Candela Documentation

Contents

Module `candela`

The Cobra ANomaly DEtection LibrAry for anomaly detection in timeseries.

The Cobra ANomaly DEtection LibrAry (candela) is a toolbox for the detection and analysis of anomalies and outliers in a wide range of data types. The toolbox has a modular structure that allows the extension of its functionalities, making it easily adaptable to the needs of various application areas.

It contains several different instruments for the analysis of different types of anomalies in the data, from easy statistical approaches to more advanced machine-learning techniques. The outlier detection techniques based on machine learning in the current version of the software are based on the Scikit-learn library (Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011).

The three submodules of candela are:

- functions: containing the actual functions for anomaly detections.
- plotting (has to be imported separately): containing advanced functions for visualization and reporting of the anomalies.
- particles2D (has to be imported separately): containing functions specialised in working with traces of particles moving on a plane.

Sub-modules

Module `candela.functions`

This submodule provides a set of function for anomaly dectection.

The content of this submodule is divided to two parts: a group of functions for anomaly detection, a class for managing anomaly detection based on a several features at once through an higher level processing.

The functions are of varying complexity moving from the simple_threshold() working only on unidimensional data to higher complexity tools as the one_class_svm(). In particular simple_threshold(), double_threshold() and tukeys_method() work only on 1D series while isolation_forest(), one_class_svm(), robust_covariance() and local_outlier_factor() work also on nD data.

The scatter_cluster class is specialised in the use with data from particles moving in 2D as produced by particles2d.preprocess. It is however easy to apply to other kind of data providing a list of relevant features. The class offers methods for segmenting the data extracting useful statistics with a sliding window.

Given generic time series or a trace (possibly preprocessed, see particles2d.preprocess) identifying anomalies can be as easy as:

```
import pandas as pd
from candela.functions import local_outlier_factor
from candela.particles2d import preprocess

df = pd.read_csv("my_trace.csv")
trace = preprocess(df, track_id=0)

outliers_velocity_LOF = local_outlier_factor(trace.vtot)
```

When using a scatter_cluster more steps are involved:

```
from candela.functions import scatter_cluster

segmented = scatter_cluster.segment(trace)
sc_model = scatter_cluster("IsolationForest")
sc_model.fit(segmented, feature_list="all")
sc_model.predict(segmented)
```

The currently available kernels for the scatter_cluster are: IsolationForest, OneClassSVM, RobustCovariance. It is possible to add new kernels (provided they expose the fit and predict methods) by updating the scatter_cluster.kernels attribute.

For more advanced segmentation options, see the documentation for scatter_cluster.segment_trace().

Functions

Function `__add_tails`

```
def __add_tails(
    data: Sequence[float],
    threshold: float,
    tailSize: int,
    piece: Sequence[int]
) -> Tuple[Sequence[int], float]
```

Adds tails to the groups identified in the process of double thresholding.

Adds up to tailSize points, given that they are not smaller than a threshold. Usually the tail is long half maxSpacing and the threshold is half the first threshold.

Args

`data` **: Sequence[float]** Sequence under processing
`threshold` **: float** This should be more permissive that the original threshold.
`tailSize` **: int** Maximum number of points to add on both ends of piece.
`piece` **: Sequence[int]** Group of points identified as outliers.

Returns

Tuple[Sequence[int], float] Indexes of the candidate anomalies in the group, total contributions of all points in the group of candidate anomalies.

Function `__simple_threshold`

```
def __simple_threshold(
    data: Sequence,
    absolute: bool = False,
    smaller: bool = False,
    exclude_nan: bool = True,
    fraction: float = 0.01,
    number: int = None,
    threshold: float = None
) -> Tuple[Sequence, float, Sequence]
```

See the documentation for simple_threshold().

Returns

Tuple[Sequence, float, Sequence] An array containing the indexes of the anomalies, but also the effective threshold (not known beforehand when using fraction or number of points) and the data used for the actual thresholding (possibly the absolute value or the negation of the input data).

Function `double_threshold`

```
def double_threshold(
    data: Sequence[float],
    secondThreshold: float,
    secondSmaller: bool = False,
    useNumpoints: bool = False,
    returnNoise: bool = False,
    maxSpacing: int = 10,
    **kwargs
) -> Union[Sequence[int], Tuple[Sequence[int], Sequence[int]]]
```

This function uses a double threshold and a simple heuristics to identify anomalies when expected to appear in groups.

It first applies a simple_threshold() (pass the arguments as for the simple_threshold function as key word arguments). As a second step it joins all the points above threshold spaced no more than maxSpacing points marking them all as potential anomalies. Finally it identifies as anomalies only the groups with at least secondThreshold points (if useNumpoints) or where the sum of the (absolute) values of the function iver the points of the group is greater (smaller) than secondThreshold.

It works only on unidimensional data.

Args

data (Sequence[float]: list, array, pd.Series): The sequence of data points. Must be a 1D sequence. **`secondThreshold`** : float : The sum of the absolute values of the sequence point in a group to qualify as anomaly.

`secondSmaller` : bool, optional If the sum shall be below threshold to qualify as anomaly. Defaults to False.
`useNumpoints` : bool, optional If to use just the number of points. Defaults to False.
`returnNoise` : bool, optional If to return a second array with the indexes of the points candidate as anomaly but not qualifying. Defaults to False.
`maxSpacing` : int, optional The maximum distance between points above the first threshold to be considered part of the same group. Half of this interval is always added at the extremes of the group. Defaults to 10.
`kwargs` Arguments for simple_threshold().

Returns

Union[Sequence[int],Tuple[Sequence[int], Sequence[int]]] An array containing the indexes of the anomalies. If returnNoise == True also the points candidate as anomaly but not qualifying are returned as a second array.

Function `isolation_forest`

```
def isolation_forest(
    data: Sequence,
    absolute: bool = False
) -> Sequence
```

This function identifies anomalies using the Isolation Forest algorithm whith a suitable set of parameters.

Uses the *Isolation Forest* (Liu, Fei Tony, Ting, Kai Ming and Zhou, Zhi-Hua. "Isolation forest." Data Mining, 2008. ICDM'08. Eighth IEEE International Conference). It works also on multidimensional data.

Args

data (Sequence: list, array, pd.Series): The data containing anomalies. **`absolute`** : bool, optional : Activates the use of absolute values. Defaults to False.

Returns

Sequence An array containing the indexes of the anomalies.

Function `local_outlier_factor`

```
def local_outlier_factor(
    data: Sequence,
    absolute: bool = False
) -> Sequence
```

This function identifies anomalies using the Local Outlier Factor algorithm with a suitable set of parameters.

Uses *Local Outlier Factor* (Breunig, Kriegel, Ng, and Sander (2000) LOF: identifying density-based local outliers. Proc. ACM SIGMOD). It works also on multidimensional data.

Args

data (Sequence: list, array, pd.Series): The data containing anomalies. **absolute** : bool, optional : Activates the use of absolute values. Defaults to False.

Returns

Sequence An array containing the indexes of the anomalies.

Function one_class_svm

```
def one_class_svm(
    data: Sequence,
    absolute: bool = False
) -> Sequence
```

This function identifies anomalies using the One-Class SVM algorithm with a suitable set of parameters.

Uses *One-class Support Vector Machine* (Estimating the support of a high-dimensional distribution Schölkopf, Bernhard, et al. Neural computation 13.7 (2001): 1443-1471.). It works also on multidimensional data.

Args

data (Sequence: list, array, pd.Series): The data containing anomalies. **absolute** : bool, optional : Activates the use of absolute values. Defaults to False.

Returns

Sequence An array containing the indexes of the anomalies.

Function robust_covariance

```
def robust_covariance(
    data: Sequence,
    absolute: bool = False
) -> Sequence
```

This function identifies anomalies using the Robust covariance algorithm with a suitable set of parameters.

Uses *Robust covariance* (Rousseeuw, P.J., Van Driessen, K. “A fast algorithm for the minimum covariance determinant estimator” Technometrics 41(3), 212 (1999)). It works also on multidimensional data.

Args

data (Sequence: list, array, pd.Series): The data containing anomalies. **absolute** : bool, optional : Activates the use of absolute values. Defaults to False.

Returns

Sequence An array containing the indexes of the anomalies.

Function simple_threshold

```
def simple_threshold(
    data: Sequence,
    absolute: bool = False,
    smaller: bool = False,
    exclude_nan: bool = True,
    fraction: float = 0.01,
    number: int = None,
    threshold: float = None
) -> Sequence
```

This function identifies anomalies by simple thresholding.

There are three different ways to provide the threshold, if more than one method is provided, the first one in this succession is applied: value, number of anomalies, fraction of anomalies. It is possible to work with absolute values and to exclude NaNs, this mainly affect the method passing fraction of anomalies by possibly reducing the total number of points.

It works only on unidimensional data.

Args

data (Sequence: list, array, pd.Series): The data containing anomalies. Must be a 1D sequence. **absolute** : bool, optional : Activates the use of absolute values. Defaults to False.

smaller : bool, optional Values below the threshold (or the lowest fraction) are considered anomalies. Defaults to False.

exclude_nan : bool, optional If True, NaN values are excluded when considering the fraction of data points to report. Defaults to True.

fraction : float, optional Fraction of data points considered as anomaly, lowest priority. Defaults to 0.01.

number : int, optional Number of data points considered as anomaly, intermediate priority. Defaults to None.

threshold : float, optional Threshold above (below) which data is considered anomaly, highest priority. Defaults to None.

Returns

Sequence An array containing the indexes of the anomalies.

Function tukeys_method

```
def tukeys_method(
    data: Sequence,
    absolute: bool = False,
    anomalyScale: float = 1.5
) -> Sequence
```

This function identifies anomalies using the Tukey's algorithm.

This is a reimplementation of the *Tukey's method* (Tukey, John W (1977). "Exploratory Data Analysis". Addison-Wesley Publishing Company). It works only on unidimensional data.

Args

data (Sequence: list, array, pd.Series): The data containing anomalies. Must be a 1-D sequence. **absolute** : bool, optional : Activates the use of absolute values. Defaults to False.

anomalyScale : float, optional The factor multiplying the interquartile distance to determine the anomalies. Defaults to 1.5.

Returns

Sequence An array containing the indexes of the anomalies.

Classes

Class scatter_cluster

```
class scatter_cluster(
    kernel: str,
    **kwargs
)
```

This class is a tool to access anomaly detection algorithms in an unified and controlled way.

It offers methods to help preprocessing data for anomaly detection in time series and allows for a single unified interface to a selection of algorithms. The currently available kernels are: IsolationForest, OneClassSVM, RobustCovariance.

Creates a scatter_cluster object with the given kernel.

Args

`kernel` **: str** The name of the kernel used for detection. Must be one of IsolationForest, OneClassSVM, RobustCovariance.

`kwargs` Arguments passed to fine tune the creation of the kernel. These may include random states or other configurable aspects.

Raises

RuntimeError In case the kernel specified at creation time is not in the list of the available ones.

Class variables

Variable `_base_features`

Convenience attribute to restore base_features if it has been modified.

Variable `_kernels`

Convenience attribute to restore kernels if it has been modified.

Variable `base_features`

The base features computed during segmentation when no user defined list is passed.

Variable `kernels`

The kernels available for anomaly detection. It is possible to add your own provided they expose the fit and predict methods.

Static methods

`Method segment`

```
def segment(
    data: pandas.core.frame.DataFrame,
    group_key: str = 'track_id',
    **kwargs
) -> pandas.core.frame.DataFrame
```

Segments a whole dataset at once. See the documentation for segment_trace.

Args

`data` **: pd.DataFrame** A dataframe as those produced by particles2d.preprocess.
`group_key` **: str, optional** The column to use for track grouping. Defaults to “track_id”.
`kwargs` Arguments for the segment_trace function.

Returns

pd.DataFrame A pd.DataFrame containing the segmented data of all traces, one row per segment.

Method `segment_trace`

```
def segment_trace(
    data: pandas.core.frame.DataFrame,
    trak_id: Optional[Any] = None,
    mwLength: int = 20,
    step: int = 5,
    basic: Sequence[str] = [],
    advanced: Sequence[Tuple[Union[Sequence[str], str], str, Callable[[numpy.ndarray], float]]]
) -> pandas.core.frame.DataFrame
```

Segment one trace for use with the anomaly detection kernel.

From every segment extracts some relevant statistic for the anomaly detection. The segmentation is carried out with a moving window of size mwLength shifting step points at the time.

It can compute basic and advanced features. Basic features are the mean and the maximum over the window for selected input features, by default it uses a list of features compatible with the output of particles2d.preprocess. Advanced features are defined by the user and can include any function of the values of a single input feature in the window. See example below.

Args

`data` **: pd.DataFrame** A pd.DataFrame as those produced by particles2d.preprocess.

`trak_id` **: Union[Any, None], optional** The id or the grouping parameter of the segmented data. Defaults to None.

`mwLength` **: int, optional** Length of the moving window for segmentation. Defaults to 20.

`step` **: int, optional** Sliding step of the window, the lower the step the more and more correlated the segments. Defaults to 5.

`basic` **: Sequence[str], optional** List of columns for which the basic mean and max values over the window is desired. Defaults to [].

`advanced` **: Sequence[Tuple[Union[Sequence[str],str], str, Callable[[np.ndarray],float]]], optional** Tuples describing advanced feature extraction. Defaults to [].

Returns

pd.DataFrame A pd.DataFrame containing the segmented trace.

Example

```
import numpy as np
from candela.functions import scatter_cluster
##### adding the standard deviation to the computed features:
advanced_feature = [("my_col", "std", lambda x: np.std(x)),]
scatter_cluster.segment_trace(data, basic=["my_col"], advanced=advanced_feature)
```

Methods

Method `fit`

```
def fit(
    self,
    data: pandas.core.frame.DataFrame,
    feature_list: Sequence[str] = None,
    **kwargs
)
```

Fits the kernel for anomaly detection.

The feature_list allows using a subset of the standard features produced by segment and ensures uniformity with prediction. The special value “all” includes all the columns in the DataFrame with the sole exception of columns

named “id”, “track_id”, “label”. Be careful if the DataFrame contains also non relevant columns or columns spuriously correlated with the labels.

Args

`data` **: pd.DataFrame** The output of segment or analogous format with a feature per column. If using custom data specify also the feature_list.

`feature_list` **: Sequence[str], optional** Lists of features to train the kernel is different from the default one, i.e., all the features from segment. Use “all” for all the columns in the DataFrame. Defaults to None.

`kwargs` Dictionary of arguments for the fit method of the kernel.

Returns

kernel The fitted estimator.

Method `predict`

```
def predict(
    self,
    data: pandas.core.frame.DataFrame,
    returnLabels: bool = False,
    **kwargs
) -> numpy.ndarray
```

Predicts if the points are anomalies or not.

It uses the same feature list as the fit method.

Args

`data` **: pd.DataFrame** The output of segment or analogous format with a feature per column.

`returnLabels` **: bool, optional** Instead of returning the indexes of the anomalies, return a vector of labels. Defaults to False.

`kwargs` Dictionary of arguments for the predict method of the kernel.

Returns

np.ndarray Same len as data, contains the labels for every segment.

Module `candela.particles2d`

This submodule provides a set of utility function for anomaly dectection on 2D particle traces.

The main content of this submodule is the preprocess() function. The minimum data required is just the list of x and y positions of the particle. However, it offers much more flexibility allowing for interpolation of missing data with improved smoothing.

The other functions of this submodule are focused on reporting to help the user getting an idea of the content of their dataset.

Preprocessing a trace can be as easy as:

```
import pandas as pd
from candela.particles2d import preprocess
df = pd.read_csv("my_trace.csv")
trace = preprocess(df, track_id=0)
```

In more common scenarios where the dataframe contain multiple traces, when they are indexed by a track_id column, the full dataset is preprocessed with:

```
df = pd.read_csv("my_dataset.csv")
preprocessed_traces = df.groupby("track_id", as_index=False).apply(preprocess, fps=24, scale=2)
```

Where we also specify a framerate of the acquisition of 24 frames per second and a factor 2 in the spatial scale (1 unit in the data corresponds to 2 m).

Even before preprocessing the whole dataset, it may be convenient to get a first overview of the data. The function anomaly_summary() serves this purpose:

```
from candela.particles2d import anomaly_summary
df = pd.read_csv("my_dataset.csv")
sample = df[df.track_id==sample_id]
anomaly_summary(sample, sample_id, fps=24, scale=2)
```

Functions

Function `__angular_features`

```
def __angular_features(
    vx: numpy.ndarray,
    vy: numpy.ndarray,
    vtot: numpy.ndarray = None,
    filterVel: bool = False,
    threshold: float = 0.1
) -> Tuple[numpy.ndarray, numpy.ndarray]
```

Computes angular features related to the heading.

Angles are in degrees in (-180, +180). Angular speed is computed such that moving from -179 to +179 gives a difference of 2°.

Args

`vx` **: np.ndarray** Velocity on x.
`vy` **: np.ndarray** Velocity on y
`vtot` **: np.ndarray, optional** Total velocity, if not provided and filterVel is True must be recomputed. Passing this argument may speed up the computation. Defaults to None.
`filterVel` **: bool, optional** If True replaces the computed headings with NaN when the speed is too low. Defaults to True.
`threshold` **: float, optional** Threshold for the total speed below which the angle is filtered out. Defaults to 0.1 assuming km/h.

Returns

Tuple[np.ndarray, np.ndarray] Two arrays containing the heading and the angular speed (degrees per frame).

Function `__interpolate_xy`

```
def __interpolate_xy(
    df: pandas.core.frame.DataFrame,
    scale: float = 1
) -> Tuple[numpy.ndarray, numpy.ndarray]
```

Given a list of observations for an ID, interpolates linearly in the missing frames.

Args

`df` **: pd.DataFrame** Dataframe containing the observations
`scale` **: float, optional** Scale for the distances. Defaults to 1.

Returns

Tuple[np.ndarray, np.ndarray] Two arrays containing the interpolated positions

Function `__smooth_speeds`

```
def __smooth_speeds(
    pos: Sequence,
    window: int
) -> Tuple[numpy.ndarray, numpy.ndarray]
```

Smoothing with exponentially weighted mean.

To reduce the bias the function is applied once forwards and once backwards. This may introduce dependency from the future in causal estimation.

Args

`pos` **: Sequence** Sequence to be smoothed
`window` **: int** Amplitude of the smoothing window (decay of the weight in terms of frames).

Returns

Tuple[np.ndarray, np.ndarray] A pair of arrays containing the smoothed values and the residuals.

Function `anomaly_summary`

```
def anomaly_summary(
    df: pandas.core.frame.DataFrame,
    track_id: int = None,
    **kwargs
)
```

Utility function providing an overview of the anomalies in a track.

Args

`df` **: pd.DataFrame** pd.DataFrame containing the data of a single track.
`track_id` **: int, optional** Id of the processed track if df does not contain a track_id column. Defaults to None.
`kwargs` Arguments passed to the preprocess() function, see preprocess().

Function `basic_stats`

```
def basic_stats(
    df: pandas.core.frame.DataFrame
)
```

This function prints information extracted from the output of preprocess.

Args

`df` **: pd.DataFrame** Output of the preprocess function.

Function `preprocess`

```
def preprocess(
    df: pandas.core.frame.DataFrame,
    track_id: int = None,
    sequential_frames: bool = False,
    filterVel: bool = False,
    window: int = 3,
    fps: float = 10,
    scale: float = 1,
    kmh: bool = True,
    startTime: float = 1356088042.042042
) -> pandas.core.frame.DataFrame
```

Preprocess the tracks of particles in a 2D space for further analysis and anomaly detection.

The input DataFrame must contain at least the following columns:

- `x, y: coordinates of the particle for every time point.`

It is suggested to include also:

- `frame_id: time point id of the measured values, allowing for missing frames;`
- `track_id: particle id of the measured values.`

The preprocessind includes three steps:

1. scaling of the positions and interpolation in case of missing time points (assuming constant speed);
2. computation of the instantaneous speed, smoothing with double pass exponential window;
3. reconstruction of positions given the smoothed speed and computation of additional statistics as the heading angle and acceleration.

Args

`df` : pd.DataFrame Input data
`track_id` : int, optional If provided, supersedes the eventual track_id column in df.
`sequential_frames` : bool, optional If True ignores frame_id and reindexes the frames. Defaults to False.
`filterVel` : bool, optional If True replaces the computed headings with NaN when the speed is too low. Defaults to False.
`window` : int, optional Decay length in time frames of the smoothing applied to speeds. Defaults to 3.
`fps` : float, optional Frames per second of data, needed to get actual speeds. Defaults to 10.
`scale` : float, optional Scale of the positions, assumed equal over x and y. Defaults to 1.
`kmh` : bool, optional Use km/h as unit for speeds. Defaults to True.
`startTime` : float, optional Timestamp of the first frame. Defaults roughly to the end of the world according to the Mayas.

Returns

pd.DataFrame Interpolated and smoothed data with additional measures (heading, speed, total speed, angular velocity).

Module `candela.plotting`

This submodule includes functions for plotting and visualisation of anomalies.

The submodule exposes two functions: plot_features() and show_trajectories2D(). The first is dedicated to plotting graphs with one or more features and the option of overlaying detected anomalies. The second is for the representation of spatial data in 2D, with a possible color coded third dimension, and again with the possible overlay of anomalies.

The functions make easy the integration with matplotlib (for as much the integration of anything with matplotlib can be easy). Combining efforts with the other two submodules of candela it is possible to obtain a complex two panles figure in four lines writing:

```
import pandas as pd
from candela.functions import local_outlier_factor
from candela.particles2d import preprocess
from candela.plotting import plot_features, show_trajectories2D

df = pd.read_csv("my_trace.csv")
trace = preprocess(df, track_id=0)

outliers_velocity_LOF = local_outlier_factor(trace.vtot)

fig, axs = plt.subplots(1, 2, figsize = (12,7))
plot_features('frame_id','vtot',outliers=outliers_velocity_LOF, outlierLabels="Anomalies", data=trac
```

```
show_trajectories2D('x','y','vtot',outliers=outliers_velocity_LOF, data=track, outlier_names="Velocity LOF")
plt.show()
```

Functions

Function `__2Dplot`

```
def __2Dplot(
    x: Sequence[float],
    y: Sequence[float],
    t: Sequence[float],
    outliers: Mapping[str, Sequence[float]],
    tistime: bool,
    xName: Optional[str] = None,
    yName: Optional[str] = None,
    cbarName: Optional[str] = None,
    ax: Optional[matplotlib.axes._axes.Axes] = None,
    vmax: Optional[float] = None,
    vmin: Optional[float] = None,
    forceCbar: bool = False,
    equalAspectRatio: bool = False
) -> Tuple[matplotlib.figure.Figure, matplotlib.axes._axes.Axes]
```

Does the actual plotting for show_trajectories2D().

Args

`x` **: Sequence[float]** x values.
`y` **: Sequence[float]** y values.
`t` **: Sequence[float]** Values for color coding.
`outliers` **: Mapping[str, Sequence[float]]** Dictionary containing anomalies labels and indexes.
`tistime` **: bool** Determine id the color coding id to be treated as a timestamp.
`xName` **: Union[str, None], optional** Name for the x axis. Defaults to None.
`yName` **: Union[str, None], optional** Name for the y axis. Defaults to None.
`cbarName` **: Union[str, None], optional** Name for the colorbar. Defaults to None.
`ax` **: Union[plt.Axes, None], optional** An instance of matplotlib.axis.Axis to plt into. Defaults to None.
`vmax` **: Union[float, None], optional** Upper limit of the colorbar. Defaults to None.
`vmin` **: Union[float, None], optional** The lower limit of the colorbar. Defaults to None.
`forceCbar` **: bool, optional** Force the creation of a colorbar even when the ax argument is passed. Defaults to False.
`equalAspectRatio` **: bool, optional** Force equal aspect ratio of the plot, prevents stretching of spatial data.. Defaults to False.

Returns

Tuple[Figure, Axes] Matplotlib figure and axes containing the plot.

Function `__add_outliers`

```
def __add_outliers(
    x: Sequence[float],
    y: Sequence[float],
    outl: Tuple[Sequence[float], str, Sequence[float]],
    colors: Tuple[Any, Any] = None
) -> Tuple[matplotlib.lines.Line2D, str, matplotlib.collections.PathCollection]
```

Overlays the anomalous points over the features.

Args

`x` **: Sequence[float]** x coordinates of the points.

`y` **: Sequence[float]** y coordinates of the points.
`outl` **: Tuple[Sequence[float], str, Sequence[float]]** The anomalies to plot represented by the indexes corresponding to anomalous points, a label for the legend and the terminal points of anomalous intervals if needed.
`colors` **: Tuple[Any, Any], optional** Colours of outliers line and markers, useful to force a single colour for all the outliers. Defaults to None.

Returns

Tuple[Line2D, str, PathCollection] A tuple containing the information for adding the anomalies to the legend (handle for the anomaly line, anomaly label, handle for the points markers).

Function `__check_columns`

```
def __check_columns(
    data: pandas.core.frame.DataFrame,
    y: Sequence[str]
) -> None
```

Verify that a list of names are among the columns of a pd.DataFrame.

Useful to give a meaningful error in case several are missing.

Args

`data` **: pd.DataFrame** The DataFrame to check.
`y` **: Sequence[str]** The desired columns.

Function `__determine_handles_labels`

```
def __determine_handles_labels(
    len_feat: int,
    len_outl: int,
    feat_plot: Sequence[Tuple[matplotlib.lines.Line2D, str]],
    out_plot: Sequence[Tuple[matplotlib.lines.Line2D, str, matplotlib.collections.PathCollectio
) -> Tuple[Sequence, Sequence[str]]
```

Determine the handles and labels for the legend associated to __draw_features().

The number of elements and their content is determined based on the kind of plot as described for plot_features().
To reduce the space taken up by the legend, uses a single entry for feature and anomaly when these are paired.

Args

`len_feat` **: int** Number of features plotted.
`len_outl` **: int** Number of anomalies plotted.
`feat_plot` **: Sequence[Tuple[Line2D, str]]** A sequence of tuples containing a Line2D and a label for every feature plotted.
`out_plot` **: Sequence[Tuple[Line2D, str, PathCollection]]** A tuple containing the information for adding the anomalies to the legend (handle for the anomaly line, anomaly label, handle for the points markers).

Returns

Tuple[Sequence, Sequence[str]] A tuple containing the sequences of handles and labels.

Function `__draw_features`

```
def __draw_features(
    x: Tuple[Sequence[float], str],
    features: Sequence[Tuple[Sequence[float], str, Mapping[str, Any]]],
    outliers: Sequence[Tuple[Sequence[float], str, Sequence[float]]],
    ax: Optional[matplotlib.axes._axes.Axes] = None
) -> Tuple[matplotlib.figure.Figure, matplotlib.axes._axes.Axes]
```

Actually plot the features with the anomalies.

It receives the input from the higher level plot_features() and generates the plot.

Args

`x` **: Tuple[Sequence[float], str]** The values of x and the label for the x axis.

`features` **: Sequence[Tuple[Sequence[float], str, Mapping[str, Any]]]** A sequence of features to plot. Every feature is represented as the y values, a label for the legend and a dictionary of kwargs for the plotting function.

`outliers` **: Sequence[Tuple[Sequence[float], str, Sequence[float]]]** A sequence of anomalies to plot. Every anomaly is represented by the indexes corresponding to anomalous points, a label for the legend and the terminal points of anomalous intervals if needed.

`ax` **: Union[plt.Axes, None], optional** An instance of matplotlib.axes.Axes used to plot the graph. Defaults to None.

Returns

Tuple[Figure, Axes] Matplotlib figure and axes containing the plot.

Function `__edges`

```
def __edges(
    labels: Sequence[bool]
) -> Sequence[int]
```

Identify the edges of intervals of consecutive True values.

The returned array contains the indices of the first and last True value of every interval and the indices of isolated ones.

Args

`labels` **: Sequence[bool]** An array containing True and False labels.

Returns

Sequence[bool] The indices of the edges.

Function `__index_from_vec`

```
def __index_from_vec(
    vec: Sequence[bool]
) -> Sequence[int]
```

Return the indexes of the true values.

Args

`vec` **: Sequence[bool]** Binary array containing class labels.

Returns

Sequence[int] Array containing the indices of the True elements.

Function `__to_sequence`

```
def __to_sequence(
    y: Any
) -> Sequence[Any]
```

Ensure that an object is a sequence of sequences.

It receives a sequence or none as input and ensures that the output is an empty list or a sequence of sequences.

If the input is not a sequence is returned as is for future error management.

Args

`y` : Any Object that must become a sequence of sequences.

Returns

Sequence[Any] A sequence of sequences.

Function `__vec_from_index`

```
def __vec_from_index(
    ind: Sequence[int],
    size: int
) -> Sequence[bool]
```

Return an array of labels.

Args

`ind` : Sequence[int] The indices of the True elements.
`size` : int The length of the output array.

Returns

Sequence[bool] An array containing ones in the positions pointed by ind.

Function `plot_features`

```
def plot_features(
    x: Union[Sequence, str],
    y: Union[Sequence[Sequence], Sequence[str]],
    outliers: Union[str, Sequence[str], Sequence[int], Sequence[bool], ForwardRef(None)] = None
    featureLabels: Union[str, Sequence[str], ForwardRef(None)] = None,
    featureKW: Sequence[Mapping[str, <built-in function any>]] = [],
    outlierLabels: Union[str, Sequence[str], ForwardRef(None)] = None,
    useOutlierLabelArray: Union[bool, str] = 'auto',
    showPoints: bool = False,
    data: Optional[pandas.core.frame.DataFrame] = None,
    ax: Optional[matplotlib.axes._axes.Axes] = None
) -> Tuple[matplotlib.figure.Figure, matplotlib.axes._axes.Axes]
```

High level function for plotting features, possibly with detected anomalies.

It can be used in different ways depending on the input:

1. it can plot features alone allowing for some fine tuning of the curve appearance;
2. it can highlight a single set of anomalies on all features, with or without showing the fist and last point of every anomalous tract. In this case it also shows a point for every isolated anomaly that would be otherwise hidden.
3. it can show a different set of anomalies for every feature.
4. it can show a single feature with overlaid different sets of anomalies.

Args

`x` : Union[Sequence, str] The values of x, these are the same for all the features, can be the name of a column in data.

`y` **: Union[Sequence[Sequence], Sequence[str]]** A sequence of sequences of values for different features. The inner sequences can be replaced with the names of columns in data.

`outliers` **: Union[str, Sequence[str], Sequence[int], Sequence[bool], None], optional** Sequences of detected anomalies to be plotted on the features. The inner sequences can be replaced with the names of columns in data. Defaults to None.

`featureLabels` **: Union[str, Sequence[str], None], optional** Label or list of labels for the features. If provided should be as many as the features, overrides the column names. Defaults to None.

`featureKW` **: Sequence[Mapping[str, any]], optional** Dictionaries of keywords to adjust the properties of every feature curve. If provided should contain as many dictionaries (even empty) as the features. Defaults to [].

`outlierLabels` **: Union[str, Sequence[str], None], optional** Labels to represent the anomalies in the legend. If provided should be as many as the oultiers, overrides the column names if passed. Defaults to None.

`useOutlierLabelArray` **: Union[bool, str], optional** If True teat the vectors containing the anomalies info as arrays of labels. Otherwise the vectors are supposed to contain the indexes of the anomalies. The keyword "auto" activates a simple heuristics to determine the correct handling. Defaults to "auto".

`showPoints` **: bool, optional** Shows fist and last point of every anomalous tract. It also shows a point for every isolated anomaly that would be otherwise hidden. Defaults to False.

`data` **: Union[pd.DataFrame, None], optional** An optional dataframe containing data. If passed it's possible to specify x, y and outliers with strings referring to columns in data. Defaults to None.

`ax` **: Union[plt.Axes, None], optional** An instance of matplotlib.axes.Axes used to plot the graph. Defaults to None.

Returns

Tuple[Figure, Axes] Matplotlib figure and axes containing the plot.

Function `show_trajectories2D`

```
def show_trajectories2D(
    x: Union[Sequence, str],
    y: Union[Sequence, str],
    t: Union[Sequence, str, ForwardRef(None)] = None,
    outliers: Optional[Sequence[Sequence[int]]] = None,
    useOutlierLabels: bool = False,
    data: Optional[pandas.core.frame.DataFrame] = None,
    tistime: bool = False,
    outlier_names: Union[str, Sequence[str], ForwardRef(None)] = None,
    cbarName: Optional[str] = None,
    ax: Optional[matplotlib.axes._axes.Axes] = None,
    vmax: Optional[float] = None,
    vmin: Optional[float] = None,
    forceCbar: bool = False,
    equalAspectRatio: bool = False
) -> Tuple[matplotlib.figure.Figure, matplotlib.axes._axes.Axes]
```

Plot trajectories in 2D with color coding and anomalies.

Plot trajectories in 2D encoding a third dimension (e.g, time, speed) as color. It is possible to plot a number of detected anomalous points over the trajectory. Up to four different anomalies are supported.

The function provides special support for timestamp based color coding.

It is possible to plot more than one trace on the same plot by passing an instance of matplotlib.axis.Axis to the function. In this case a new colorbar is not generated (unless explicitly requested). To ensure that all the traces share the same color coding, pass the vmax and vmin limits of the colorbar.

Args

`x` **: Union[Sequence, str]** x coordinates of the trajectory or name of a column in data.

`y` **: Union[Sequence, str]** y coordinates of the trajectory or name of a column in data.

`t` **: Union[Sequence, str, None], optional** Values for the colour coding or name of a column in data. Defaults to None.

`outliers` **: Union[Sequence[Sequence[int]], None], optional** A sequence of sequences of indexes representing the anomalous points. Defaults to None.

`useOutlierLabels` **: bool, optional** If this flag is True the outliers sequences are considered to be labels for every point marking th anomalies with a true value. Defaults to False.

`data` **: Union[pd.DataFrame, None], optional** If x, y or t are strings, the function looks for data in the columns of this DataFrame. Defaults to None.

`tistime` **: bool, optional** A flag to signal that the t sequence contains times for proper formatting. Defaults to False.

`outlier_names` **: Union[str, Sequence[str], None], optional** Sequence of strings providing names for the anomalies for the legend. Use the special value "data" to signal that the outliers contains the names of columns in data. Defaults to None.

`cbarName` **: Union[str, None], optional** The label for the colorbar describing the color coding. Defaults to None.

`ax` **: Union[plt.Axes, None], optional** An instance of matplotlib.axis.Axis to plt into. Defaults to None.

`vmax` **: Union[float, None], optional** Upper limit of the colorbar. Defaults to None.

`vmin` **: Union[float, None], optional** The lower limit of the colorbar. Defaults to None.

`forceCbar` **: bool, optional** Force the creation of a colorbar even when the ax argument is passed. By default when the ax is passed a colorbar is not generated to avoid having multiple colorbars in the same plot. Defaults to False.

`equalAspectRatio` **: bool, optional** Defaults to False.

Returns

Tuple[Figure, Axes] Matplotlib figure and axes containing the plot.
