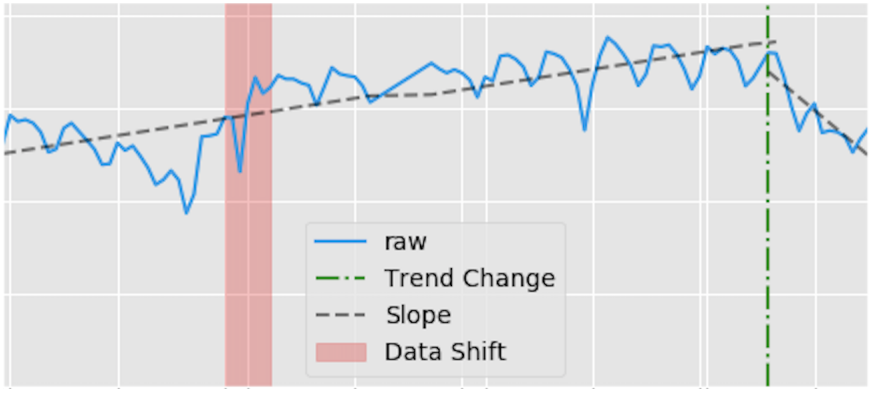
Anomaly Detection for Streaming Data

Luminaire performs anomaly detection over streaming data by comparing the volume density of the incoming data stream with a preset baseline time series window. Luminaire is capable of tracking time series windows over different data frequencies and is autoconfigured to support most typical streaming use cases.

Tutorials

Data Profiling

Luminaire *DataExploration* implements different exploratory data analysis to detect important information from time series data. This method can be used to impute missing data, detect the set of historical trend changes and change points (steady data shifts) which information can later be leveraged downstream in Luminaire outlier detection models.



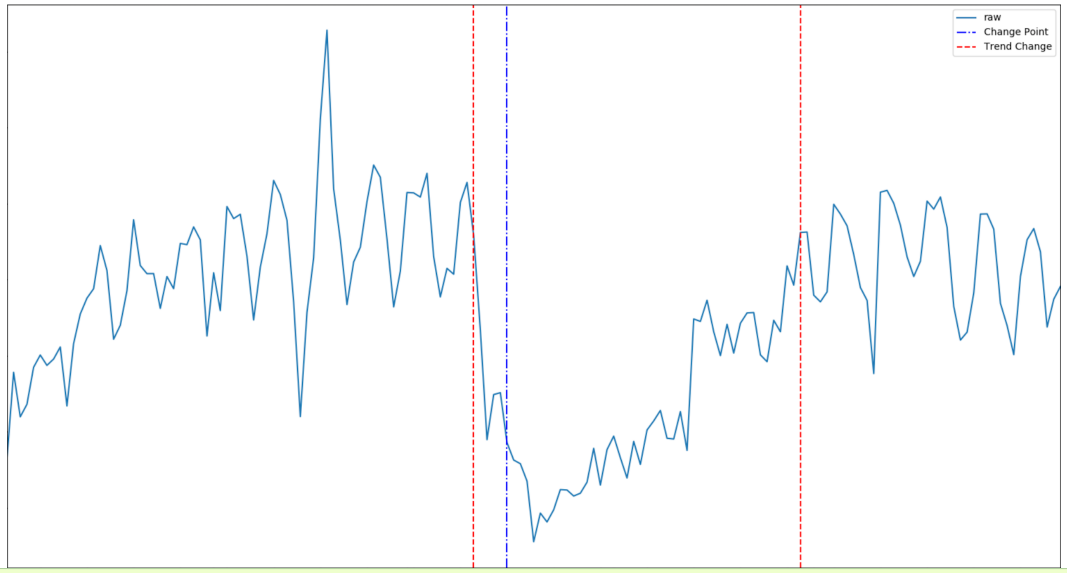
Luminaire data exploration and profiling runs two different workflows. The impute only option in profiling performs imputation for any missing data in the input time series and does not run any profiling to generate insights from the input time series.

```
>>> data
                raw
index
2020-01-01  1326.0
2020-01-02  1552.0
2020-01-03  1432.0
2020-01-04  1470.0
2020-01-05  1565.0
...            ...
2020-06-03  1934.0
2020-06-04  1873.0
2020-06-05     NaN
2020-06-06  1747.0
2020-06-07  1782.0
>>> de_obj = DataExploration(freq='D')
>>> imputed_data, pre_prc = de_obj.profile(data, impute_only=True)
>>> print(imputed_data)
                    raw
2020-01-01  1326.000000
2020-01-02  1552.000000
2020-01-03  1432.000000
```

```
2020-01-04  1470.000000
2020-01-05  1565.000000
...                 ...
2020-06-03  1934.000000
2020-06-04  1873.000000
2020-06-05  1823.804535
2020-06-06  1747.000000
2020-06-07  1782.000000
>>>print(pre_prc)
None
```

In order to get the data profiling information, the impute only option should be disabled (that is the default option). Disabling the impute only option allows Luminaire to impute missing data along with detecting all the trend changes and the change points in the input time series.

The key utility of Luminaire data profiling is this being a pre-processing step for outlier detection model training. Hence, the user can enable several option to prepare the time series before ingested by the training process. For example, the log transformation option can be enabled for exponential modeling during training. User can also check for the fill rate to constrain the proportion of missing data upto some threshold. Moreover, the pre processed data can also be truncated if there is any change points (data shift) observed.




```
>>> de_obj = DataExploration(freq='D', data_shift_truncate=True, is_log_transformed=True, fi
>>> imputed_data, pre_prc = de_obj.profile(data)
>>> print(pre_prc)
{'success': True, 'trend_change_list': ['2020-04-01 00:00:00'], 'change_point_list': ['2020-
>>> print(imputed_data)
               raw  interpolated
2020-03-16  1371.0      7.224024
2020-03-17  1325.0      7.189922
2020-03-18  1318.0      7.184629
2020-03-19  1270.0      7.147559
2020-03-20  1116.0      7.018401
...            ...           ...
2020-06-03  1934.0      7.567862
2020-06-04  1873.0      7.535830
2020-06-05     NaN      7.610539
2020-06-06  1747.0      7.466227
2020-06-07  1782.0      7.486052
```

Configuration Optimization

Luminaire *HyperparameterOptimization* performs auto-selection of the best data preprocessing configuration and the outlier detection model training configuration with respect to the input time series. This option enables Luminaire to work as a hands-off system where the user only has to provide the input data along with its frequency. This option should be used if the user wants avoid any manual configuration and should be called prior to the data pre-processing and training steps.

```
>>> print(data)
               raw
index
2020-01-01  1326.0
2020-01-02  1552.0
2020-01-03  1432.0
2020-01-04  1470.0
2020-01-05  1565.0
...            ...
2020-06-03  1934.0
2020-06-04  1873.0
2020-06-05  1674.0
2020-06-06  1747.0
2020-06-07  1782.0
>>> hopt_obj = HyperparameterOptimization(freq='D')
>>> opt_config = hopt_obj.run(data=data)
>>> print(opt_config)
{'LuminaireModel': 'LADStructuralModel', 'data_shift_truncate': 0, 'fill_rate': 0.7423534446
```

Fully Automatic Outlier Detection

Since the optimized configuration contains all the parameters required for data pre-processing and training, this can be used downstream for performing the data pre-processing and training.

```
>>> de_obj = DataExploration(freq='D', **opt_config)
>>> training_data, pre_prc = de_obj.profile(data)
>>> print(training_data)
                raw  interpolated
2020-01-01  1326.0      7.190676
2020-01-02  1552.0      7.347943
2020-01-03  1432.0      7.267525
2020-01-04  1470.0      7.293697
2020-01-05  1565.0      7.356279
...            ...           ...
2020-06-03  1934.0      7.567862
2020-06-04  1873.0      7.535830
2020-06-05  1674.0      7.423568
2020-06-06  1747.0      7.466227
2020-06-07  1782.0      7.486052
```

The above piece of code makes the data ready to be ingested for training. The only step left before training is to extract the luminaire outlier detection model object for the optimized configuration.

```
>>> model_class_name = opt_config['LuminaireModel']
>>> module = __import__('luminaire.model', fromlist=[''])
>>> model_class = getattr(module, model_class_name)
>>> print(model_class)
<class 'luminaire_models.model.lad_structural.LADStructuralModel'>
```

Since, we have to optimal model class along with other optimal configurations, we can run training as follows:

```
>>> model_object = model_class(hyper_params=opt_config, freq='D')
>>> success, model_date, trained_model = model_object.train(data=training_data, **pre_prc)
>>> print(success, model_date, trained_model)
(True, '2020-06-07 00:00:00', <luminaire_models.model.lad_structural.LADStructuralModel obje
```

This trained model is now ready to be used for scoring future data points.

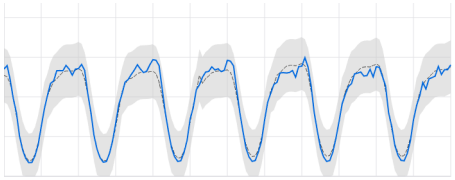
```
>>> trained_model.score(2000, '2020-06-08')
{'Success': True, 'IsLogTransformed': 1, 'LogTransformedAdjustedActual': 7.601402334583733,
```

Outlier Detection

Luminaire can detect outliers in time series data by modeling the predictive and the variational patterns of a time series trajectory. Luminaire is capable of tracking outliers for any time series data by applying two specific modeling capabilities:

- **Structural Model**: This technique is suitable for time series datasets that show periodic patterns and contains good predictive signals through temporal correlations.
- **Filtering Model**: This technique is suitable for noisy time series datasets that contains almost no predictive signals from the periodic or temporal correlation signals.

Anomaly Detection using Structural Model



Luminaire provides the full capability to have user-specified configuration for structural modeling. Under the hood, Luminaire implements a linear or an exponential model allowing multiple user specified auto regressive and moving average components to track any temporal correlational patterns. Fourier transformation can also be applied under the hood if the data shows strong seasonality or periodic patterns. As external structural information, Luminaire allows holidays to be added as external exogenous features (currently supported for daily data only) inside the structural model.

```
>>> hyper_params = {"include_holidays_exog": True, "is_log_transformed": False, "max_ft_freq
>>> lad_struct_obj = LADStructuralModel(hyper_params=hyper, freq='D')
>>> print(lad_struct_obj)
<luminaire_models.model.lad_structural.LADStructuralModel object at 0x7fc91882bb38>
```

Luminaire allows some data-specific information to be added during the training process of the structural model through *preprocessing_parameters*. The *preprocessing_parameters* can either be specified by the user if the data-specific information is available through external sources OR can be obtained using *Luminaire DataExploration*. The data-specific information includes a list of trend changes, change points (data shifts), and start and end of the input time series.

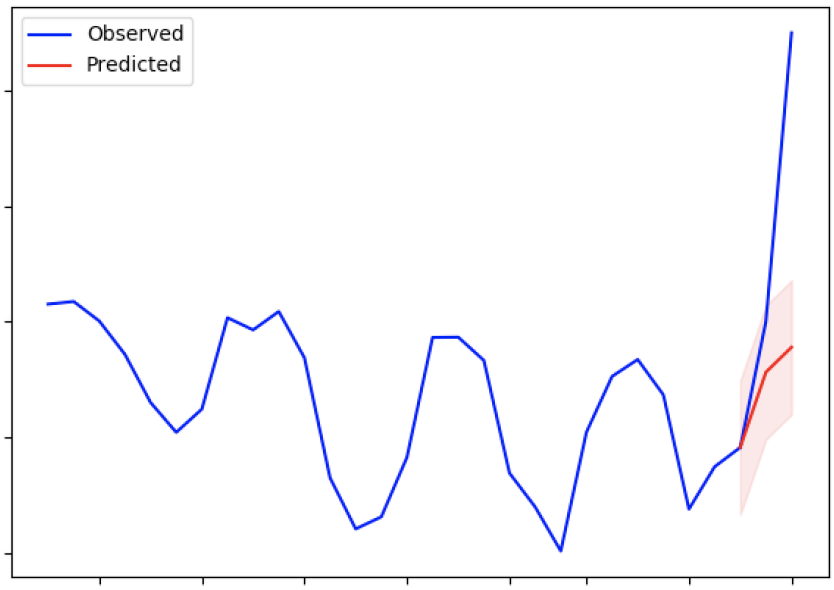
```
>>> de_obj = DataExploration(freq='D', data_shift_truncate=False, is_log_transformed=True, f
>>> data, pre_prc = de_obj.profile(data)
>>> print(pre_prc)
{'success': True, 'trend_change_list': ['2020-04-01 00:00:00'], 'change_point_list': ['2020-
'is_log_transformed': 1, 'min_ts_mean': None, 'ts_start': '2020-01-01 00:00:00', 'ts_end': '
```

These *preprocessing_parameters* are used for training the structural model.

```
>>> success, model_date, model = lad_struct_obj.train(data=data, **pre_prc)
>>> print(success, model_date, model)
(True, '2020-06-07 00:00:00', <luminaire_models.model.lad_structural.LADStructuralModel obje
```

The trained model works as a data-driven source of truth to evaluate any future time series values to be monitored. The *score* method is used to check whether new data points are anomalous.

```
>>> model.score(2000, '2020-06-08')
{'Success': True, 'IsLogTransformed': 1, 'LogTransformedAdjustedActual': 7.601402334583733,
>>> model.score(2500, '2020-06-09')
{'Success': True, 'IsLogTransformed': 1, 'LogTransformedAdjustedActual': 7.824445930877619,
```



The scoring function outputs several fields. The key to identifying whether a data point has been detected as an anomaly is the *AnomalyProbability* field (for anomalous fluctuations in either direction) and *DownAnomalyProbability*, *UpAnomalyProbability* for one-sided fluctuations that are lower or higher than expected, respectively. The user can set any anomaly threshold to identify whether a pont is an anomaly or not. From the above example, by setting the anomaly threshold at *0.99* for both sided fluctuations, we can see the the the value corresponding to *2020-06-08* is non anomalous whereas the value for *2020-06-09* is anomalous. Luminaire also has its own pre-specified thresholds at *0.9* and at *0.999* for identifying mild an extreme anomalies (see the keys *IsAnomaly* and *IsAnomalyExtreme*).

Luminaire generates a *ModelFreshness* score to identify how fresh the model is (i.e. what is the difference between the scoring data date and the model date). This freshness scores varies between 0 to 1 and the model object expires whenever the freshness score exceeds the value 1.

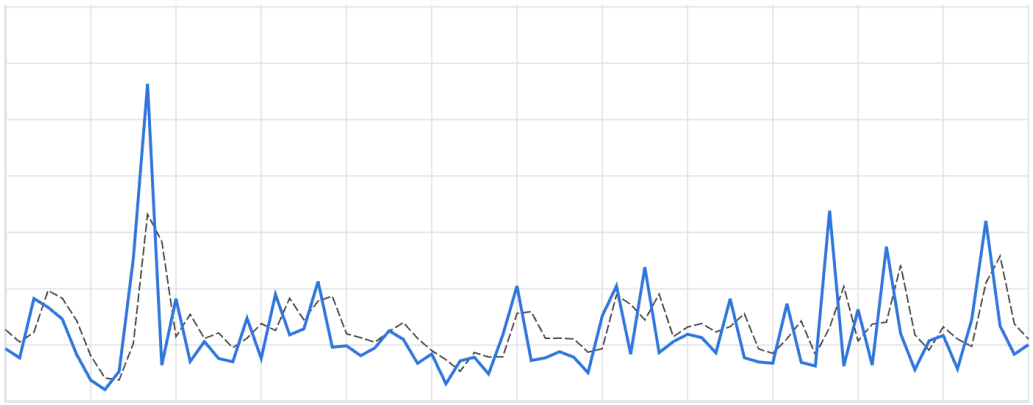
Forecasting

Since the anomaly detection process through structural modeling depends on quantifying the predictive and the variational patterns of the underlying data, Luminaire also outputs several forecasting metrics such as *Prediction*, *StdErr*, *CILower* and *CIUpper* that can be used for time series forecasting use cases.

Note

The *ConfLevel* in the scoring output corresponds to the generated confidence intervals and to the *IsAnomaly* flag

Anomaly Detection using Filtering Model



Luminaire allows monitoring noisy and not too well behaved time series data by tracking the residual process from a filtering model. This model should not be used for predictive purposes but can be used to measure variational patterns and irregular fluctuations.

Filtering requires very minimal specification in terms of configurations. The user needs to only configure whether to implement a linear or exponential model.

```
>>> hyper = {"is_log_transformed": 1}
>>> lad_filter_obj = LADFilteringModel(hyper_params=hyper, freq='D')
>>> print(lad_filter_obj)
<luminaire_models.model.lad_filtering.LADFilteringModel object at 0x7fd2b1832dd8>
```

Similar to the structural model, the user can specify the *preprocessing_parameters* (see lad structural modeling tutorial for further information). These *preprocessing_parameters* are required to train the Luminaire filtering model.

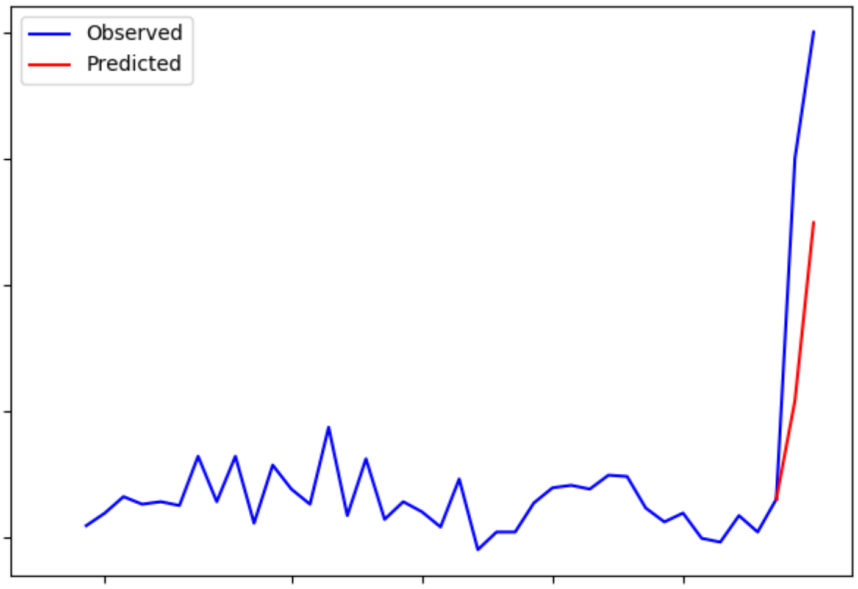
```
>>> success, model_date, model = lad_filter_obj.train(data=data, **pre_prc)
>>> print(success, model_date, model)
(True, '2019-08-27 00:00:00', <luminaire_models.model.lad_filtering.LADFilteringModel object
```

Similar to the structural model, this trained filtering model can be used to score any future time series values. Moreover, the filtering model updates some components of the model object every time it scores to keep the variational information updated.

```
>>> scores, model_update = model.score(400, '2019-08-28')
>>> print(scores, model_update)
({'Success': True, 'AdjustedActual': 1.4535283491638031, 'ConfLevel': 90.0, 'Prediction': 20
```

The trained *model* can only be used to score the next innovation after the training. To score any futher points in the future, the iterative *model_update* needs to be used.

```
>>> scores_2, model_update_2 = model_update.score(500, '2019-08-29')
>>> print(scores_2, model_update_2)
({'Success': True, 'AdjustedActual': -0.591849553174421, 'ConfLevel': 90.0, 'Prediction': 34
```



Note

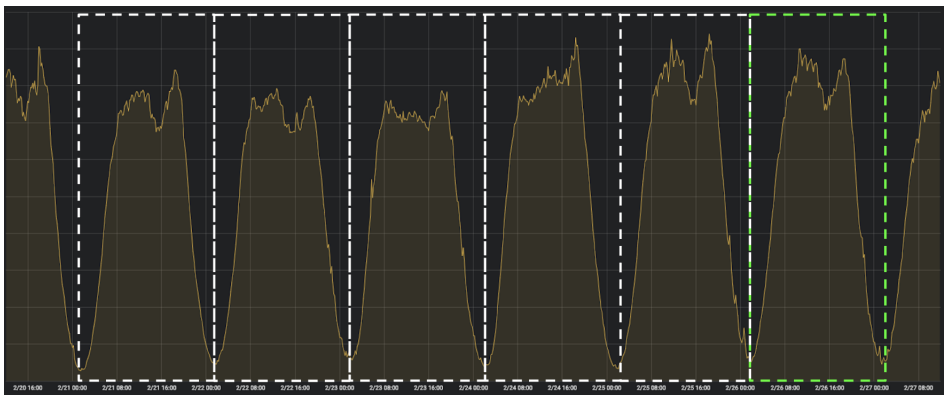
Prediction for the filtering model is a posterior prediction, which means the prediction is made after observing the data to score. See kalman_filter for more information.

Note

It is important to note that the model update process during scoring only updates a small portion of the model component. It is a good practice to train the model over some schedule to acheive the best performance.

Anomaly Detection for Streaming data

Luminaire *WindowDensityModel* implements the idea of monitoring data over comparable windows instead of tracking individual data points as outliers. This is a useful approach for tracking anomalies over high frequency data, which tends to show a higher level of noise. Hence, tracking anomalies over streaming data essesntially means tracking sustained fluctuations.



Although *WindowDensityModel* is designed to track anomalies over streaming data, it can be used to track any sustained fluctuations over a window for any frequency. This detection type is suggested for up to hourly data frequency.

Anomaly Detection: Pre-Configured Settings

Luminaire provides the capability to configure model parameters based on the frequency that the data has been observed and the methods that can be applied (please refer to the Window density Model user guide for detailed configuration options). Luminaire settings for the window density model are already pre-configured for some typical pandas frequency types and settings for any other frequency types should be configured manually (see the user guide for more information).

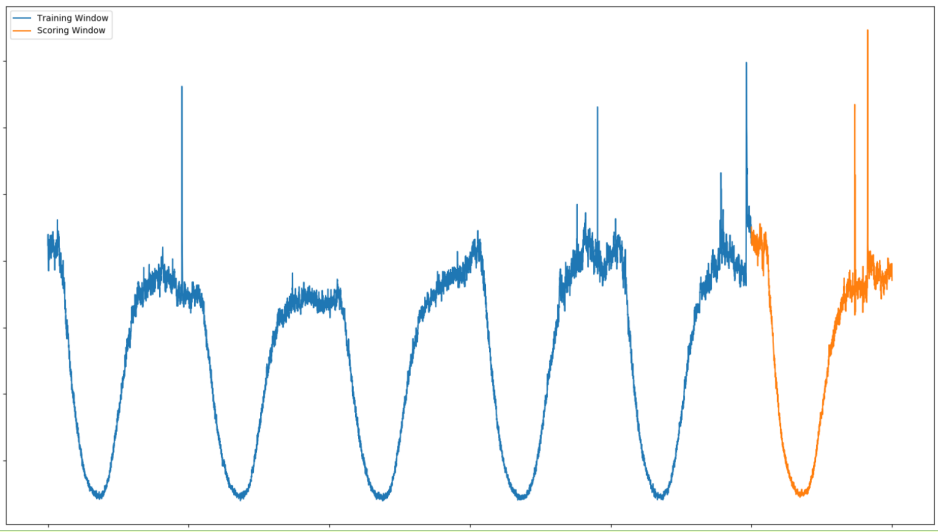
```
>>> print(data)
                        raw  interpolated
index
2020-05-25 00:00:00  10585.0       10585.0
2020-05-25 00:01:00  10996.0       10996.0
2020-05-25 00:02:00  10466.0       10466.0
2020-05-25 00:03:00  10064.0       10064.0
2020-05-25 00:04:00  10221.0       10221.0
...                      ...           ...
2020-06-16 23:55:00  11356.0       11356.0
2020-06-16 23:56:00  10852.0       10852.0
2020-06-16 23:57:00  11114.0       11114.0
2020-06-16 23:58:00  10663.0       10663.0
2020-06-16 23:59:00  11034.0       11034.0
>>> hyper_params = WindowDensityHyperParams(freq='M').params
>>> wdm_obj = WindowDensityModel(hyper_params=hyper_params)
>>> success, model = wdm_obj.train(data=data)
>>> print(success, model)
(True, <luminaire_models.model.window_density.WindowDensityModel object at 0x7f8cda42dcc0>)
```

The model object contains the data density structure over a pre-specified window, given the frequency. Lumianire sets the following defaults for some typical pandas frequencies (any custom requirements can be updated in the hyperparameter object instance):

- ‘S’: Hourly windows
- ‘M’: Daily windows
- ‘QM’: Weekly windows
- ‘H’: 12 hours windows
- ‘D’: 10 days windows

- ‘custom’: User specified windows

In order to score a new window innovation given the trained model object, we have to provide a equal sized window that represents a similar time interval. For example, if each of the windows in the training data represents a 24 hour window between 9 AM to 8:59:59 AM for last few days, the scoring data should represent the same interval of a different day and should have the same window size.




```
>>> scoring_data
                        raw interpolated
index
2020-06-17 00:00:00  1121.0       1121.0
2020-06-17 00:01:00  1091.0       1091.0
2020-06-17 00:02:00  1063.0       1063.0
2020-06-17 00:03:00  1085.0       1085.0
2020-06-17 00:04:00  1063.0       1063.0
...                     ...          ...
2020-06-17 23:55:00   968.0        968.0
2020-06-17 23:56:00   995.0        995.0
2020-06-17 23:57:00   963.0        963.0
2020-06-17 23:58:00   968.0        968.0
2020-06-17 23:59:00   920.0        920.0
>>> scores = model.score(scoring_data)
>>> print(scores)
{'Success': True, 'ConfLevel': 99.9, 'IsAnomaly': False, 'AnomalyProbability': 0.69567457348
```

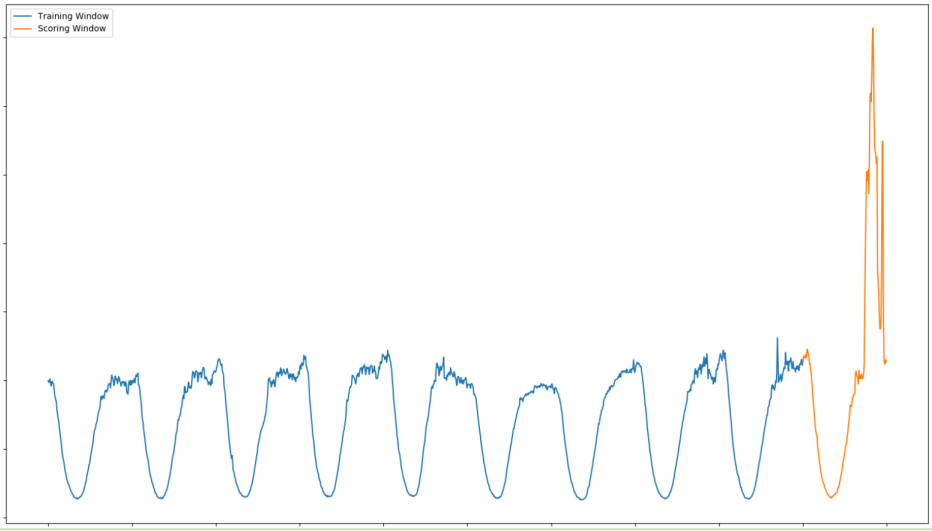
Anomaly Detection: Manual Configuration

There are several options in the *WindowDensityHyperParams* class that can be manually configured. The configuration should be selected mostly based on the frequency that the data has been observed.

```
>>> print(data)
                            raw interpolated
index
2020-05-20 00:03:00  6393.451190  6393.451190
2020-05-20 00:13:00  6491.426190  6491.426190
2020-05-20 00:23:00  6770.469444  6770.469444
2020-05-20 00:33:00  6490.798810  6490.798810
2020-05-20 00:43:00  6273.786508  6273.786508
...                          ...          ...
2020-06-09 23:13:00  5619.341270  5619.341270
2020-06-09 23:23:00  5573.001190  5573.001190
2020-06-09 23:33:00  5745.400000  5745.400000
2020-06-09 23:43:00  5761.355556  5761.355556
2020-06-09 23:53:00  5558.577778  5558.577778
>>>hyper_params = WindowDensityHyperParams(freq='custom',
```

```
                                            detection_method='kldiv',
                                            baseline_type="last_window",
                                            min_window_length=6*12,
                                            max_window_length=6*24*84,
                                            window_length=6*24,
                                            ma_window_length=24,
                                            ).params
>>> wdm_obj = WindowDensityModel(hyper_params=hyper_params)
>>> success, model = wdm_obj.train(data=data)
>>> print(success, model)
(True, <luminaire_models.model.window_density.WindowDensityModel object at 0x7f8d5f1a6940>)
```

The trained model object can be used to score data representing the same interval from a different day and having the same window size.




```
>>> scoring_data
                            raw interpolated
index
2020-06-10 00:00:00  5532.556746  5532.556746
2020-06-10 00:10:00  5640.711905  5640.711905
2020-06-10 00:20:00  5880.368254  5880.368254
2020-06-10 00:30:00  5842.397222  5842.397222
2020-06-10 00:40:00  5827.231746  5827.231746
...                          ...          ...
2020-06-10 23:10:00  7210.905952  7210.905952
2020-06-10 23:20:00  5739.459524  5739.459524
2020-06-10 23:30:00  5590.413889  5590.413889
2020-06-10 23:40:00  5608.291270  5608.291270
2020-06-10 23:50:00  5753.794444  5753.794444
>>> scores = model.score(scoring_data)
>>> print(scores)
{'Success': True, 'ConfLevel': 99.9, 'IsAnomaly': True, 'AnomalyProbability': 0.999999985183
```

User Guide

Luminaire Outlier Detection Models: Structural Modeling

exception `luminaire.model.lad_structural.`**`LADStructuralError`** (message)
Exception class for Luminaire structural anomaly detection model.

class `luminaire.model.lad_structural.`**`LADStructuralHyperParams`** (include_holidays_exog=True, p=2, q=2, is_log_transformed=True, max_ft_freq=3)

Exception class for Luminaire structural anomaly detection model.

Parameters:

- **include_holidays_exog** (*bool, optional*) – whether to include holidays as exogenous variables in the regression. Holidays are defined in **`LADHolidays`**
- **p** (*int, optional*) – Order for the AR component of the model.
- **q** (*int, optional*) – Order for the MA component of the model.
- **is_log_transformed** (*bool, optional*) – A flag to specify whether to take a log transform of the input data. If the data contain negatives, is_log_transformed is ignored even though it is set to True.
- **max_ft_freq** (*int, optional*) – The maximum frequency order for the Fourier transformation.

class `luminaire.model.lad_structural.`**`LADStructuralModel`**(hyper_params: {'include_holidays_exog': True, 'p': 2, 'q': 2, 'is_log_transformed': True, 'max_ft_freq': 3}, freq, min_ts_length=None, max_ts_length=None, min_ts_mean=None, min_ts_mean_window=None, **kwargs)

A LAD structural time series model.

Parameters:

- **hyper_params** (*dict*) – Hyper parameters for Luminaire structural modeling. See `luminaire.optimization.hyperparameter_optimization.HyperparameterOptimization`_ for detailed information.
- **freq** (*str*) – The frequency of the time-series. A Pandas offset such as 'D', 'H', or 'M'.
- **min_ts_length** (*int, optional*) – The minimum required length of the time series for training.
- **max_ts_length** (*int, optional*) – The maximum required length of the time series for training.
- **min_ts_mean** (*float, optional*) – Minimum average values in the most recent window of the time series. This optional parameter can be used to avoid over-alerting from noisy low volume time series.
- **min_ts_mean_window** (*int, optional*) – Size of the most recent window to calculate min_ts_mean.

Note

This class should be used to manually configure the structural model. Exact configuration parameters can be found in *luminaire.hyperparameter_optimization.HyperparameterOptimization*. Optimal configuration can be obtained by using LAD hyperparameter optimization.

```
>>> hyper = {"include_holidays_exog": 0, "is_log_transformed": 1, "max_ft_freq": 2, "p": 5
lad_struct_model = LADStructuralModel(hyper_params=hyper, freq='D')
>>> lad_struct_model
<luminaire.model.lad_structural.LADStructuralModel object at 0x103efe320>
```

`score` **(**observed_value**,** pred_date**,** **kwargs**)**

This function scores a value observed at a data date given a trained LAD structural model object.

Parameters:

- **observed_value** (*float*) – Observed time series value on the prediction date.
- **pred_date** (*str*) – Prediction date. Needs to be in yyyy-mm-dd or yyyy-mm-dd hh:mm:ss format.

Returns: Anomaly flag, anomaly probability, prediction and other related metrics.

Return type: dict

```
>>> model
<luminaire.model.lad_structural.LADStructuralModel object at 0x11c1c3550>
```

```
>>> model._params['training_end_date'] # Last data date for training time series
'2020-06-07 00:00:00'
```

```
>>> model.score(2000 ,'2020-06-08')
{'Success': True, 'IsLogTransformed': 0, 'AdjustedActual': 2000, 'Prediction': 1943.20426
'StdErr': 93.084646777553, 'CILower': 1785.519523590432, 'CIUpper': 2100.88899967807, 'Co
'ExogenousHolidays': 0, 'IsAnomaly': False, 'IsAnomalyExtreme': False, 'AnomalyProbabilit
'DownAnomalyProbability': 0.286642755841402, 'UpAnomalyProbability': 0.713357244158598, '
>>> model.score(2500 ,'2020-06-09')
{'Success': True, 'IsLogTransformed': 0, 'AdjustedActual': 2500, 'Prediction': 2028.98993
'StdErr': 93.6623172459385, 'CILower': 1861.009403637476, 'CIUpper': 2186.97046407242, 'C
'ExogenousHolidays': 0, 'IsAnomaly': True, 'IsAnomalyExtreme': True, 'AnomalyProbability'
'DownAnomalyProbability': 6.489152464261849e-07, 'UpAnomalyProbability': 0.9999993510847
'ModelFreshness': 0.2}
```

`train` **(**data**,** optimize=False**,** **kwargs**)**

This function trains a structural LAD model for a given time series.

Parameters:

- **data** (*pandas.DataFrame*) – Input time series data
- **optimize** (*bool, optional*) – Flag to identify whether called from hyperparameter optimization

Returns: success flag, the model date and the trained lad structural model object

Return type: tuple[bool, str, LADStructuralModel object]

```
>>> data
               raw interpolated
2020-01-01  1326.0       1326.0
2020-01-02  1552.0       1552.0
2020-01-03  1432.0       1432.0
2020-01-04  1470.0       1470.0
2020-01-05  1565.0       1565.0
...            ...          ...
2020-06-03  1934.0       1934.0
2020-06-04  1873.0       1873.0
2020-06-05  1674.0       1674.0
2020-06-06  1747.0       1747.0
2020-06-07  1782.0       1782.0
>>> hyper = {"include_holidays_exog": 0, "is_log_transformed": 0, "max_ft_freq": 2, "p":
>>> de_obj = DataExploration(freq='D', is_log_transformed=0)
>>> data, pre_prc = de_obj.profile(data)
>>> pre_prc
{'success': True, 'trend_change_list': ['2020-04-01 00:00:00'], 'change_point_list': ['20
'is_log_transformed': 0, 'min_ts_mean': None, 'ts_start': '2020-01-01 00:00:00',
'ts_end': '2020-06-07 00:00:00'}
>>> lad_struct_obj = LADStructuralModel(hyper_params=hyper, freq='D')
>>> model = lad_struct_obj.train(data=data, **pre_prc)
```

```
>>> model
(True, '2020-06-07 00:00:00', <luminaire.model.lad_structural.LADStructuralModel object a
```

Luminaire Outlier Detection Models: Factoring holidays as exogenous

class `luminaire.model.model_utils.`**`LADHolidays`** (name=None, holiday_rules=None)

A class that generates holiday calendars to be used as external features in the batch outlier detection model. By default, holidays include:

- Memorial Day, plus the weekend leading into it

- Veterans Day, plus the weekend leading into it
- Labor Day
- President's Day
- Martin Luther King Jr. Day
- Valentine's Day
- Mother's Day
- Father's Day
- Independence Day (actual and observed)
- Halloween
- Superbowl
- Easter
- Thanksgiving, plus the following weekend
- Christmas Eve, Christmas Day, and all dates up to New Year's Day (actual and observed)

Luminaire Outlier Detection Models: Kalman Filter

class `luminaire.model.lad_filtering.`**`LADFilteringHyperParams`** (is_log_transformed=True)

Exception class for Luminaire filtering anomaly detection model.

Parameters: **is_log_transformed** (*bool, optional*) – A flag to specify whether to take a log transform of the input data. If the data contain negatives, is_log_transformed is ignored even though it is set to True.

class `luminaire.model.lad_filtering.`**`LADFilteringModel`** (hyper_params: {'is_log_transformed': True}, freq, min_ts_length=None, max_ts_length=None, **kwargs)

A Markovian state space model. This model detects anomaly based on the residual process obtained through Kalman Filter based model estimation.

Parameters:

- **hyper_params** (*dict*) – Hyper parameters for Luminaire structural modeling. See `luminaire.optimization.hyperparameter_optimization.HyperparameterOptimization`_ for detailed information.
- **freq** (*str*) – The frequency of the time-series. A Pandas offset such as 'D', 'H', or 'M'.
- **min_ts_length** (*int, optional*) – The minimum required length of the time series for training.
- **max_ts_length** (*int, optional*) – The maximum required length of the time series for training.

```
>>> hyper = {"is_log_transformed": 1}
lad_filtering_model = LADFilteringModel(hyper_params=hyper, freq='D')
```

```
>>> lad_filtering_model
<luminaire.model.filtering.LADFilteringModel object at 0x103efe320>
```

`score` **(**observed_value**,** pred_date**,** synthetic_actual=None**,** **kwargs**)**

This function scores a value observed at a data date given a trained LAD filtering model object.

Parameters:

- **observed_value** (*float*) – Observed time series value on the prediction date.
- **pred_date** (*str*) – Prediction date. Needs to be in yyyy-mm-dd or yyyy-mm-dd hh:mm:ss format.
- **synthetic_actual** (*float, optional*) – Synthetic time series value. This is an artificial value used to optimize classification accuracy in Luminaire hyperparameter optimization.

Returns: Model results and LAD filtering model object

Return type: tuple[dict, LADFilteringlModel object]

```
>>> model
<luminaire.model.lad_filtering.LADFilteringModel object at 0x11f0b2b38>
>>> model._params['training_end_date']
'2020-06-07 00:00:00'
```

```
>>> model.score(2000 ,'2020-06-08')
({'Success': True, 'AdjustedActual': 0.10110881711268949, 'ConfLevel': 90.0, 'Prediction'
'PredStdErr': 212.4399633739204, 'IsAnomaly': False, 'IsAnomalyExtreme': False,
'AnomalyProbability': 0.4244056403219776, 'DownAnomalyProbability': 0.2877971798390112,
'UpAnomalyProbability': 0.7122028201609888, 'NonStationarityDiffOrder': 2, 'ModelFreshnes
<luminaire.model.lad_filtering.LADFilteringModel object at 0x11f3c0860>)
```

train (data**,** **kwargs**)**

This function trains a filtering LAD model for a given time series.

Parameters: **data** (*pandas.DataFrame*) – Input time series data

Returns: The success flag, model date and a trained lad filtering object

Return type: tuple[bool, str, LADFilteringModel object]

```
>>> data
               raw interpolated
2020-01-01  1326.0       1326.0
2020-01-02  1552.0       1552.0
2020-01-03  1432.0       1432.0
2020-01-04  1470.0       1470.0
2020-01-05  1565.0       1565.0
...            ...          ...
2020-06-03  1934.0       1934.0
2020-06-04  1873.0       1873.0
2020-06-05  1674.0       1674.0
2020-06-06  1747.0       1747.0
2020-06-07  1782.0       1782.0
>>> hyper = {"is_log_transformed": 1}
>>> de_obj = DataExploration(freq='D', is_log_transformed=1, fill_rate=0.95)
>>> data, pre_prc = de_obj.profile(data)
>>> pre_prc
{'success': True, 'trend_change_list': ['2020-04-01 00:00:00'], 'change_point_list': ['20
'is_log_transformed': 1, 'min_ts_mean': None, 'ts_start': '2020-01-01 00:00:00',
'ts_end': '2020-06-07 00:00:00'}
>>> lad_filter_obj = LADFilteringModel(hyper_params=hyper, freq='D')
>>> model = lad_filter_obj.train(data=data, **pre_prc)
```

```
>>> model
(True, '2020-06-07 00:00:00', <luminaire.model.lad_filtering.LADFilteringModel object at
```

exception luminaire.model.lad_filtering.**LADFilteringModelError** (message)

Exception class for Luminaire filtering anomaly detection model.

Data Exploration and Profiling

class `luminaire.exploration.data_exploration.`**`DataExploration`** (freq='D', min_ts_mean=None, fill_rate=None, max_window_size=24, window_size=None, sig_level=0.05, min_ts_length=None, max_ts_length=None, is_log_transformed=None, data_shift_truncate=True, min_changepoint_padding_length=None, change_point_threshold=2, *args, **kwargs)

This is a general class for time series data exploration and pre-processing.

Parameters:

- **freq** (*str*) – The frequency of the time-series. A Pandas offset such as 'D', 'H', or 'M'.
- **sig_level** (*float*) – The significance level to use for any statistical test withing data profile. This should be a number between 0 and 1.
- **min_ts_mean** (*float, optional*) – The minimum mean value of the time series required for the model to run. For data that originated as integers (such as counts), the ARIMA model can behave erratically when the numbers are small. When this parameter is set, any time series whose mean value is less than this will automatically result in a model failure, rather than a mostly bogus anomaly.
- **fill_rate** (*float, optional*) – Minimum proportion of data availability in the recent data window.
- **max_window_size** (*int, optional*) – The maximum size of the sub windows for input data segmentation.
- **window_size** (*int, optional*) – The size of the sub windows for input data segmentation.
- **min_ts_length** (*int, optional*) – The minimum required length of the time series for training.
- **max_ts_length** (*int, optional*) – The maximum required length of the time series for training.
- **is_log_transformed** (*bool, optional*) – A flag to specify whether to take a log transform of the input data. If the data contain negatives, is_log_transformed is ignored even though it is set to True.
- **data_shift_truncate** (*bool, optional*) – A flag to specify whether left side of the most recent change point needs to be truncated from the training data.
- **min_changepoint_padding_length** (*bool, optional*) – A padding length between two change points. This parameter makes sure that two consecutive change points are not close to each other.
- **change_point_threshold** (*float, optional*) – Minimum threshold (a value > 0) to flag change points based on KL divergence.

`kf_naive_outlier_detection` **(**input_series**,** idx_position**)**

This function detects outlier for the specified index position of the series.

Parameters:

- **input_series** (*numpy.array*) – Input time series
- **idx_position** (*int*) – Target index position

Returns: Anomaly flag

Return type: bool

```
>>> input_series = [110, 119, 316, 248, 451, 324, 241, 275, 381]
>>> self.kf_naive_outlier_detection(input_series, 6)
False
```

`profile` **(**df**,** impute_only=False**,** **kwargs**)**

This function performs required data profiling and pre-processing before hyperparameter optimization or time series model training.

Parameters:
- **df** (*list/pandas.DataFrame*) – Input time series.
- **impute_only** (*bool, optional*) – Flag to perform preprocessing until imputation OR full preprocessing.

Returns: Preprocessed dataframe with batch data summary.

Return type: tuple[pandas.dataFrame, dict]

```
>>> de_obj = DataExploration(freq='D', data_shift_truncate=1, is_log_transformed=0, fill_
>>> data
               raw
index
2020-01-01  1326.0
2020-01-02  1552.0
2020-01-03  1432.0
2020-01-04  1470.0
2020-01-05  1565.0
...            ...
2020-06-03  1934.0
2020-06-04  1873.0
2020-06-05  1674.0
2020-06-06  1747.0
2020-06-07  1782.0
>>> data, summary = de_obj.profile(data)
>>> data, summary
(              raw interpolated
2020-03-16  1371.0       1371.0
2020-03-17  1325.0       1325.0
2020-03-18  1318.0       1318.0
2020-03-19  1270.0       1270.0
2020-03-20  1116.0       1116.0
...            ...          ...
2020-06-03  1934.0       1934.0
2020-06-04  1873.0       1873.0
2020-06-05  1674.0       1674.0
2020-06-06  1747.0       1747.0
2020-06-07  1782.0       1782.0
[84 rows x 2 columns], {'success': True, 'trend_change_list': ['2020-04-01 00:00:00'], 'c
['2020-03-16 00:00:00'], 'is_log_transformed': 0, 'min_ts_mean': None, 'ts_start': '2020-
'ts_end': '2020-06-07 00:00:00'})
```

exception `luminaire.exploration.data_exploration.`**`DataExplorationError`** (message)
Exception class for Luminaire Data Exploration.

Luminaire Configuration Optimization

class `luminaire.optimization.hyperparameter_optimization.`**`HyperparameterOptimization`** (freq, detection_type='OutlierDetection', min_ts_mean=None, max_ts_length=None, min_ts_length=None, scoring_length=None, **kwargs)
Hyperparameter optimization for LAD outlier detection configuration for batch data.

Parameters:

- **freq** (*str*) – The frequency of the time-series. A Pandas offset such as ‘D’, ‘H’, or ‘M’.
- **detection_type** (*str, optional*) – Luminaire anomaly detection type. Only Outlier detection for batch data is currently supported.
- **min_ts_mean** (*float, optional*) – Minimum average values in the most recent window of the time series. This optional parameter can be used to avoid over-alerting from noisy low volume time series.
- **max_ts_length** (*int, optional*) – The maximum required length of the time series for training.
- **min_ts_length** (*int, optional*) – The minimum required length of the time series for training.
- **scoring_length** (*int, optional*) – Number of innovations to be scored after training window with respect to the frequency.

run (data**,** max_evals=50**)**

This function runs hyperparameter optimization fort LAD batch outlier detection models

Parameters:

- **data** (*list[list]*) – Input time series.
- **max_evals** (*int, optional*) – Number of iterations for hyperparameter optimization.

Returns: Optimal hyperparameters.

Return type: dict

```
>>> data
[[Timestamp('2020-01-01 00:00:00'), 1326.0],
[Timestamp('2020-01-02 00:00:00'), 1552.0],
[Timestamp('2020-01-03 00:00:00'), 1432.0],
. . . ,
[Timestamp('2020-06-06 00:00:00'), 1747.0],
[Timestamp('2020-06-07 00:00:00'), 1782.0]]
>>> hopt_obj = HyperparameterOptimization(freq='D', detection_type='OutlierDetection')
>>> hyper_params = hopt_obj._run(data=data, max_evals=5)
```

```
>>> hyper_params
{'LuminaireModel': 'LADStructuralModel', 'data_shift_truncate': 0, 'fill_rate': 0.840924
'include_holidays_exog': 1, 'is_log_transformed': 1, 'max_ft_freq': 3, 'p': 4, 'q': 3}
```

Luminaire Streaming Anomaly Detection Models: Window Density Model

class luminaire.model.window_density.**WindowDensityHyperParams** (freq='M', ignore_window=None, max_missing_train_prop=0.1, is_log_transformed=False, baseline_type='aggregated', detection_method=None, min_window_length=None, max_window_length=None, window_length=None, ma_window_length=None, detrend_method='ma')

Hyperparameter class for Luminaire Window density model.

Parameters:

- **freq** (*str*) – The frequency of the time-series. Luminaire supports default configuration for ‘S’, ‘M’, ‘QM’, ‘H’, ‘D’. Any other frequency type should be specified as ‘custom’ and configuration should be set manually.
- **ignore_window** (*int, optional*) – ignore a time window to be considered for training.
- **max_missing_train_prop** (*float, optional*) – Maximum proportion of missing observation allowed in the training data.
- **is_log_transformed** (*bool, optional*) – A flag to specify whether to take a log transform of the input data. If the data contain negatives, is_log_transformed is ignored even though it is set to True.
- **baseline_type** (*str, optional*) – A string flag to specify whether to take set a baseline as the previous sub-window from the training data for scoring or to aggregate the overall window as a baseline. Possible values: ”last_window” ”aggregated”
- **detection_method** (*str, optional*) – A string that select between two window testing method. Possible values: ”kldiv” (KL-divergence) ”sign_test” (Wilcoxon sign rank test).
- **min_window_length** (*int, optional*) – Minimum size of the scoring window / a stable training sub-window length.

Note

This is not the minimum size of the whole training window which is the combination of stable sub-windows.

Parameters: **max_window_length** (*int, optional*) – Maximum size of the scoring window / a stable training sub-window length.

Note

This is not the maximum size of the whole training window which is the combination of stable sub-windows.

Parameters: **window_length** (*int, optional*) – Size of the scoring window / a stable training sub-window length.

Note

This is not the size of the whole training window which is the combination of stable sub-windows.

Parameters: **ma_window_length** (*int, optional*) – Size of the window for detrending scoring window / stable training sub-windows through moving average method.

Note

ma_window_length should be small enough to maintain the stable structure of the training / scoring window and large enough to remove the trend. The ideal size can be somewhere between (0.1 * window_length) and (0.25 * window length).

Parameters: **detrend_method** (*str, optional*) – A string that select between two stationarizing method. Possible values: - “ma” (moving average based) - “diff” (differencing based).

class `luminaire.model.window_density.`**`WindowDensityModel`** (hyper_params: {‘freq’: ‘M’, ‘ignore_window’: None, ‘max_missing_train_prop’: 0.1, ‘is_log_transformed’: False, ‘baseline_type’: ‘aggregated’, ‘detection_method’: ‘kldiv’, ‘min_window_length’: 720, ‘max_window_length’: 120960, ‘window_length’: 1440, ‘ma_window_length’: 60, ‘detrend_method’: ‘ma’}, **kwargs)

This model detects anomalous windows using KL divergence (for high frequency data) and Wilcoxon sign rank test (for low frequency data).

Parameters: **hyper_params** (*dict*) – Hyper parameters for Luminaire window density model. See `luminaire.model.window_density.WindowDensityHyperParams`_ for detailed information.

Returns: Anomaly probability for the execution window and other related model outputs

Return type: list[dict]

score (data, **kwargs)

Function scores input series for anomalies

Parameters: **data** (*pandas.DataFrame*) – Input time series to score

Returns: Output dictionary with scoring summary.

Return type: dict

```
>>> data
                        raw interpolated
index
2018-10-06 00:00:00  204800       204800
2018-10-06 01:00:00  222218       222218
2018-10-06 02:00:00  218903       218903
2018-10-06 03:00:00  190639       190639
2018-10-06 04:00:00  148214       148214
2018-10-06 05:00:00  106358       106358
2018-10-06 06:00:00   70081        70081
2018-10-06 07:00:00   47748        47748
2018-10-06 08:00:00   36837        36837
2018-10-06 09:00:00   33023        33023
2018-10-06 10:00:00   44432        44432
2018-10-06 11:00:00   72773        72773
2018-10-06 12:00:00  115180       115180
2018-10-06 13:00:00  157568       157568
2018-10-06 14:00:00  180174       180174
2018-10-06 15:00:00  190048       190048
2018-10-06 16:00:00  188391       188391
2018-10-06 17:00:00  189233       189233
2018-10-06 18:00:00  191703       191703
2018-10-06 19:00:00  189848       189848
2018-10-06 20:00:00  192685       192685
2018-10-06 21:00:00  196743       196743
2018-10-06 22:00:00  193016       193016
2018-10-06 23:00:00  196441       196441
>>> model
<luminaire.model.window_density.WindowDensityModel object at 0x7fcaab72fdd8>
```

```
>>> model.score(data)
{'Success': True, 'ConfLevel': 99.9, 'IsAnomaly': False, 'AnomalyProbability': 0.6963188
```

train (data, **kwargs)

Input time series for training.

Parameters: **data** – Input time series.

Returns: Training summary with a success flag.

Return type: tuple(bool, python model object)

```
>>> data
                        raw interpolated
index
2017-10-02 00:00:00  118870       118870
2017-10-02 01:00:00  121914       121914
```

```
2017-10-02 02:00:00  116097       116097
2017-10-02 03:00:00   94511        94511
2017-10-02 04:00:00   68330        68330
...                     ...          ...
2018-10-10 19:00:00  219908       219908
2018-10-10 20:00:00  219149       219149
2018-10-10 21:00:00  207232       207232
2018-10-10 22:00:00  198741       198741
2018-10-10 23:00:00  213751       213751
>>> hyper_params = WindowDensityHyperParams(freq='H').params
>>> wdm_obj = WindowDensityModel(hyper_params=hyper_params)
>>> success, model = wdm_obj.train(data)
```

```
>>> success, model
(True, <luminaire.model.window_density.WindowDensityModel object at 0x7fd7c5a34e80>)
```

Indices and tables

- **genindex**
- **modindex**
- **search**

Index

Python Module Index

l